Materializzazione del Canale: Un Nuovo Modello per la Riflessione. Analisi e Confronto con gli altri Modelli ed una loro Applicazione alla Fault Tolerance.

Walter Cazzola

15 Aprile 1996

I sogni sono duri a morire, l'unico modo per ucciderli è smettere di crederci.

... A Sabrina, ed ai miei Genitori.

Ringraziamenti

Il lavoro di ricerca svolto per questa tesi è frutto della mia collaborazione con il Professore Massimo Ancona ed il suo gruppo di Ricerca.
Vorrei, quindi, ringraziare il Professore Massimo Ancona, la Dottoressa Gabriella Dodero e la Dottoressa Vittoria Gianuzzi per il tempo che mi hanno dedicato ed i loro preziosi consigli.

Un doveroso ringraziamento va anche al Professore Giovanni Chiola per i suoi consigli che hanno rappresentato un apporto prezioso per lo svolgimento di questo lavoro.

Ringrazio, anche, i miei genitori ed i miei amici per il loro supporto morale per l'odissea rappresentata dalla stesura di questa tesi, in particolare devo ringraziare la mia ragazza Sabrina Boaretto, che mi ha pazientemente sopportato e moralmente aiutato in tutto questo tempo e senza la quale non ci sarebbe stata neanche questa tesi.

That's all, Folks

Indice

Elenco delle figure

Introduzione

Descrizione ed Importanza delle Riflessioni

Una delle proprietà principali dei linguaggi object-oriented è l'astrazione: un oggetto è libero di impersonare il proprio ruolo all'interno del sistema in qualunque modo desideri. Risulta, quindi, naturale pensare che un oggetto non solo elabori la computazione sul proprio dominio, ma che possa definire anche il modo in cui si realizza tale computazione.

Con il termine *riflessione computazionale* o più semplicemente *riflessione* si intende l'attività svolta da un sistema computazionale relativamente alla propria computazione.

I progettisti di linguaggi object-oriented hanno sentito la necessità di introdurre le riflessioni nei propri linguaggi per risolvere in modo adeguato tutti i problemi computazionali. È chiaro che uno specifico linguaggio di programmazione object-oriented risulta adatto a risolvere efficacementee solo una certa gamma di problematiche e non tutte.
Alcuni linguaggi di object-oriented, come SmallTalk, hanno cercato di ovviare a queste carenze. Questi linguaggi presentano però un'"apertura" limitata, rappresentata da un insieme ridotto di caratteristiche riflessive, introdotte in modo "inconsapevole", cioè senza un riferimento esplicito al concetto di riflessione. L'aggiunta di nuove caratteristiche comporta il cambiamento dell'interprete stesso. Un secondo limite è rappresentato dall'integrazione dell'aspetto riflessivo con l'aspetto computazionale, che conduce ad una scarsa chiarezza del codice prodotto (le classi, in SmallTalk, contengono informazioni sul dominio dei dati (attributi) e su come questi vengono rappresentati (nome degli attributi), questa promiscuità genera confusione).

Le riflessioni forniscono la soluzione a questo problema. Un linguaggio con caratteristiche riflessive è detto aperto, in quanto le riflessioni permettono di costruire interpreti specializzati del linguaggio all'interno del linguaggio stesso. Ad esempio, le classi potrebbero specificare una rappresentazione, personalizzabile da istanza ad istanza, di come le istanze vengono visualizzate o create.

Un linguaggio object-oriented dotato di architettura riflessiva, cioè che riconosce le riflessioni come una componente base del linguaggio gode delle seguenti proprietà:

- Estendibilità: il linguaggio ottenuto è estendibile (o aperto) in quanto è possibile definire, tramite le riflessioni, nuovi comportamenti per le componenti del linguaggio (attivazione dei metodi, accesso agli attributi, istanziazione, persistenza, concorrenza, ...), ottenendo un linguaggio diverso da quello di partenza senza dover costruire un nuovo interprete|compilatore.
- Separazione dell'aspetto "funzionale" dall'aspetto "non funzionale" del sistema computazionale: il sistema viene scomposto in più livelli logici (livello base, meta-livello, ...). Le entità appartenenti al livello base compongono l'applicazione vera e propria espletandone l'aspetto funzionale. Le meta-entità (cioè le entità operanti nei restanti livelli) gestiscono il comportamento riflessivo, che è un comportamento aggiuntivo, non funzionale alla risoluzione del problema.
- Trasparenza: i comportamenti riflessivi, definiti grazie alle proprietà di estendibilità del sistema, sono utilizzati trasparentemente dal programmatore dell'aspetto funzionale del sistema computazionale.

Applicazioni delle Riflessioni

Le riflessioni hanno diverse applicazioni. Vediamo subito un'applicazione banale: si può definire una meta-entità con l'incarico di tenere statistiche sull'utilizzo delle istanze di una classe fissata. L'applicazione più significativa, comunque, è quella di estendere i linguaggi di programmazione con nuovi comportamenti. A titolo di esempio si riportano le comunità di oggetti definite in [Chiba-Masuda 93], che introducono in OpenC++ la possibilità di costruire sistemi computazionali distribuiti, l'introduzione dell'ereditarietà multipla in 3-KRS descritta in [Maes 87] e l'introduzione della persistenza in CLOS descritta in [Lee-Zachary 94].
Le riflessioni sono state utilizzate per gestire|risolvere diverse problematiche dei sistemi concorrenti. Ad esempio in [Matsuoka-Watanabe-Yonezawa 91] viene modellato, grazie alla riflessione il problema di schedulazione per la simulazione in ambiente parallelo di eventi discreti noto come *time warp algorithm*. In [Matsuoka-Yonezawa 90] viene proposta una soluzione riflessiva per la nota anomalia dell'ereditarietà che affligge i sistemi concorrenti ad oggetti. Un'implementazione riflessiva della migrazione degli oggetti è riportata in [Watanabe-Yonezawa 90]. Le riflessioni sono state utilizzate nel sistema operativo MUSE [Yokote-Mitsuzawa-Fujinami-Tokoro 91] per poter introdurre la possibilità di modificare dinamicamente il comportamento del sistema. Altre applicazioni, non ancora adeguatamente approfondite in letteratura, consistono nello sfruttare il monitoraggio offerto dalle meta-entità per realizzare dei meta-debugger. Le riflessioni offrono la possibilità di controllare ogni azione effettuata dal

sistema computazionale, in questo modo è il sistema stesso che può effettuare del debugging sul proprio codice, valutare le strategie di correzione e sostituire, durante l'esecuzione, nuovo codice a quello che origina l'errore. In modo analogo è stato costruito un sistema riflessivo basato sulle asserzioni in cui il programmatore definisce invarianti, pre e post condizioni di cui le meta-entità, a run-time, controllano la violazione effettuando il trattamento delle eccezioni [Sturman 94].
Per ultima, ma non perché meno importante, è la possibilità di utilizzare le riflessioni per dotare un sistema computazionale di comportamento fault tolerant. L'applicazione della riflessione alla tolleranza ai guasti è stata presa in considerazione in diversi lavori, a partire da [Ancona-Clematis-Dodero-Fernandez-Gianuzzi 90] in cui è definita , per la prima volta, un'architettura fault tolerant (Recovery Meta Program) strutturata su più livelli, e in [Ancona-Clematis-Dodero-Gianuzzi-Lisboa 95b], [Ancona-Cazzola-Dodero-Gianuzzi 95], [Fabre-Nicomette-Pérennou-Stroud-Wu 95] e [Agha-Frølund-Panwar-Sturman 93] che utilizzano architetture effettivamente riflessive.

Descrizione ed Importanza della Fault Tolerance

La tolleranza di fallimenti a cui può andare incontro un sistema software è un problema che ricopre un ruolo tanto più importante quanto più è critico l'operato del sistema. È sufficiente pensare ai sistemi per la guida automatica degli aerei e dei treni, ai sistemi che mantengono sotto controllo gli impianti di fusione di una centrale nucleare o che mantengono in vita un paziente costretto in un polmone di acciaio, per rendersi conto di quanto potrebbe costare caro un fallimento.
I fallimenti possono essere dovuti ad errori di progettazione del software od a fallimenti dell'hardware; le principali politiche di fault tolerance si basano sulla ridondanza. La ridondanza presente nelle politiche di tolleranza ai guasti si manifesta nella replicazione delle componenti, utile per gestire errori dovuti all'hardware o nel versionamento delle componenti, utile per superare errori di progettazione del software.

Le politiche per la gestione della tolleranza dei fallimenti fino ad oggi sono risultate troppo costose per trovare impiego su larga scala. I costi sono dovuti:

- Alla difficoltà di inserimento delle politiche di fault tolerance all'interno di sistemi software già esistenti, che può obbligare ad una completa riscrittura del sistema stesso.

- Alla necessità di progettare e quindi produrre diverse copie della stessa componente, se non è una copia identica di altre componenti già prodotte.

- Alla scarsa o nulla riusabilità delle diverse politiche di fault tolerance. Quasi sempre il codice sviluppato per la fault tolerance risulta troppo integrato con il codice applicativo, per permetterne la separazione ed il riuso.

Motivazioni della Tesi

Questa tesi trae spunto da due obiettivi: analisi delle riflessioni e loro applicazione alla fault tolerance.

L'analisi delle riflessioni ha l'obiettivo di determinare|progettare la "migliore" architettura riflessiva per un linguaggio object-oriented e di determinare un meccanismo per l'introduzione della riflessione in un sistema software concorrente.
Il modello, di cui si vedrà una realizzazione in diversi linguaggi, non è fine a se stesso, ma serve per realizzare architetture fault tolerant riflessive.

L'idea di integrare le riflessioni con le politiche di fault tolerance nasce con l'obiettivo di superare i problemi della fault tolerance sopra citati favorendone, quindi, l'utilizzo.

Come vedremo l'uso delle riflessioni permette di separare l'aspetto non funzionale (cioè il codice dell'algoritmo per la gestione della fault tolerance) dall'aspetto funzionale dell'applicativo, aumentandone la modularità. Il codice della politica di fault tolerance viene incapsulato in apposite entità (le meta-entità) che possono essere riusate in applicazioni diverse.
La proprietà di estendibilità dei sistemi riflessivi unita alle proprietà di trasparenza delle computazioni riflessive consente di introdurre agevolmente le politiche di tolleranza ai guasti, in un sistema software già esistente, riducendo al minimo le modifiche da effettuare e rendendo possibile riutilizzarle da altri progetti.

Struttura della Tesi

La tesi è suddivisa in due parti. Nella prima parte (capitoli ①, ② e ③) si esaminano le riflessioni nei linguaggi object-oriented sia da un punto di vista modellistico che implementativo. Nella seconda parte (capitoli ④ e ⑤) si discutono alcune politiche di fault tolerance ed i vantaggi apportati dalle riflessioni alla costruzione di sistemi software tolleranti i guasti.

Parte 1: Riflessioni e Linguaggi Object-Oriented

Capitolo ①: Riflessioni

In questo capitolo vengono introdotte le riflessioni da un punto di vista modellistico. Si analizzano pregi, difetti e potenzialità dei principali modelli di riflessione: modello a meta-classe, modello a meta-oggetto e modello a materializzazione del messaggio. I diversi modelli vengono analizzati basandosi su linguaggi object-oriented non tipati e ne viene costruito un prototipo di implementazione in SmallTalk. L'analisi che si effettua porta alla definizione di un nuovo modello di riflessione: il *modello a materializzazione del canale*. Questo nuovo modello nasce per sfruttare le potenzialità del modello a materializzazione del messaggio superandone i limiti.

Capitolo ②: Riflessioni e Linguaggi Fortemente Tipati

In questo capitolo si esaminano i problemi implementativi che si incontrano nel costruire un'architettura riflessiva nell'ambito di un linguaggio object-oriented fortemente tipato. I linguaggi esaminati sono C++ e Oberon, mentre l'analisi è riferita ai modelli a meta-oggetto ed a materializzazione del canale.

L'analisi ha permesso di determinare un meccanismo riusabile (la terna di metodi send-HandleMessage-apply), indipendente dal linguaggio di programmazione e dal modello di riflessione, che agevola la costruzione di sistemi software riflessivi.

Il meccanismo definisce un'interfaccia univoca degli oggetti riflessivi e delle meta-entità separando, chiaramente, le operazioni di cambio di contesto (shift-up e shift-down) dalla meta-computazione vera e propria.

Capitolo ③: Riflessioni in Ambiente Concorrente

In questo capitolo, l'analisi del capitolo precedente viene estesa all'ambiente concorrente. In particolare si esaminano i problemi dovuti alla sincronizzazione delle operazioni delle meta-entità con quelle degli oggetti controllati (referenti) e diverse possibili soluzioni al problema. Inoltre viene esaminato dettagliatamente il linguaggio OpenC++ ed il suo approccio alla concorrenza. Il linguaggio OpenC++ definito in [Chiba 93] costruisce un'architettura riflessiva basata sul modello a meta-oggetto nel linguaggio C++. Per quanto riguarda il nuovo modello presentato si studia la realizzazione, in C++ e PVM, di un'architettura riflessiva in ambiente concorrente basata sui canali. Infine si è abbozzato uno studio sulla costruzione di un'architettura riflessiva nel sistema distribuito ad oggetti CORBA.

Parte 2: Riflessioni e Fault Tolerance

Capitolo ④: Fault Tolerance

In questo capitolo vengono affrontate le diverse politiche di fault tolerance presentando un breve escursus sulle più note. Si è presentata, inoltre, una classificazione delle possibili politiche basata sul tipo di errori che vengono gestiti e le motivazioni che portano ad utilizzare le riflessioni per costruire sistemi software tolleranti i guasti.

Capitolo ⑤: Riflessioni Applicate alla Fault Tolerance

In questo capitolo si affronta, da un punto di vista modellistico, il problema dell'integrazione delle riflessioni alle politiche di tolleranza ai guasti nella costruzione di sistemi software. La modellazione è rivolta soltanto alla system fault tolerance. In particolare si sceglie di modellare, usando

meta-oggetti e canali, le politiche di system fault tolerance basate sulla replicazione.
La modellazione che si fornisce relativamente all'uso dei meta-oggetti è un approfondimento del lavoro di Fabre et al. sull'argomento e presentato in [Fabre-Nicomette-Pérennou-Stroud-Wu 95]. Le modellazioni ottenute con meta-oggetti e canali vengono confrontate nel caso particolare della replicazione passiva.

Nelle conclusioni vengono riassunti i risultati ottenuti ed illustrati alcuni problemi aperti e sviluppi futuri correlati a questa tesi.

Parte I
Riflessioni e Linguaggi Object-Oriented

1 Riflessioni

1.1 Introduzione

Riflessione Termine indicante l'attività del pensiero nei confronti di un determinato contenuto, e soprattutto nei confronti dell'azione stessa del pensare, considerata come oggetto della riflessione, che si identifica quindi con l'autocoscienza.

È comune convinzione credere che se si fornisse ai sistemi computazionali tale capacità, questi sarebbero estremamente flessibili e conseguentemente le loro funzionalità ne risulterebbero avvantaggiate; è sufficiente pensare alle possibilità offerte da un sistema computazionale in grado di risolvere situazioni critiche autonomamente, senza l'ausilio del programmatore.
Evidentemente dotare un sistema computazionale di "autocoscienza" e quindi della possibilità di effettuare osservazioni riflessive equivale al fornirgli i mezzi per monitorare il proprio comportamento ed eventualmente modificarlo.
Viste le enormi potenzialità offerte dalla riflessione, quest'ultime sono state applicate a qualsiasi modello computazionale, sia esso procedurale, deduttivo, orientato agli oggetti o altro; per una panoramica più dettagliata dei linguaggi che le implementano si rimanda al lavoro svolto da Pattie Maes in [Maes 87].
Con il termine riflessione computazionale si intende l'attività effettuata da un sistema computazionale quando esegue delle elaborazioni relative al proprio stato computazionale.

1.2 Riflessioni : Queste Sconosciute

Seguendo quanto detto da P. Maes [Maes 87] vediamo i concetti base delle riflessioni computazionali.

def. *Sistema Computazionale* è un sistema basato sui computer, il cui scopo è quello di rispondere a domande e|o supportare azioni riguardanti un certo dominio.

def. *Un sistema computazionale è autocosciente in modo causalmente connesso,* se incorpora delle strutture rappresentanti il dominio in cui opera e tali strutture sono collegate al dominio che rappresentano, in modo tale che una loro modifica provoca una modifica nel dominio ed una modifica a quest'ultimo si riperquote sulle strutture aggiornandole.

def. *Sistema Riflessivo* è un sistema computazionale che è autocosciente in modo causalmente connesso.

def. *Riflessione Computazionale* o più semplicemente *Riflessione* è il comportamento esibito da un sistema computazionale riflessivo.

Quindi è importante sottolineare che un sistema causalmente connesso al suo dominio ha sempre una accurata rappresentazione del suo dominio e può effettivamente causare un cambiamento nel suo dominio semplicemente effettuando delle elaborazioni.

---

esempio n^o 1.1 ***Sistema computazionale causalmente connesso.***

Consideriamo un sistema computazionale che gestisce gli accessi ad un sito Internet che permette il collegamento di non più di un certo numero fissato di utenti anonimi contemporaneamente, il cui tempo a disposizione, per poter accontentare tutti, è limitato ad un breve periodo dopo il quale vengono automaticamente scollegati; tale sistema ha al suo interno le strutture e le operazioni necessarie per gestire tale situazione.
Se un utente anonimo riesce a collegarsi (causa esterna) allora la struttura interna viene aggiornata conseguentemente; invece, nel momento in cui un collegamento raggiunge il tempo limite prefissato, il sistema aggiorna la struttura e tale utente si trova scollegato (causa interna).
Questo quindi è un sistema computazionale causalmente connesso ed è consapevole della sua attività e come tale è anche un sistema riflessivo.

Ricapitolando un sistema riflessivo è un sistema computazionale che incorpora strutture rappresentanti sé stesso o aspetti di sé; tali strutture sono dette *la propria rappresentazione* del sistema.

1.3 Architetture Riflessive

def. Con *Architettura Riflessiva* si intende la struttura assunta da un sistema riflessivo ed in particolare l'insieme dei meccanismi necessari ad un sistema computazionale per effettuare delle computazioni riflessive.

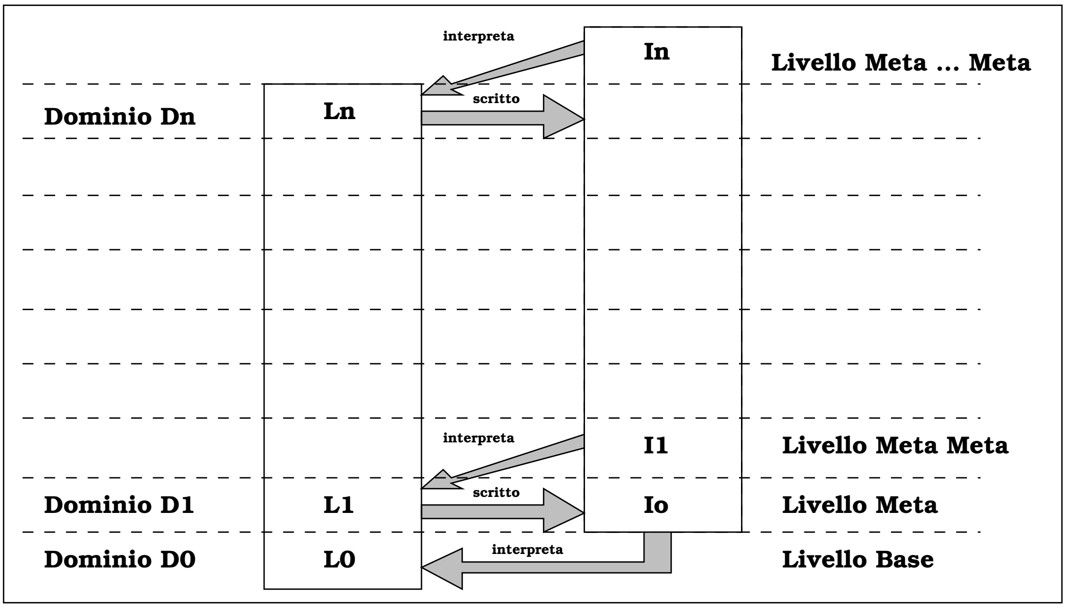


Figura 1.1: **Torre Riflessiva**

Le architetture riflessive forniscono un nuovo paradigma per costruire sistemi software. Tali sistemi riflessivi sono strutturati in una Torre Riflessiva; ogni livello L_i rappresenta un determinato dominio D_i. Il livello L_i è il livello base per il livello L_{i+1} ed un meta-livello per il livello L_{i-1}. Ogni entità operante in un certo livello ha la possibilità di manipolare le entità operanti nel suo livello base ed è a sua volta manipolata dalle entità operanti nel suo meta-livello. Ovviamente fa eccezione il livello L_0 che è solo il livello base per il livello L_1, e le entità operanti nel livello L_0 che possono solo essere manipolate e non possono manipolare nessun'altra entità.
Il modello tradizionale di riflessione è basato sull'interpretazione[1]. Il dominio D_n è costituito, come detto, da un insieme di entità ed espressioni scritte in un linguaggio L_n e di un interprete I_n, che interpreta queste espressioni. Tale interprete è scritto in un linguaggio L_{n+1} a sua volta interpretato da un interprete I_{n+1}, e così via all'infinito (vedi figura [1.1]); la regressione infinita viene evitata usando un interprete I' scritto in un linguaggio L' che sostituisce, l'interprete I_n dove n è il più elevato grado di computazione richiesto.
Un'ulteriore distinzione, che si può fare sulla computazione riflessiva, è tra riflessione strutturale e riflessione computazionale vera e propria. La prima è quell'attività riflessiva che è svolta da un'entità appartenente ad un meta-livello e che coinvolge la struttura di una entità dei livelli sottostanti, mentre la seconda identifica l'attività

[1]Nel caso tradizionale i vari livelli sono gestiti da un interprete però esistono altri casi in cui la riflessione è ottenuta sfruttando dei preprocessori o dei compilatori.

riflessiva che è svolta da un'entità del meta-livello e che, però, non coinvolge la struttura di alcun elemento dei livelli sottostanti.
I sistemi riflessivi hanno un interesse pratico nell'implementazione di sistemi entro il sistema stesso. Alcune delle caratteristiche principali di tali sistemi sono:

- **monitoraggio di entità del linguaggio**: possibilità di ispezionare, analizzare e modificare dinamicamente il codice entro il linguaggio stesso.

- **interfacciamento e debugging**: implementare tool all'interno del linguaggio (es. watching, tracing).

- **tolleranza ai guasti**: costruzione di sistemi in grado di rilevare|prevenire situazioni d'errore non previste dal progettista.

- **gestione entità interne**: si può direttamente accedere alle entità proprie del programma, nomi di variabili, procedure e oggetti, mantenendo statistiche o effettuando azioni particolari; questo tipo di attività riflessiva è detta riflessione strutturale.

- **aumento delle capacità di un linguaggio di programmazione**: introducendo i meccanismi riflessivi in un linguaggio di programmazione è possibile aggiungere diverse proprietà: in un linguaggio basato sugli oggetti è possibile aggiungere l'ereditarietà o in uno con solo l'ereditarietà singola, quella multipla; o l'ereditarietà tra metodi come in BETA o i before-after methods come in FLAVORS.

def. Dotare un linguaggio di programmazione di una *architettura riflessiva* significa dotarlo dei meccanismi necessari per riconoscere la riflessione come un concetto fondamentale della programmazione e degli strumenti necessari per trattare le computazioni riflessive esplicitamente, cioè:

(i) Il compilatore o l'interprete di tale linguaggio fornisce al sistema la possibiltà di effettuare elaborazioni riflessive includendo semplicemente delle direttive e delle istruzioni che indicano come fare.

(ii) Il compilatore o l'interprete garantisce il mantenimento delle proprietà di connessione causale esistente tra la rappresentazione propria del sistema e il sistema stesso. Quindi le modifiche apportate dal sistema alla sua propria rappresentazione si riflettono sul suo stato ed elaborazione.

Quando poi si implementa effettivamente la struttura riflessiva in un linguaggio di programmazione ci si scontra con i limiti imposti dal linguaggio; di seguito si espongono dei limiti indipendenti dal linguaggio adottato:

- La finitezza degli strumenti che si usano per implementare la torre riflessiva, limita il numero dei livelli che vengono effettivamente implementati della torre, in particolare si definisce la struttura come costituita da solo un livello (il livello base) e si forniscono i meccanismi per agganciare questo livello con il suo meta-livello e così via per il meta-meta-livello, ecc...

- A livello teorico risulta ovvia la possibilità di implementare ogni livello in un linguaggio differente. Quando si implementa la torre riflessiva, sebbene questa scelta sia possibile, risulta un'inutile complicazione e quindi, spesso e volentieri si riducono i linguaggi ad uno solo.

La torre di livelli è una suddivisione logica. Suddivisione che in una eventuale implementazione, non è detto venga rispettata o perlomeno non è detto che sia così evidente.

1.4 Caratteristiche Riflessive

Come si può facilmente osservare, riducendo al minimo i concetti finora espressi, il comportamento riflessivo si manifesta deviando il normale flusso di esecuzione dal livello base ad un meta-livello per effettuare delle computazioni ed infine tornare nel normale flusso di esecuzione; a questo punto risulta ovvio chiedersi se non esistono già delle caratteristiche riflessive nei linguaggi o nelle metodologie di programmazione finora adottate. La risposta è affermativa e di seguito vengono elencati alcuni dei casi più interessanti:

- Le interruzioni permettono di deviare dal flusso di esecuzione corrente ad un'altro in modo da gestire la richiesta di interruzione.

- Il trattamento delle eccezioni permette, quando si verifica un errore di interrompere la normale esecuzione, effettuare delle operazioni di recupero ed infine ritornare a rieseguire la normale computazione.

- Le asserzioni di EIFFEL permettono di verificare prima e dopo l'esecuzione di un metodo (computazione riflessiva) determinate condizioni.

- I metodi before-after di FLAVORS che vengono eseguiti, automaticamente, prima e dopo l'esecuzione di un metodo, effettuando così delle computazioni riflessive.

- Gli interpreti, i debugger, gli emulatori, i simulatori e tutte quelle applicazioni che costruiscono una macchina virtuale su una macchina reale al fine dello svolgimento delle funzionalità di un'altra applicazione hanno caratteristiche riflessive.

- La stratificazione dei sistemi operativi in cui sul firmware della macchina vengono costruiti diversi strati software, ed ognuno dei quali sovraintende determinate funzioni, e sull'ultimo livello vengono costruite le applicazioni utenti, le quali usufruiscono dei servizi (computazioni) offerte dallo strato sottostante, che usufruisce a sua volta dei servizi forniti dallo strato sottostante e così via fino a raggiungere il firmware; come nella torre di interpreti presentata in precedenza. Il tutto ha forti connotati riflssivi.

1.5 Linguaggi Object Oriented e Riflessioni

I linguaggi orientati agli oggetti sembrano essere i migliori candidati per applicare i concetti della riflessione, questo a causa della loro naturale distribuzione dei dati e dei metodi all'interno degli oggetti.
In questo capitolo prenderemo in considerazione il linguaggio SmallTalk che è un linguaggio ad oggetti puro[2], per le caratteristiche descritte di seguito.

1.5.1 SmallTalk e riflessione

Il linguaggio SmallTalk per le sue caratteristiche è particolarmente adatto all'applicazione dei concetti riflessivi.
Peculiarità di SmallTalk è il fatto che ogni entità del linguaggio è un oggetto, inclusa quindi la descrizione degli oggetti stessi (classi). Quindi le classi sono a loro volta oggetti, istanze di altre classi, dette meta-classi. Ogni classe è l'unica istanza della propria meta-classe. Tali meta-classi non hanno un vero e proprio nome ma sono indicate dal nome della classe che è loro unica istanza, seguito dal suffisso class, quindi Point class non è altro che la meta-classe della classe Point. Si chiarisce questo concetto con un esempio basato sulle classi Point e Circle il cui codice è descritto di seguito.

SmallTalk n° 1.2 ***Classi Point e Circle***

```
! Class Point !

Object subclass : #Point
 instanceVariableNames: ´x y´
 classVariableNames: ´´
 poolDictionaies: ´´

!Point class methods!

x: aFloatX y: aFloatY
  "costruttore del punto di coordinate ( aFloatX, aFloatY )"
 ^(self new) setX: aFloatX setY: aFloatY.
```

[2]Un linguaggio di programmazione ad oggetti si dice puro quando ogni entità del linguaggio è vista come un oggetto

```
!Point methods!

x
  "selettore della coordinata x"
 ^x.
y
  "selettore della coordinate y"
 ^y.

setX: aFloatX setY: aFloatY
  "inizializza le coordinate di un punto"
 x := aFloatX.
 y := aFloatY.

! Class Circle !

Point subclass: #Circle
 instanceVariableNames: ´ray´
 classVariableNames: ´´
 poolDictionaies: ´´

!Circle class methods!

x: aFloatX y: aFloatY ray: aFloatRay
  "costruttore del cerchio di centro il punto ( aFloatX, aFloatY ) e
   raggio aFloatRay"
 ^(( self class ) x: aFloatX y: aFloatY) setRay: aFloatRay.

!Circle methods!

ray
  "selettore per il raggio"
 ^ray.

setRay: aFloatRay
  "inizializza il raggio del cerchio"
 ray := aFloatRay.
```

Si considerino, quindi, le classi Point e Circle che sono legate tra loro da una relazione di sottoclasse (Circle è una sottoclasse di Point), inoltre la classe Point è sottoclasse di Object che è la classe radice del sistema SmallTalk e come tale non è sottoclasse di nessun'altra classe.
Quando un messaggio è inviato ad un'istanza di una classe (ad esempio di Circle), l'interprete SmallTalk ne cerca il selettore nella classe corrispondente, se non lo trova risale tramite il link di ereditarietà alla super classe, nel nostro caso la classe Point, se anche qui non è presente esso continua a risalire fino ad arrivare nella classe Object e nell'ipotesi che non sia neanche qui, segnala un errore (invio del messaggio doesNotUnderstand all'oggetto di partenza).
Essendo le classi degli oggetti, vediamo di quale classe sono istanza Point, Circle e Object e dove vengono cercati i selettori di messaggi inviati ad una classe (come ad esempio il selettore x:y:ray: inviato alla classe Circle). Come detto ogni classe è istanza della propria meta-classe quindi Circle è istanza di Circle class, Point di Point

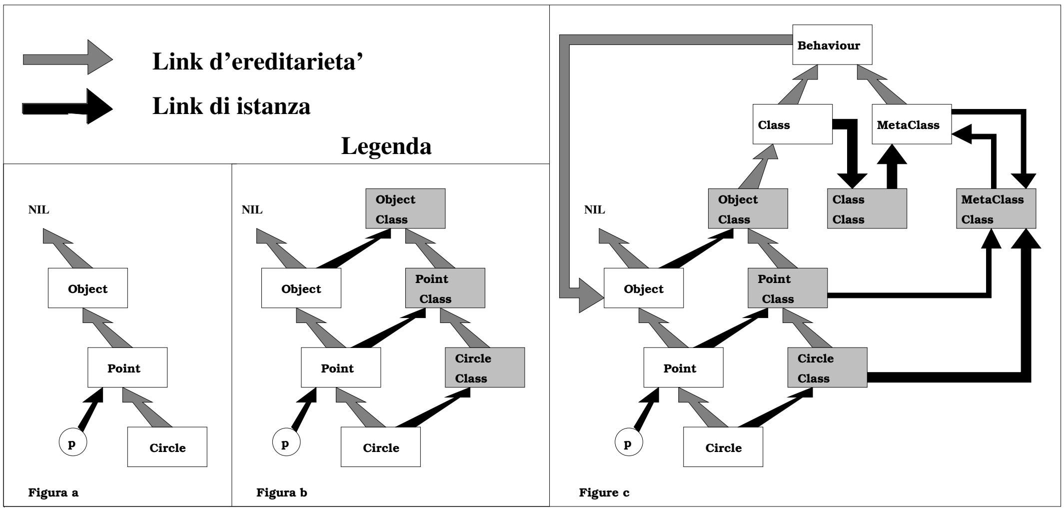


Figura 1.2: **Gerarchia delle classi in SmallTalk:** a) Classi e istanze di Object, Point e Circle b) Classi istanze e Meta Classi c) sistema completo.

class e Object di Object class; inoltre la gerarchia esistente fra le classi continua ad esistere tra le meta-classi, quindi Circle class è sottoclasse di Point class, che a sua volta è sottoclasse di Object class.
Per i messaggi mandati alle classi, i corrispondenti selettori sono cercati nelle meta-classi, seguendo la gerarchia come per il caso di messaggi inviati agli oggetti. I metodi e le variabili delle meta-classi sono descritti come *ClassVariables* e *ClassMethods* nella classe corrispondente.
Per completare la spiegazione occorre dire di cosa sono istanze le meta classi; le meta-classi sono tutte istanze della classe MetaClass, inoltre la Object class eredita dalla classe Class, sia Class che MetaClass sono sottoclassi della classe Behaviour che è a sua volta sottoclasse di Object. Ovviamente tali classi sono istanze delle rispettive meta-classi che sono istanze della classe MetaClass.
In figura [1.2] è rappresentata la gerarchia indicando le classi con un rettangolo, le meta-classi con un rettangolo grigio e gli oggetti con un cerchio; la freccia piena indica la relazione di istanza, quella tratteggiata quella di ereditarietà; inoltre per ragioni di chiarezza grafica, si sono tralasciati il link di ereditarietà tra Object class e MetaClass class e la classe di Behaviour con i relativi link di ereditarietà.
Avendo descritto come gestisce l'ereditarietà SmallTalk (in particolare SmallTalk-V, per vedere il caso di SmallTalk-80 in dettaglio rifarsi a [Goldberg-Robson 83]) si è in grado di giustificare l'affermazione fatta ad inizio sezione:
è molto semplice in questo linguaggio avere un'entità che ne monitorizza un'altra, è sufficiente usare i metodi e le variabili di classe per controllare il comportamento di ogni istanza di tale classe.

Ad esempio supposto di voler sapere quante istanze esistono della classe Point descritta prima, è sufficiente dotarla di una variabile di classe Instance e modificare il metodo di classe x:y:, che è il costruttore delle istanze, facendo si che questo incrementi tale variabile (vedi codice in figura) ed aggiungendo un metodo d'istanza Statistics per visualizzare tale informazione

SmallTalk n° 1.3 ***Riflessione in SmallTalk.***

```
x: aFloatX y: aFloatY
  "costruttore di un punto"
 ((self class) Instance isNil)
   ifTrue: [Instance := 1]
   ifFalse: [Instance := Instance + 1].
 ^(self new) x: aFloatX y: aFloatY.

statistics
  "statistiche sul numero di istanze"
 Transcript
  NextPutAll: ´Sono state create ´ .
 ((self class) Instance) storeOn: Transcript.
 Transcript
  NextPutAll: ´ istanze della classe ´ ;
  NextPutAll ( self class ) name ; cr.
```

Vorrei notare che grazie all'ereditarietà vengono contate come nuove istanze di Point anche eventuali nuove istanze di Circle.
Questo è un esempio di programmazione riflessiva, l'unico particolare è che essendo imposta dal programmatore si perde la proprietà di trasparenza[3].

1.6 Approcci alla costruzione di Architetture Riflessive

Durante la definizione delle strutture linguistiche necessarie, ad un linguaggio object-oriented, per potervi sviluppare delle applicazioni riflessive, il progettista deve effettuare delle scelte relativamente ai successivi tre quesiti:

- Quali sono le entità operanti nei meta-livelli e qual è la loro struttura ed il loro comportamento.
- Come si trattano i messaggi e la ricerca dei metodi definiti nel meta-livello relativamente al passaggio dei messaggi.

[3]Trasparenza: proprietà grazie alla quale, una caratteristica, nel nostro caso la riflessione, diventa parte integrante dell'architettura, e quindi il suo uso diventa invisibile (trasparente) all'applicazione perché incapsulata altrove.

- Quando il sistema effettua il passaggio della computazione da un'entità del livello inferiore a quella del livello superiore e viceversa.

Rispondendo a tali quesiti si dà origine ai diversi modelli.
I principali modelli di riflessione computazionale applicati ai linguaggi orientati agli oggetti basati sulle classi possono essere classificati in base a due approcci:

- approccio a *meta-oggetto*: suppone che le entità operanti nei meta-livelli siano delle particolari entità dette oggetti di livello meta o meta-oggetti, che supervisionano ed eventualmente alterano il comportamento delle entità appartenenti ai livelli sottostanti. Questo approccio è principalmente rappresentato da due modelli: il modello a meta classe[4] ed il modello a meta-oggetto.

- approccio a *materializzazione del messaggio*: suppone che le entità operanti nei meta-livelli siano la "materializzazione" dell'attivazione di un metodo, da parte di un oggetto del livello inferiore, in un oggetto. Questo approccio è rappresentato dal modello omonimo, il modello a materializzazione del messaggio.

Una trattazione di questi approcci è stata fatta da Ferber in [Ferber 89]; la cui impostazione, è seguita in queste pagine con la sola differenza dell'uso del linguaggio SmallTalk invece di ObjVLisp [Cointe 87] per esaminare le possibili implementazioni.

Scegliendo SmallTalk per la spiegazione dei modelli implicitamente ci si appoggia alla sua gerarchia di classi per la ricerca dei metodi da attivare (vedere [1.5.1]).
Il primo e forse maggior problema dovuto all'uso di SmallTalk come ambiente di sviluppo nasce dall'approccio funzionale proprio dello scambio dei messaggi di questo linguaggio.
L'approccio funzionale, non consente di modificare le regole di attivazione dei metodi, modifica necessaria nella costruzione di applicazioni riflessive, in quanto sono proprio queste regole a determinare il passaggio della computazione da un'entità di un livello ad un'entità di un altro livello.
Ci sono due possibilità, la prima è quella di addentrarsi nell'interprete stesso, modificandone il codice al fine di implementare la semantica desiderata (più elegante ma impossibilitata dall'indisponibilità del codice sorgente e comunque non adatta ai nostri scopi, rivolti a linguaggi tipati staticamente), la seconda, adottata per la sua semplicità, consiste nell'imporre una filosofia di programmazione che si basa sull'interprete standard, cioè, nello scrivere un metodo **send**, appartenente alla classe **Object** e quindi comune a tutte le classi di sistema, e far sì che sia questo

[4] Il termine meta-classe, non è legato alle meta-classi di SmallTalk, ma indica che le entità appartenenti ai meta-livelli (i meta-oggetti) sono le classi, che in questo caso chiameremo meta-classi, degli oggetti appartenenti ai livelli inferiori; come tale bisogna prestare attenzione a non confondere le due terminologie.

metodo ad implementare lo scambio di messaggi; in questo modo si ottiene uno stile di programmazione che usa tale metodo per passare i messaggi invece del tradizionale approccio funzionale tipico di SmallTalk (un'altra soluzione che non modifica l'interprete è stata proposta da Foote e Johnson in [Foote-Johnson 89]. Tale proposta si basa sugli incapsulatori di Pascoe [Pascoe 86]).

Vediamo di capire cosa fa la send: e che relazione ha con il solito metodo di passare i messaggi.
Si supponga di avere un oggetto **obj** in grado di rispondere ad un selettore di messaggio **msg** (supposto che sia un messaggio unario, cioè l'unico parametro è l'oggetto ricevente), in SmallTalk l'attivazione|l'invio del messaggio è:

```
obj msg
```

se, invece, **msg** ha più di un parametro (keyword message), ad esempio due, in SmallTalk si scrive:

```
obj msg: par
```

adottando la filosofia basata su **send** occorre prima costruire il messaggio[5] (istanza della classe Message a cui vengono aggiunti i metodi setSelector: e setArguments: come illustrato nella sezione seguente) e poi eseguire il metodo send: come segue:

```
|aMessage|
(aMessage new) setSelector: 'msg'.
obj send: aMessage.
```

se il metodo richiede degli argomenti allora essi vanno inseriti, nell'ordine corretto, nel messaggio ottenendo il seguente stub di codice

```
|aMessage|
(aMessage new) setSelector:'msg' ;
                 setArguments: Array with: a 1 ... with: a n.
obj send: aMessage.
```

[5] In realtà ho semplificato la programmazione demandando la costruzione del messaggio allo stesso metodo **send**, prevedendo quattro metodi sostitutivi, **send: send:with: send:with:with:** e **send:withArgs:** , che inviano rispettivamente messaggi unari, binari, ternari ed n-ari, comunque durante la trattazione ci si continua a riferire a **send:** indipendentemente dal numero dei parametri. Ad esempio l'invio di un messaggio ternario:

```
|aMessage|
(aMessage new) setSelector: 'msg' ; setArguments: Array with: a1 with: a2.
obj send: aMessage.
```

si trasforma in

```
|aMessage|
obj send: 'msg' with: a1 with: a2.
```

invece di obj msg:a_1 k:a_2 ... k:a_n.
Il metodo è più flessibile in quanto nel momento in cui si vuole cambiare modo di inviare i messaggi è sufficiente mascherare il metodo send: ridefinendolo in una classe discendente di Object con il nuovo comportamento. Inoltre si ottiene una maggiore trasparenza nell'uso delle riflessioni e si offre all'utente la possibilità di dotare o meno la computazione di comportamento riflessivo.
Tenendo presenti le considerazioni e le modifiche|aggiunte fatte sinora a SmallTalk, vediamo come si possono implementare i diversi modelli, esaminandone più a fondo le caratteristiche e le potenzialità.

1.6.1 Approccio a Meta-Oggetto

In questo approccio, ad ogni oggetto è potenzialmente associato un meta-oggetto che descrive il comportamento base dell'oggetto stesso. Noi diremo che il meta-oggetto rappresenta l'oggetto e che quest'ultimo è il *referente* del meta-oggetto.
Essendo i meta-oggetti stessi degli oggetti, anch'essi possono avere meta-oggetti e così via, ne consegue una "torre infinita" di meta-oggetti, come già detto nella sezione [1.3].
Per ottenere un tale comportamento occorre modificare la semantica dell'azione invio dei messaggi, facendo sì che la gestione dei messaggi venga trattata dal meta-oggetto invece che dall'oggetto stesso o da meccanismi "standard" implementati dal compilatore o dall'interprete.
Supposto che lo scambio di messaggi sia effettuato dalla primitiva send: allora, l'invio di un messaggio *msg* con argomenti $a_1 \ldots a_n$ ad un oggetto *obj* dotato di meta-oggetto *m-obj* equivale, informalmente, ad inviare a *m-obj* un messaggio speciale HandleMessage, avente come parametri il messaggio stesso.
I requisiti da valutare, proposti ad inizio sezione, sono: *chi è il meta-oggetto* e *quando l'elaborazione passa da oggetto a meta-oggetto e viceversa*. Ci sono due possibili soluzioni ognuna con i propri vantaggi e svantaggi: considerare la classe dell'oggetto come suo meta-oggetto e passare al livello meta quando uno dei due lo richiede, oppure considerare come meta-oggetto un'istanza di una classe apposita Meta-Object e passare il controllo al meta livello se e solo se l'oggetto è dotato di un meta-oggetto. Quindi esistono due possibili modelli:

- **Modello a Meta-Classe** che si basa sull'idea di far coincidere il meta-oggetto con la classe[6] stessa dell'oggetto.

- **Modello a Meta-Oggetto** che si basa sull'idea che i meta-oggetti siano istanze di una classe apposita detta Meta-Object o di sue sottoclassi.

[6] Nel seguito, spesso e volentieri, si userà il termine meta-classe in luogo di classe quando questa è il meta-oggetto dell'oggetto riflessivo.

Nelle sezioni successive si analizzano i due modelli in maggior dettaglio presentandone possibili implementazioni in SmallTalk.

1.6.1.1 Modello a Meta-Classe

Nei linguaggi object-oriented puri, cioè quei linguaggi object-oriennted in cui ogni entità è vista come un oggetto, come SmallTalk è normale considerare la classe di un oggetto come se fosse il suo meta-oggetto, questo a causa della naturale riflessività della struttura del modello (per maggiori dettagli si rimanda alla sezione [1.5.1]). Inoltre le classi contengono informazioni sulla struttura degli oggetti (nomi della super classe, delle variabili, dei metodi, etc...)[7].
Quindi risulta naturale far coincidere il meta-oggetto con la classe dell'oggetto interessato, per cui la semantica del passaggio dei messaggi diventa:

$$\boxed{\mathrm{obj}\ \mathbf{send}\ \mathrm{msg}\ \mathrm{a}_1 \cdots \mathrm{a}_\mathrm{n} \equiv \mathrm{obj\ class}\ \mathbf{HandleMessage}\ \mathrm{msg}\ \mathrm{a}_1 \cdots \mathrm{a}_\mathrm{n}}$$

Questo significa che, da un punto di vista comportamentale, per un oggetto *obj* ricevere un messaggio *msg*, equivale da parte della sua classe ricevere il messaggio speciale HandleMsg *msg*.
Le caratteristiche di questo modello sono:

- Tutte le istanze di una classe condividono lo stesso meta-oggetto e quindi tutte le istanze di tale classe per difetto hanno lo stesso comportamento riflessivo. Usando l'ereditarietà o dei dizionari è possibile differenziare il comportamento riflessivo degli oggetti istanza della stessa classe:
    - Grazie all'ereditarietà è possibile creare delle classi fittizie, che non aggiungono o tolgono niente alla classe padre. Queste sono delle vere e proprie classi, quindi le istanze di queste classi hanno meta-oggetto diverso da quello delle istanze della classe padre (classe diversa, meta-oggetto diverso) anche se a tutti gli effetti sono istanze della stessa classe. Questa soluzione, comunque, presenta degli inconvenienti, oltre all'innaturalezza del metodo si può incorrere, a seconda del linguaggio usato, in oggetti con struttura a tutti gli effetti identica, ma incompatibili tra loro.
    - Dotando la classe di un dizionario indiciato sulle istanze della classe e avente come informazioni dei metodi utilizzati per mascherare il me-

[7] Tali informazioni sono usate per ottenere *riflessioni di tipo strutturale*. Si parla di riflessione strutturale o inerente la struttura, se si effettuano computazioni riflessive, che coinvolgono la struttura rappresentante il dominio in cui opera il sistema computazionale, e viene distinta dalla riflessione computazionale, in cui tali strutture non vengono coinvolte. Tale distinzione si è rivelata necessaria nel momento in cui è stato introdotto un modello computazionale in cui l'entità che riflette è distinta dall'entità che mantiene le strutture rappresentanti il dominio computazionale.

todo HandleMessage predefinito, è possibile differenziare il comportamento riflessivo delle istanze di una certa classe; gli inconvenienti sono rappresentati dalla complessità implementativa della metodologia.

comunque questi sono solo dei palliativi che tendono a snaturare il modello per renderlo più flessibile.

- Le informazioni mantenute dal meta-oggetto sono condivise da tutti gli oggetti istanze di quella classe, quindi mantenere informazioni, proprie su un oggetto o nascoste ad un oggetto è possibile solo usando uno dei metodi descritti nel caso precedente, per i quali continuano a valere le considerazioni fatte.
- La riflessione computazionale e quella strutturale sono affidate alla stessa entità, la classe del ricevente. In un linguaggio object-oriented (ad esempio SmallTalk), la classe possiede tutte le informazioni strutturali; quindi essendo, in questo modello, la classe a svolgere il ruolo di meta-oggetto, la riflessione strutturale e quella computazionale sono svolte dalla stessa entità.
- Un meta-oggetto, può monitorizzare solo oggetti istanza della stessa classe o di classi aventi un antenato comune, questo limita la possibilità di interazione di entità al livello base tramite il meta-oggetto.

Per implementare questo tipo di modello all'interno del sistema SmallTalk si è dovuto modificare la classe Object stessa inserendovi una variabile di istanza reflective. Essa assume valori booleani, usati per stabilire se la computazione deve essere elaborata dall'oggetto in questione (elaborazione normale) o dal suo meta-oggetto (elaborazione riflessiva). Si ha, quindi, bisogno per gestirla dei metodi setReflective: e Reflective, che permettono, rispettivamente, di modificarla e di accedere al suo valore. Si è inserito tale variabile addirittura nella radice della gerarchia di SmallTalk per poter far si che qualsiasi oggetto, istanza sia di classi già esistenti che di classi nuove, possa avere un meta-oggetto che lo monitorizzi. Si osserva che la torre riflessiva si implementa banalmente: siccome in SmallTalk anche le classi sono oggetti, che ereditano da Object (vedi figura [1.2]), di conseguenza anche le meta-classi, che non sono altro che i meta-oggetti, hanno lo slot reflective e quindi la possibilità di essere monitorati. Se, invece, reflective fosse stato inserito solo in alcune classi, si sarebbe limitato le possibilità di riflessione solo a quelle classi ed ai loro discendenti. Inoltre reflective è stato inserito come variabile d'istanza e non di classe per avere un'organizzazione più flessibile, in questo modo, infatti, si possono avere oggetti, istanze della stessa classe, che possono essere controllati separatamente.

Occorre aggiungere anche dei metodi usati per la gestione della riflessioni, tali metodi sono quelli che permettono il passaggio dal livello base al livello meta, e ritorno quando richiesto. Essi sono il metodo send:, il metodo apply: usato per eseguire effettivamente il messaggio ed infine il gestore del messaggio riflessivo, il selettore

HandleMessage:To:. I primi due metodi sono metodi di istanza di Object, in modo che qualunque oggetto vi possa rispondere, per cambiarne il comportamento è sufficiente mascherarli nelle sottoclassi. Il terzo è un messaggio a cui devono saper rispondere tutte le meta-classi e come tale è stato inserito tra i metodi di classe di Object (per maggiori informazioni vedere il paragrafo [1.5.1]); di seguito è riportato il codice di tali aggiornamenti.
Si noti che, per agevolare la lettura degli esempi, il metodo **send**: prevede solo messaggi unari, gli altri casi differiscono solo per la presenza dei parametri e per come è costruito il messaggio.

SmallTalk n° 1.4 *Modifiche effettuate alla classe* **Object** *per ottenere un'architettura riflessiva secondo il modello a Meta Classe.*

```
 nil subclass: #Object
  instanceVariableNames: 'reflective'
  classVariableNames: 'RecursionInError Dependents RecursiveSet '
  poolDictionaries: ''

!Object class methods !

HandleMessage: aMessage To: anObject
  "gestore metodi riflessivi"
 Transcript
  nextPutAll: '+++ sono in HandleMessage ' ; cr ;
  nextPutAll: '+++ e` stato chiamato ' ;
  nextPutAll: ( aMessage selector ) .
 ((aMessage arguments) isNil)
  ifFalse:
   [Transcript nextPutAll: ' con ' .
    ( aMessage arguments ) size storeOn: Transcript.
    Transcript nextPutAll: ' argomenti'.
   ].
 Transcript cr.
 ^(anObject apply: aMessage).

 !Object methods !

 apply: aMessage
   "applica il messaggio aMessage a self"
  ^(self perform: ( aMessage selector ) withArguments: ( aMessage arguments )).

 Reflective
   "true se possiede un meta-oggetto e' false altrimenti"
  ^reflective.

 send: aSelector
   "metodo per il trattamento delle reflection (modello MetaClasse)"
  |aMessage|
              "materializzazione del messaggio"
  aMessage := ( Message new ) setSelector: aSelector.
              "gestione della riflessione"
  ((self Reflective isNil) or: [(self Reflective) not])
   ifFalse: [^(( self class ) send: 'HandleMessage:To:' with: aMessage with: self)]
   ifTrue: [^(self apply: aMessage)].
```

```
setReflective: aBoolean
  "permette il passaggio da livello base a livello meta"
 reflective := aBoolean.
```

Il meccanismo di funzionamento è il seguente: il metodo send: controlla se si deve gestire in modo riflessivo il messaggio, in caso affermativo manda il messaggio HandleMessage:To: al meta-oggetto, passando così al livello meta, il quale per default si limita a notificare qual è il metodo che si vuole eseguire e con quanti argomenti è stato chiamato (per modificarne il comportamento è sufficiente ridefinirlo in una sottoclasse) e poi esegue effettivamente il messaggio ritornando al livello base; altrimenti esegue solo il messaggio al livello base, senza passare al livello meta. Si è inserita una chiamata ad apply: anche nel metodo HandleMessage:To: in modo da localizzare eventuali modifiche comportamentali.

Il messaggio è passato come una istanza della classe di sistema Message e viene costruito dal metodo send:. In Message sono stati inseriti due nuovi metodi: setSelector: e setArguments: necessari rispettivamente per inizializzare il selettore e gli argomenti del messaggio; tali metodi non alterano il comportamento della classe Message grazie alla proprietà di incapsulamento propria dei linguaggi ad oggetti. Il codice dei due nuovi metodi è mostrato di seguito.

SmallTalk n° 1.5 *classe* Message.

```
Object subclass: #Message
  instanceVariableNames: ´selector arguments ´
  classVariableNames: ´´
  poolDictionaries: ´´

!Message methods !

setArguments: anArray
  "in anArray ci sono gli argomenti del messaggio"
 arguments := anArray.

setSelector: aString
  "aString e` il selettore del messaggio"
 selector := ( aString asSymbol ).
```

1.6.1.2 Modello a Meta-Oggetto

In questo modello il meta-oggetto, invece di coincidere con la classe dell'oggetto che monitorizza, coincide con un'istanza di una classe, appositamente pensata detta Meta-Object.
Partendo dal presupposto che ogni oggetto ha un meta-oggetto che lo monitorizza,

anche quest'ultimi, essendo oggetti, possono avere un meta-oggetto che li monitorizza.
In questo modello, inviare un messaggio ad un oggetto riflessivo, equivale ad inviare il messaggio speciale **HandleMessage** al suo meta-oggetto, più formalmente la semantica del passaggio dei messaggi diventa:

$$\text{obj } \textsf{send} \text{ msg } a_1 \ldots a_n \equiv \begin{array}{l} \text{if (exist meta-obj) then} \\ \text{meta-obj } \textsf{HandleMessage} \text{ msg } a_1 \ldots a_n \\ \text{else obj msg } a_1 \ldots a_n \end{array}$$

Questo, da un punto di vista comportamentale, significa che per un oggetto **obj** ricevere un messaggio **msg**, o per il meta-oggetto **meta-obj** ad esso associato ricevere il messaggio speciale **HandleMessage: msg** è equivalente.

Nota Tra i due modelli, derivati dall'approccio a meta-oggetto, cambia solamente chi o cosa rappresenta il meta-oggetto; nel primo caso è la classe dell'oggetto ricevente, nel secondo è un oggetto, istanza di una apposita classe (quest'ultima rappresentazione porta a distinguere tra chi effettua la riflessione computazionale e chi quella strutturale).

Le caratteristiche di questo modello sono:

- Il meta-oggetto di un oggetto è distinto dalla classe dello stesso, però è sempre la classe che nei linguaggi object-oriented puri, come SmallTalk mantiene le informazioni strutturali sull'oggetto; quindi l'entità che effettua la riflessione computazionale è diversa da quella che effettua la riflessione strutturale (meta-oggetto e classe).

- Il legame tra oggetto e meta-oggetto non è più ottenuto staticamente, ma è di carattere dinamico. Quindi è possibile in esecuzione cambiare il meta-oggetto di un certo oggetto. Il metodo con cui si cambia il meta-oggetto è dipendente da come è definito il legame tra oggetto e meta-oggetto. Il modo più ovvio per legarli è quello di inserire una variabile nell'oggetto riflessivo che tiene il riferimento con il meta-oggetto, quindi la modifica è ottenuta assegnando a tale variabile un nuovo valore.

- Essendo il riferimento al meta-oggetto proprio dell'oggetto riflessivo e non della sua classe, allora istanze della stessa classe possono avere meta-oggetti distinti. Quindi è possibile, in modo semplice, far mantenere al meta-oggetto informazioni su singoli oggetti, come su intere classi o gruppi di oggetti istanze di classi diverse. Quindi ogni meta-oggetto può monitorare più oggetti, istanze di classi completamente distinte.

- È facile definire comportamenti riflessivi nuovi, è sufficiente costruire delle nuove classi discendenti da MetaObject in cui si maschera il metodo HandleMessage: con quello che effettua il comportamento riflessivo desiderato. Quindi

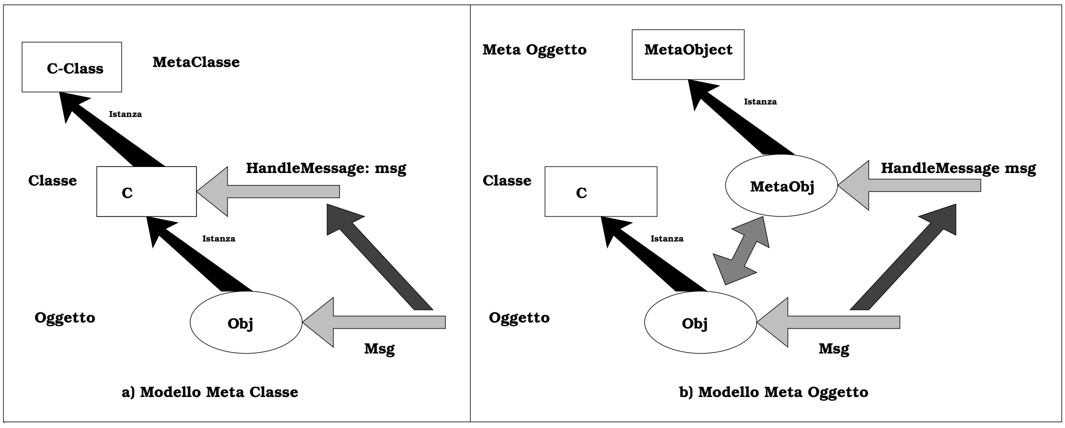


Figura 1.3: **Approccio a Meta Oggetto**

ogni meta-oggetto istanza di questa nuova classe manifesta un comportamento riflessivo differente da quello definito in MetaObject.

Per implementare questo tipo di modello all'interno del sistema SmallTalk, si è dovuto modificare la classe Object, aggiungendo una variabile di istanza meta. Lo slot meta rappresenta il collegamento che ogni oggetto ha con il proprio meta-oggetto. Inoltre sono stati aggiunti i metodi necessari per gestirla meta e meta: che sono usati rispettivamente per sapere qual è il meta-oggetto e per creare il legame con tale meta-oggetto; si ritorna sul codice di questi metodi nel momento in cui si descrive la classe MetaObject. Oltre a questi si devono anche aggiungere i metodi necessari per la gestione del passaggio di messaggi, che sono: apply: che resta identica, sia come concetto che come codice, a quella del modello a meta-classe, il gruppo delle send:, che gestiscono il passaggio della computazione tra il livello meta e quello base (queste differiscono da quelle descritte nel modello a meta-classe solo per l'entità cui si demanda la computazione riflessiva). Il codice di queste estensioni è riportato di seguito; come al solito il metodo send: è presentato solo per il caso di messaggi unari.

SmallTalk n° 1.6 *Modifiche effettuate alla classe* **Object** *per ottenere un'architettura riflessiva secondo il modello a Meta Oggetto.*

```
 nil subclass: #Object
  instanceVariableNames: 'meta'
  classVariableNames: 'RecursionInError Dependents RecursiveSet'
  poolDictionaries: ''

!Object methods!

apply: aMessage
  " applica il messaggio all'oggetto a livello base"
 ^(self perform: (aMessage selector) withArguments: (aMessage arguments)).
```

```
send: aSelector
  "metodo per il trattamento delle riflessioni (modello meta oggetto, messaggi unari)"
 |aMessage|
 aMessage := ( Message new ) setSelector: aSelector.
 ((self meta) isNil)
   ifTrue: [^(self apply: aMessage)]
   ifFalse:
    [(self meta) setObj: self.
     ^((self meta) send: 'HandleMessage:' with: aMessage)
    ].
```

Per come è pensato il modello è necessaria anche una nuova classe (MetaObject) e in base a come viene definita cambia la semantica e quindi il codice del metodo meta: (sempre della classe Object).
La prima cosa da vedere è, per default, cosa deve essere in grado di fare un'istanza della classe MetaObject; sicuramente deve poter rispondere al messaggio HandleMessage:. Rispetto al caso precedente il metodo HandleMessage: diventa un metodo d'istanza e non di classe e sono in grado di rispondergli solo le istanze di MetaObject o di classi sue discendenti, invece che tutti gli oggetti.
Nel modello a meta-classe gli oggetti monitorati dal meta-oggetto sono tutti noti, in quanto sono tutti istanza di quella classe che fa da meta-oggetto, pertanto per effettuare delle computazioni riflessive che coinvolgono tutto il gruppo dei referenti non è necessaria nessuna informazione aggiuntiva oltre all'oggetto che richiede effettivamente la computazione riflessiva.
Nel modello a meta-oggetto gli oggetti monitorati dal meta-oggetto possono essere diversi e non sono noti a priori, quindi per effettuare delle computazioni riflessive che coinvolgono tutto il gruppo dei referenti è necessario che il meta-oggetto li conosca e sappia anche qual è l'oggetto richiedente.
La struttura della classe deve contenere una lista di riferimenti ai possibili referenti (referents). Per poter tornare al livello base dopo la computazione riflessiva, deve sapere qual è il referente attuale, quindi deve tenere tale oggetto in uno slot (obj).
Si è rimandato la descrizione del codice di meta: proprio in vista di questa precisazione; meta: deve, infatti, mantenere consistente, oltre al collegamento tra oggetto e meta-oggetto anche la lista dei referenti.
Di seguito si riporta il codice della classe MetaObject e il codice dei metodi meta e meta: della classe Object.

SmallTalk n° 1.7 *Classe* **MetaObject**

```
Object subclass: #MetaObject
 instanceVariableNames: 'referents obj'
 classVariableNames: ''
 pooldictionaries: ''

!MetaObject class methods!
```

```
new
  "costruttore di meta-oggetti"
 ^((super new) setReferents).

!MetaObject methods!

HandleMessage: aMessage
  "gestore metodi riflessivi (default) si limita ad effettuare il ritorno al livello base"
 ^(obj apply: aMessage).

setObj: anObject
  "Inizializzatore per l´oggetto controllato"
 obj := anObject.

referents
  "selettore per i referenti"
 ^referents.

setReferents
  "inizializza la lista dei referenti"
 referents := Set new.
```

SmallTalk n° 1.8 *Metodi* **meta** *e* **meta:** *per la gestione dei referenti.*

```
meta
  "selettore del meta-oggetto"
 ^meta.

meta: aMetaObject
  "cambia il meta-oggetto di un oggetto"
 (aMetaObject isNil)
  ifTrue: [(meta referents) remove: self]
  ifFalse:
   [((self meta) isNil)
     ifFalse:[(meta referents) remove: self].
     (aMetaObject referents) add: self.
    ].
 meta := aMetaObject.
```

Il meccanismo di funzionamento è il seguente: se esiste un meta-oggetto allora il metodo send: invia il messaggio speciale HandleMessage: a tale meta-oggetto. Quindi la computazione è effettuata da entità del meta-livello. La HandleMessage per default si limita a tornare al livello base ed eseguire la computazione richiesta; se non esiste il meta-oggetto, si limita a seguire il normale flusso di esecuzione senza passare al livello meta.
Per modificare il comportamento riflessivo di un oggetto in esecuzione, è necessario innanzitutto definire una nuova classe discendente di MetaObject che specifichi il nuovo comportamento riflessivo; ad esempio se vogliamo che l'oggetto riflessivo abbia un comportamento verboso, cioè che informi su quale metodo è stato attivato, si definisce la classe VerboseMetaObject, in cui il metodo HandleMessage: stampa il

nome e gli argomenti del metodo invocato al livello base ed infine si deve dotare l'oggetto di un meta-oggetto istanza di tale classe per fornirgli un comportamento riflessivo.

|p mo vmo|
p := Point new.
mo := MetaObj new.
⋮
p send: 'meta:' with: mo.
⋮
vmo := VerboseMetaObject new.
p send: 'meta:' with: vmo.

Il precedente stub di codice SmallTalk mostra come si cambia il meta-oggetto ad una istanza della classe Point, durante l'esecuzione; da notare che qualsiasi chiamata del metodo send:, da parte di p, precedente alla prima attivazione di meta: non fornisce alcuna computazione riflessiva (non è stato ancora associato alcun meta-oggetto all'oggetto in questione), qualsiasi chiamata da parte di p del metodo send:, successiva alla prima chiamata del metodo meta: e precedente alla seconda chiamata del metodo meta:, fornisce una computazione riflessiva del tipo definita nella classe MetaObject; qualsiasi chiamata del metodo send: da parte dell'oggetto p, successiva alla seconda chiamata del metodo meta:, fornisce una computazione riflessiva del tipo definita nella classe VerboseMetaObjet.

SmallTalk n° 1.9 *Classe* **VerboseMetaObject.**

```
MetaObject subclass: #VerboseMetaObject
 instanceVariableNames: ´´
 classVariableNames: ´´
 poolDictionaries: ´´

!VerboseMetaObject methods!

HandleMessage: aMessage
  "gestore metodi riflessivi ( override quello presente nella sua
   super classe ) fornisce informazioni sul metodo eseguito"
 Trascript
  nextPutAll: ´+++ HandleMessage il metodo chiamato e` : ´ ;
  nextPutAll: ( aMessage selector ) .
 (( aMessage arguments ) isNil)
  ifFalse:
   [Transcript nextPutAll: ´ con ´ .
    (aMessage arguments ) size storeOn: Transcript.
    Transcript nextPutAll: ´ argomenti´ .
   ].
 Transcript cr.
 ^(super send: ´HandleMessage:´ with: aMessage).
```

In figura [1.3] è schematizzato il comportamento dello scambio dei messaggi, per entrambi i modelli.

1.6.2 Approccio a Materializzazione del Messaggio

Questo approccio è basato sulla materializzazione della comunicazione stessa, ogni comunicazione può essere un oggetto in grado di rispondere al messaggio HandleMessage: che gestisce la comunicazione reale che rappresenta.

1.6.2.1 Modello a Materializzazione del Messaggio

In questo modello, ogni comunicazione porta alla concretizzazione del messaggio inviato, cioè l'azione di **send** msg da origine ad un nuovo oggetto, istanza della classe Message o di classi sue discendenti, che incarna il messaggio stesso (questo quindi contiene tutte le informazioni che gli necessitano per la sua esecuzione).
Informalmente la semantica dello scambio dei messaggi diventa:

$$\boxed{\text{obj } \textbf{send} \text{ msg } a_1 \ldots a_n \equiv (\text{Message(msg } a_1 \ldots a_n \text{) new) } \textbf{HandleMessage}}$$

Si può notare che praticamente ogni messaggio inviato provoca la creazione di un'istanza della classe messaggio, eccettuando il messaggio HandleMessage onde evitare un ciclo infinito.
Caratteristiche di questo modello sono:

- Per ogni scambio di messaggi l'esecuzione passa al livello meta, tranne nel caso del messaggio speciale HandleMessage.
- È facile aggiungere nuovi metodi di comunicazione creando sottoclassi della classe Message e passandoli come argomento del metodo send:.
- Non essendoci un meta-oggetto non si possono ottenere informazioni su oggetti o classi (come il numero di istanze di una certa classe) e mantenere informazioni tra le computazioni riflessive.

Quest'ultima caratteristica è una forte limitazione del modello, in quanto inficia la possibilità di effettuare alcuni tipi di elaborazioni come ad esempio monitorare gli oggetti; per ovviare a tale inconveniente si può ricorrere a modelli ibridi in cui l'approccio a materializzazione si fonde con quello a meta-oggetto.
In SmallTalk per implementare il modello a materializzazione del messaggio occorre innanzitutto modificare la classe di sistema Message, aggiungendogli i soliti metodi setSelector: e setArguments: già visti negli altri modelli, il metodo HandleMessage: necessario per gestire effettivamente il messaggio ed anche il metodo apply:.
Di seguito si riporta il codice della classe Message.

SmallTalk n° 1.10 *Classe* Message *per il modello a materializzazione del messaggio.*

```
Object subclass: #Message
 instanceVariableNames: ´selector arguments´
 classVariableNames: ´´

!Message methods!
apply: anObject
  "applica il messaggio self a anObject"
 ^(anObject perform: selector withArguments: arguments).

setArguments: anArray
  "in anArray ci sono gli argomenti del messaggio"
 arguments := anArray.

setSelector: aString
  "aString e´ il selettore del messaggio"
 selector := ( aString asSymbol ).

HandleMessage: anObject
  "gestore del messaggio"
 ^(apply: anObject).
```

Ovviamente è necessario modificare anche la classe di sistema; è sufficiente aggiungere dei metodi, in particolare quelli per iniziare la computazione riflessiva (send:). Proprio da questi ultimi si può facilmente notare che ogni scambio di messaggi da origine ad un nuovo oggetto istanza di Message e quindi la computazione passa sempre a livello meta. In realtà la gestione del metodo send: dà una possibilità in più a questo modello, infatti esso passa sempre a livello meta, ma solo quando è usato il metodo send:, dando in questo modo la possibilità di un comportamento non riflessivo, e più efficiente, ad una certa computazione (possibilità offerta negli altri modelli dalle variabili reflective e meta).
Di seguito si allegano le modifiche effettuate alla classe Object, riportando il codice relativo al metodo send: solo per il caso relativo ai messaggi unari.

SmallTalk n° 1.11 *Modifiche fatte alla classe* **Object** *per ottenere un'architettura riflessiva secondo il modello a materializzazione del messaggio.*

```
nil subclass: #Object
 instanceVariableNames: ´´
 classVariableNames: ´RecursionInError Dependents RecursiveSet´

!Object methods!
send: aSelector To: aMessageClass
  "metodo per il trattamento delle riflessioni (modello a materializzazione del messaggio)"
 ^((aMessageClass new) setSelector: aSelector ; HandleMessage: self).
```

Il funzionamento è semplice; partendo dal presupposto che ogni scambio di messaggio

è gestito dal metodo send:, ne segue che ogni scambio di messaggi dà origine ad un oggetto istanza di una classe derivata da Message; siccome il metodo send: non effettua alcun controllo, usare quest'ultimo per inviare il messaggio HandleMessage: porta sicuramente ad un ciclo[8].
Per modificare il comportamento riflessivo è sufficiente definire una nuova sottoclasse di Message (vedi esempio VerboseMessage) e poi passarla al metodo send:.

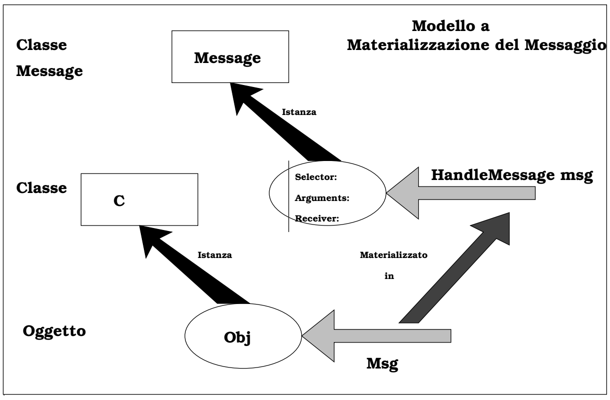


Figura 1.4: Approccio a Materializzazione del messaggio

Siccome non si ha alcun meta-oggetto legato alla classe o all'oggetto non si possono monitorare gli oggetti; limite, come suggerito da Ferber, superabile da modelli ibridi meta-oggetto – materializzazione del messaggio, che hanno la caratteristica di poter agevolmente implementare nuovi scambi di messaggi e di poter monitorare le azioni dell'oggetto; nel paragrafo [1.8] si presenta un modello derivato dal modello a materializzazione del messaggio, che corregge i difetti di quest'ultimo senza essere un modello ibrido.

SmallTalk n° 1.12 *Classe* VerboseMessage.

```
Message subclass: #VerboseMessage
 instanceVariableNames: ´´
 classVariableNames: ´´
 poolDictionaries: ´´

!Message methods!

HandleMessage: anObject
  "gestore del messaggio in modo verbose"
 Transcript
  nextPutAll: ´+++ VerboseHandleMessage: e` stato chiamato il metodo ´ ;
  nextPutAll: selector .
 (arguments isNil)
  ifFalse:
   [Transcript nextPutAll: ´ con ´.
    arguments size storeOn: Transcript.
    Transcript nextPutAll: ´ argomenti´ .
   ].
 Transcript cr.
 ^(super HandleMessage: anObject).
```

[8] Nel caso fosse necessaria una meta computazione è possibile definire una sottoclasse di Message, chiamata ad esempio MetaMessage, che ha una variabile di classe contenente il tipo di messaggio che effettua la meta-meta computazione e una send diversa, che prevede l'attivazione del meta-meta livello evitando così il ciclo infinito; maggiori ragguagli su questa tecnica vengono forniti nella sezione [1.8].

1.7 Analisi di Alcuni Requisiti delle Riflessioni

Di seguito vengono analizzate delle proprietà dei modelli riflessivi, non trattate da Ferber come il ciclo di vita e la granularità delle riflessioni. Lo studio di queste proprietà serve a delineare le qualità che un buon modello deve possedere. Questi requisiti sono tenuti presente nella progettazione del modello proposto nella sezione [1.8].

1.7.1 Ciclo di Vita delle Riflessioni

Con ciclo di vita delle riflessioni si intende il periodo di tempo in cui gli oggetti che compongono il meta livello (meta-classi, meta-oggetti e messaggi) continuano ad esistere ed ad effettuare le loro computazioni riflessive.
Si esamina il ciclo di vita dei modelli sinora presentati:

- **Modello a Meta Classe:** in questo modello la computazione riflessiva è effettuata dalla classe dell'oggetto, perciò, il ciclo di vita di tale entità coincide con il tempo di esecuzione dell'intero programma; la meta-classe esiste indipendentemente dall'esistenza di una sua istanza, per tutta la durata del programma.

- **Modello a Meta Oggetto:** in questo modello, la computazione riflessiva è effettuata da un oggetto distinto dalla classe, tale oggetto (meta-oggetto) può monitorare diversi oggetti al livello base quindi il suo ciclo vitale è legato all'esistenza dei suoi referenti; ogni meta-oggetto esiste, almeno finchè esiste un oggetto al livello base che lo riferisce.

- **Modello a Materializzazione del Messaggio:** in questo modello, la computazione riflessiva è effettuata da un oggetto che incarna un messaggio (metodo attivato da un oggetto al livello base); quindi attivazioni di metodi diversi hanno messaggi diversi che le materializzano. Il ciclo di vita del messaggio è legato al periodo di tempo necessario per effettuare la computazione riflessiva; un oggetto messaggio esiste, almeno dal momento in cui si sale di livello (creazione del messaggio) al momento in cui il flusso computazionale ritorna al livello base.

Concludendo si può osservare che i tre modelli hanno un ciclo vitale differente, dal più lungo (meta-classe) al più breve (messaggio), se l'entità riflessiva permane in memoria più di quanto effettivamente necessita, si ha uno spreco di spazio mentre se ci rimane troppo poco è impossibile mantenere informazioni tra le computazioni riflessive, inficiando quindi la potenza computazionale del modello.

1.7.2 Granularità delle Riflessioni

Per granularità delle riflessioni si intende l'entità minima dell'architettura riflessiva, che può effettuare delle computazioni riflessive distinte da quelle di altre entità dello stesso tipo.
Si esamina la granularità dei modelli sinora presentati:

- **Modello a Meta Classe:** in questo modello, il meta-oggetto coincide con la classe, obbligando quindi le istanze di questa classe ad aver tutte lo stesso comportamento riflessivo (classi diverse hanno comportamenti riflessivi diversi); quindi la granularità è a livello di <u>classe</u>.

- **Modello a Meta Oggetto:** in questo modello, il meta-oggetto è istanza della classe speciale MetaObj non avente, di norma, alcun legame con la classe dell'oggetto che controlla, quindi oggetti istanza della stessa classe possono avere meta-oggetti differenti (oggetti diversi possono avere comportamenti riflessivi diversi); quindi la granularità è a livello di <u>oggetto</u>.

- **Modello a Materializzazione del Messaggio:** in questo modello, il meta-oggetto è sostituito dall'oggetto che incarna il messaggio, quindi gli scambi di messaggi, anche provenienti dallo stesso oggetto e anche nel caso di ripetuti invii dello stesso messaggio, hanno oggetti diversi che li incarnano e quindi un comportamento riflessivo differente (metodi diversi possono avere comportamento riflessivo diverso); la granularità è a livello di <u>attivazione di metodo</u>.

Ognuno dei tre modelli ha una granularità differente, questo significa che ognuno è particolarmente adatto a risolvere problemi con necessità riflessive diverse. Il modello a meta-classe presenta la granularità più grossa, mentre il modello a materializzazione del messaggio quella più fine (i metodi sono usati dagli oggetti e gli oggetti sono istanze delle classi).
Si osserva che è possibile pensare di usare il modello a meta-oggetto in modo che tutti gli oggetti istanza della stessa classe abbiano lo stesso meta-oggetto. Si ha così che con la granularità a livello di oggetto si ottiene quella a livello di classe. Non è altrettanto agevole il viceversa.
Si può anche osservare che il modello a materializzazione del messaggio, pur avendo una granularità più fine e che quindi offre un controllo maggiore, ha le potenzialità limitate dall'impossibilità di effettuare computazioni riflessive di tipo strutturale, mentre il modello a meta-oggetto può effettuare computazioni di questo tipo, ma ha una granularità più grossa.
Nel successivo paragrafo si propone un modello che ha le proprietà di entrambi; la strada seguita per definirlo consiste nel modificare il modello a materializzazione del messaggio, eliminando i suoi difetti.

1.8 Alternativa all'approccio a Materializzazione del Messaggio

L'idea è quella di definire un modello che elimini le limitazioni presenti nel modello a materializzazione del messaggio mantenendone i pregi. In questa sezione, dapprima si esaminano i pregi ed i difetti del modello a materializzazione del messaggio, in seguito si propone una possibile soluzione con il prototipo in SmallTalk.

Difetti del modello a materializzazione del messaggio

La maggior parte delle limitazioni di questo modello sono imputabili alla mancanza di continuità tra computazioni riflessive e quindi all'impossibilità di poter controllare il comportamento dei referenti.
La mancanza di continuità è dovuta al fatto che ogni attività riflessiva è ottenuta materializzando un differente messaggio, questo impedisce di condividere informazioni e mantenere statistiche, ed inoltre presenta anche problemi di efficienza, in quanto si effettuano molte creazioni|distruzioni di oggetti (uno per ogni metodo attivato).
L'impossibilità di monitorare gli oggetti coinvolti è legata alla mancanza di continuità, ma anche al fatto che il messaggio materializzato, in genere[9], non è tenuto a conoscere la struttura del sender o del receiver, questo impedisce operazioni di monitoraggio|debugging. Inoltre per come è stato definito tale modello è anche impossibile avere un meta-meta livello, cioè non si può monitorare l'azione del monitorare (pena l'incappare in un ciclo infinito[10]).

Pregi del modello a materializzazione del messaggio

I pregi di questo modello sono tutti legati alla sua granularità molto fine; questa permette con estrema facilità di definire per lo stesso metodo o metodi diversi, comportamenti riflessivi differenti; come pure la possibilità di raggruppare i metodi in modo da avere comportamento riflessivo analogo. Nel seguito il comportamento riflessivo fornito da un messaggio viene caratterizzato come ***genere del messaggio*** ed è concettualmente associato alla classe di cui il messaggio è istanza.

[9] Anche se la riflessione strutturale non si adatta a questo modello, va comunque sottolineato che non è esclusa a priori. Infatti, escludendo il modello a meta-classe, la riflessione strutturale è una capacità ortogonale per ogni modello, nel senso che può essere inclusa in ogni modello senza farne effettivamente parte.

[10] Questo in effetti non è sempre vero, in quanto si può prevedere una classe messaggio che chiude la catena di creazioni, l'unico requisito è quello di assicurarsi che prima o poi si crei un'istanza di tale classe.

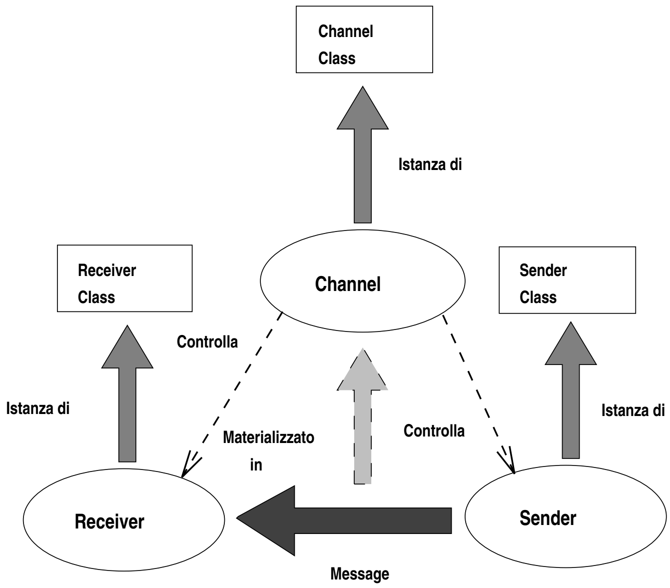


Figura 1.5: **Modello a Materializzazione del Canale**

1.8.1 Introduzione del Canale

Consideriamo come sia possibile risolvere i problemi del modello a materializzazione del messaggio; la continuità è ottenibile evitando di distruggere l'oggetto messaggio una volta svolto il suo compito, ma riutilizzandolo ogniqualvolta sia necessario un oggetto messaggio di quel genere.
È possibile controllare il comportamento degli oggetti coinvolti sia accedendo alle informazioni strutturali, sia manipolando i messaggi scambiati. L'oggetto messaggio incarna un messaggio scambiato tra due oggetti, quindi, per avere un controllo completo del comportamento degli oggetti coinvolti è sufficiente che il messaggio possa accedere al receiver ed in alcuni casi al sender. Unendo i due accorgimenti si evita, inoltre, di far collassare la monitorizzazione dei metodi di oggetti diversi in un unico messaggio, mantenendo così un'identità di messaggio.

Immaginando l'applicazione di un metodo come lo scambio di un messaggio (concetto alla SmallTalk) tra sender e receiver, dove con sender si intende l'oggetto richiedente mentre con receiver si intende l'oggetto che ha il compito di rispondere ad un certo messaggio. L'applicazione di un metodo, allora, può essere vista come la comunicazione di un messaggio lungo un *canale logico* instaurato tra sender e receiver, tale canale si occupa di comunicare la richiesta al receiver ed i valori di ritorno al sender; quindi è possibile materializzare il canale con un oggetto che mantiene il riferimento al sender ed al receiver, e gestisce, non solo il messaggio (cioè il metodo), ma anche l'interazione tra sender e receiver. Trovandosi ad un livello superiore rispetto al sender ed al receiver, tale canale può essere usato anche per effettuare delle computazioni riflessive (meta livello). La computazione riflessiva espletata dal canale è discriminata dal ***genere*** dello stesso.

1.8.2 Modello a Materializzazione del Canale

Questo nuovo modello è una variante dell'approccio a materializzazione del messaggio, e sfrutta l'idea introdotta nei paragrafi precedenti; ogni comunicazione viene gestita da un oggetto canale, che è la materializzazione del canale logico lungo il quale si svolge la normale comunicazione. Il canale è caratterizzato da una terna composta dal sender, dal receiver e dal genere.
Informalmente la semantica dello scambio dei messaggi è:

$$\text{obj } \mathsf{send} \text{ msg } a_1 \ldots a_n \text{ kind} \equiv \begin{array}{l} \text{if (not exist Channel(self, obj, kind)) then} \\ \text{channel\_kind new(self, obj, kind)} \\ \text{Channel(self, obj, kind) } \mathsf{HandleMessage} \text{ msg } a_1 \ldots a_n \end{array}$$

Questo da un punto di vista comportamentale, significa che per un oggetto obj ricevere un messaggio msg equivale a far gestire il messaggio **HandleMessage** msg da un oggetto canale, del genere (kind) specificato, che lo connette al richiedente (self).

Nota Come già sottolineato, le differenze tra questo modello e quello a materializzazione del messaggio sono rappresentate dalla "persistenza" dei canali rispetto ai messaggi e dalla conoscenza da parte dei canali degli oggetti che connettono. Le differenze con il modello a meta-oggetto, invece, sono rappresentate dalla possibilità di monitorare sia il sender che il receiver, dalla creazione su richiesta delle meta-entità e dal concetto di genere del canale.

Le caratteristiche di questo modello sono:

- Granularità più fine nella gestione della computazione riflessiva (si scende al livello di metodo, invece che al livello di oggetto come nel modello a meta-oggetto), consentendo, comunque, qualunque granularità ad un livello più alto, cioè ogni oggetto o classe può essere gestito da canali differenti. Questa è la principale differenza con il modello a meta-oggetto.

- Possibilità di monitorare le comunicazioni mantenendo una continuità che permette la monitorizzazione dell'oggetto o dei due oggetti coinvolti nella comunicazione (si può avere un meta-meta canale per mantenere informazioni complessive sull'oggetto, e un meta-meta-meta canale per mantenere informazioni complessive sulla classe).

- Possibilità di avere più canali per ogni oggetto, quindi di avere la possibilità di fornire comportamenti riflessivi diversi (discriminati dal genere) per ogni richiesta.

- Facilitazione delle computazioni riflessive su singoli metodi (non è più necessario un meta-oggetto factotum).

- I canali vengono creati solo quando effettivamente necessari, cioè quando non esistono già dei canali instaurati tra quegli oggetti e di quel genere, riducendo così al minimo il numero di oggetti di tipo canale che vengono creati.

- Il ciclo di vita della riflessione in questo modello dipende da quello degli oggetti tra cui il canale viene instaurato, quindi un canale esiste finché entrambi gli oggetti tra cui è instaurato esistono, o finché non viene esplicitamente chiuso.

- I canali possono essere strutturati in una gerarchia di ereditarietà, ognuno dei quali ha caratteristiche proprie; tale gerarchia può comprendere, ad esempio, i canali permanenti e quelli temporanei, la cui differenza consiste nel fatto che il primo genere, una volta instaurato viene usato per ogni comunicazione tra quelle entità, quindi sopravvivendo all'applicazione del metodo consente la continuità di informazioni, mentre il secondo materializza il canale ogni volta con un oggetto diverso.

Nota Si è affermato che i canali persistono in memoria, anche dopo che la comunicazione è terminata, ma per quanto rimangono in memoria? Questo ovviamente è legato al periodo in cui è possibile che siano riutilizzati. I canali vengono distinti in base al genere e soprattutto agli oggetti che collegano, quindi nel momento in cui il sender o il receiver cessano di esistere, i canali instaurati tra di essi non hanno più ragione di essere. Nel resto della trattazione anche se non si specifica come, si suppone che i canali vengano rimossi nel momento in cui non hanno più ragione di essere.

Per ottenere una possibile implementazione in SmallTalk innanzitutto si deve definire una classe Channel, la cui principale peculiarità è dovuta alla variabile di classe OpenChannel,[11] che si occupa di tenere traccia dei canali usati in precedenza in modo da poterli riutilizzare nelle successive occasioni. È l'operazione di inizializzazione (from:to:for:) che si occupa di gestire tale dizionario; deve inizializzarlo, se nessuno lo ha fatto prima, ed aggiornarlo nel caso della creazione di un nuovo canale, inoltre è sempre l'operazione stessa a ritrovare il canale nel caso in cui sia già stato creato in precedenza.
Oltre a questo, serve un metodo per gestire la riflessione (HandleMessage), uno per gestire il metodo vero e proprio (apply) ed i metodi necessari per l'inizializzazione

[11] In questa trattazione ci si appoggia alle strutture dati offerte da SmallTalk, in particolare alla classe Dictionary, anche se è possibile implementare strutture dati più efficienti.

del canale (setMessage:, setReceiver: e setSender:).

Nota il canale ha delle variabili contenenti i riferimenti al mittente (sender), al destinatario (receiver) ed al messaggio (message); una domanda spontanea è: "a cosa serve mantenere traccia del mittente?" Innanzitutto serve per effettuare computazioni riflessive strutturali, infatti, tramite tale riferimento si può accedere alla struttura dell'oggetto chiamante ed alla sua classe permettendo, quindi, di monitorare una richiesta completamente. Inoltre permette di differenziare facilmente l'attività riflessiva al variare sia dell'oggetto chiamante che di quello ricevente.

Di seguito si riporta il codice della classe Channel.

SmallTalk n° 1.13 *classe* **Channel**

```
 Object subclass: #Channel
  instanceVariableNames: 'sender receiver message'
  classVariableNames: 'OpenChannel'
  poolDictionaries: ''

!Channel class methods!

from: aSender to: aReceiver for: aMessage
  "crea un nuovo canale se necessario"
 |channel|
 (OpenChannel isNil)
  ifTrue:[OpenChannel := Dictionary new]
  ifFalse:
   [channel := OpenChannel at: ( Array with: self with: aSender with: aReceiver )
              ifAbsent: [nil]
   ].
 (channel isNil)
  iftrue:
   [channel := (self new) setSender: aSender ;
              setReceiver: aReceiver ; setMessage: aMessage.
    OpenChannel at: (Array with: self with: aSender with: aReceiver) put: channel
   ]
  ifFalse: [channel setMessage: aMessage].
 ^channel.

!Channel methods!

HandleMessage
  "gestore del messaggio"
 ^(self apply).

apply
  "applica il messaggio"
 ^(receiver apply: message).

setSender: aSender
  "inizializza il sender del canale"
 sender := aSender.

setReceiver: aReceiver
```

```
  "inizializza il receiver del canale"
 receiver := aReceiveer.

setMessage: aMessage
  "inizializza il messaggio da inviare"
 message := aMessage.
```

Alla classe di sistema Object vengono aggiunti il metodo send:, che effettua il passaggio della computazione dal livello base al meta livello e il metodo apply:, che applica un messaggio all'oggetto ricevente effettuando così il passaggio dal meta-livello al livello base (in realtà è richiamata dal metodo apply presente nella classe Channel o nelle sue discendenti, ed è necessaria per effettuare l'esecuzione di messaggi attivati quando il canale non viene creato).
Di seguito si riportano le modifiche effettuate alla classe Object, come al solito il metodo send: è presentato solo per il caso dei messaggi unari.

SmallTalk n° 1.14 *Modifiche effettuate alla classe* **Object** *per implementare il modello a materializzazione del canale.*

```
nil subclass: #Object
 instanceVariableNames: ´´
 classVariableNames: ´RecursionInError Dependants RecursiveSet´
 poolDictionaries: ´´

!Object methods!

apply: aMessage
  "applica il messaggio all´oggetto al livello base"
 ^( self perform: ( aMessage selector ) withArguments: ( aMessage arguments ) ).

send: aSelector by: aChannelClass from: anObject
  "metodo per il trattamento della riflessione (modello a materializzazione del canale)"
 ^((aChannelClass from: anObject to: self for:
    ((Message new) setSelector: aSelector)) HandleMessage).
```

Il funzionamento è il seguente: ogniqualvolta sia richiesta l'esecuzione di un metodo, la richiesta è effettuata mandando il messaggio send: all'oggetto a cui è richiesta tale computazione, specificando il genere di computazione riflessiva che si vuole ottenere tramite la classe del canale che viene usata per la materializzazione (tale classe deve essere la classe Channel, descritta in precedenza o una sua discendente). Il metodo send: si occupa della materializzazione del canale (from:to:for:), usandone, se possibile, uno già esistente (di quel genere ed instaurato tra quegli oggetti) se no creandone uno nuovo.
A questo punto il canale esegue la HandleMessage effettuando così la computazione riflessiva ed il ritorno al livello base.
Si può osservare che ogni metodo ha il suo proprio comportamento riflessivo definito dal genere di canale usato (granularità a livello di metodo); si ha anche la possibilità di mantenere informazioni relative alle computazioni riflessive, grazie al riuso

dei canali già esistenti (continuità nella computazione riflessiva), e la possibilità di accedere alla struttura degli oggetti, grazie al fatto che i canali possono risalire alle classi sia del mittente (sender class) che del ricevente (receiver class) (riflessione strutturale).
Essendo il modello a materializzazione del canale una variante del modello a materializzazione del messaggio, presenta anch'esso il rischio di un ciclo infinito di creazioni di canali, se si tentasse di attivare la HandleMessage tramite il metodo send: (per confronto vedere la sezione [1.6.2.1]). In alcuni problemi, comunque può essere necessario poter effettuare delle meta-meta computazioni (ad esempio per raggruppare a livello di oggetto o di classe, informazioni inerenti le comunicazioni effettuate da un oggetto e mantenute dai canali da lui utilizzati). Questo è possibile definendo una classe discendente di Channel (ad esempio MetaChannel) che attiva la meta-meta-computazione; quest'ultima classe ha una variabile globale che contiene il genere di computazione riflessiva che deve avere il metodo (canale da instaurare per la meta-meta computazione), un metodo per modificare tale slot (meta) ed un metodo che attiva la meta-meta computazione (HandleMessage), filtrando la normale computazione riflessiva. In modo analogo è possibile gestire una gerarchia di meta-livelli. Il codice in SmallTalk è riportato di seguito.

SmallTalk n° 1.15 *classe* MetaChannel *per la gestione di meta-meta computazioni.*

```
Channel subclass: #MetaChannel
 instanceVariableNames: ''
 classVariableNames: 'MetaChannelClass'
 poolDictionaries: ''

!MetaChannel class methods!

meta: aChannelClass
  "cambia il tipo del meta canale"
 MetaChannelclass := aChannelClass.

!MetaChannel methods!

HandleMessage
  "gestore del messaggio (sale di un livello nella meta computazione)"
 ^(self send: 'HandleMessage' by: MetaChannelClass from: self).
```

1.9 Un Esempio Completo

Si supponga di voler scrivere una classe, che mantenga un file copia per ogni file creato, con le operazioni da essa fornita, e che le operazioni di scrittura e di lettura concernenti tali file, mantengano consistenti le due copie, cioè per ogni file le modifiche (scritture) vengono fatte prima sull'originale e poi sulla copia, mentre le operazioni di lettura verificano che i dati letti dal file originale coincidano con quelli

letti dalla copia; in tal caso la lettura va a buon fine altrimenti si incorre in un errore ed occorre segnalarlo tramite un apposito codice d'errore, ad esempio -1.
Si vuole rendere la gestione del file copia completamente trasparente all'utente finale, quindi questa è un'ottima occasione per utilizzare la metodologia riflessiva. Le possibili operazioni di cui è dotata la classe che si sta costruendo sono open:; che crea il file, read che legge un carattere dal file e write: che scrive un carattere sul file. Si vuole che:

- All'atto della creazione di un file, venga creato un secondo file, usato come copia di sicurezza dell'originale.
- L'operazione di scrittura di un carattere sul file, scriva tale carattere sul file originale ed automaticamente anche sulla copia.
- L'operazione di lettura legga un carattere dal file originale, verificando con la copia la correttezza del carattere che è stato letto.

Le operazioni di controllo, come già affermato, devono avvenire in modo trasparente, ad insaputa dell'utente. Se si applica il modello a meta-classe per costruire tale classe, è HandleMessage:To: che gestisce i vari casi. Questo metodo si presenta come un intricato susseguirsi di istruzioni condizionali, tanto più intricato quante più possibilità ci sono. Un metodo per gestire la riflessione così strutturato viene detto: metodo factotum.
È inoltre necessario aggiungere uno slot contenente il riferimento al file copia tra le variabili di istanza della classe. Questo è necessario per non perdere le tracce del file copia. Non può essere inserito, nonostante sia un'informazione riflessiva, con le altre informazioni riflessive, perché essendo mantenute nella classe, due oggetti, istanza di tale classe, legati a file differenti, avrebbero lo stesso file copia associato. Di seguito si riporta il codice di tale classe.

SmallTalk n° 1.16 *Classe* **SafeDiskAccess** *realizzata usando il Modello a Meta-Classe*

```
Object subclass: #SafeDiskAccess
 instanceVariableNames: 'fileStream fixedFileStream'
 classVariableNames: ''
 poolDictionaries: ''

!SafeDiskAccess class methods!

HandleMessage: aMessage To: anObject
  "gestore della riflessione"
 |fr r|
 ((aMessage selector) sameAs: 'open:')
  ifTrue:[fixedFileStream := pathName: (((aMessage arguments)at:1),'.stab')]
  ifFalse:
   [((aMessage selector) sameAs: 'write:')
    ifTrue:
```

```
        [anObject apply: aMessage.
         fixedFileStream nextPut:((aMessage arguments)at:1).
        ]
       ifFalse:
        [r := anObject apply: aMessage.
         fr := fixedFileStream next.
         (r == fr)
          ifTrue:[^r]
          ifFalse:[^-1].
        ]
      ]


!SafeDiskAccess methods!

open: aString
  "apre il file"
 fileStream := pathName: aString.

read
  "legge il carattere successivo alla posizione corrente"
 ^(fileStream next).

write: aChar
  "scrive un carattere sul file"
 fileStream nextPut: aChar.
```

Si supponga ora di voler definire una seconda classe anch'essa con operazioni di creazione, lettura e scrittura e stesso comportamento riflessivo, però completamente distinta dalla classe appena costruita, quindi senza alcuna relazione di ereditarietà con quest'ultima. Non è importante la motivazione di tale scelta, ma piuttosto il fatto che le peculiarità proprie del modello a meta-classe costringono a riscrivere, anche se completamente identica, la parte riflessiva (classi distinte non possono condividere la stessa meta-classe). Occorre separare la classe, dall'entità che effettua la computazione riflessiva, permettendone così un più agevole riuso. Negli altri tre modelli ciò è possibile; si riscrive l'esempio adottando il modello a materializzazione del messaggio.
Usando questo modello si ha un messaggio distinto per ogni metodo che attua la computazione riflessiva. Così si perde la necessità di un metodo factotum (beneficio di una granularità più fine), e si riesce a separare l'entità che effettua la computazione riflessiva dalla classe, permettendo la riusabilità di tale entità per più classi distinte con le stesse necessità riflessive. Putroppo per i difetti propri di questo modello il riferimento al file copia continua ad essere tenuto nell'oggetto da monitorare. Di seguito è riportato il codice di tale soluzione.

SmallTalk n° 1.17 *Classe* **SafeDiskAccess** *realizzata usando il Modello a Materializzazione del Messaggio*

```
Message subclass: #FixedWriteMessage
 instanceVariableNames: ´´
```

```
 classVariableNames: ´´
 poolDictionaries: ´´

!FixedWriteMessage methods!

HandleMessage: anObject
  "gestore della riflessione"
 anObject fixedFileStream nextPut: ((self Arguments) at:1).
 ^(apply: anObject).

Message subclass: #FixedOpenMessage
 instanceVariableNames: ´´
 classVariableNames: ´´
 poolDictionaries: ´´

!FixedOpenMessage methods!

HandleMessage: anObject
  "gestore della riflessione, crea il file copia"
 (anObject fixedFileStream) := pathName:(((self Arguments)at:1),´.stab´).
 ^(apply: anObject).

Message subclass: #FixedReadMessage
 instanceVariableNames: ´´
 classVariableNames: ´´
 poolDictionaries: ´´

!FixedReadMessage methods!

HandleMessage: anObject
  "gestore della riflessione, lettura con controllo di correttezza"
 |r fr|
 fr := (anObject fixedFileStream) next.
 r := apply: anObject.
 (fr == r)
   ifTrue:[^r]
   ifFalse:[^-1].

Object subclass: #SafeDiskAccess
 instanceVariableNames: ´fileStream fixedFileStream´
 classVariableNames: ´´
 poolDictionaries: ´´

!SafeDiskAccess methods!

open: aString
  "crea il file"
 fileStream := pathName: aString.

read
  "legge un carattere"
 ^(fileStream next).

write: aChar
  "scrive un carattere sul file"
 fileStream nextPut: aChar.
```

Si supponga di voler arricchire ulteriormente la classe mantenendo un'informazione sul numero di errori riscontrati in fase di lettura, ovviamente per i già citati difetti del modello a materializzazione del messaggio non è possibile mantenere informazioni tra le computazioni riflessive. È necessario, quindi, riscrivere l'esempio usando il modello a meta-oggetto, che permette di mantenere tali informazioni; chiaramente il mantenimento delle informazioni è possibile anche con il modello a meta-classe, però questo fornisce una scarsa riusabilità della componente riflessiva cosa che il modello a meta-oggetto invece permette. Si definisce, quindi, una classe discendente di MetaObject (MetaSafeDiskAccess) e qui si tiene un dizionario (errors) in cui per ogni oggetto monitorato si mantiene il numero di errori rilevati; il metodo HandleMessage si occupa di gestire la riflessione. In questo modello il metodo che gestisce la riflessione è tornato ad essere un metodo factotum. Di seguito è riportato il codice relativo all'esempio.

SmallTalk n° 1.18 *Classe* **SafeDiskAccess** *realizzata usando il Modello a Meta Oggetto*

```
MetaObject subclass: #MetaSafeDiskAccess
 instanceVariableNames: ´errors´
 classVariableNames: ´´
 poolDictionaries: ´´

!MetaSafeDiskAccess class methods!

new
  "crea un nuovo meta-oggetto"
 ^((super new) initialize).

!MetaSafeDiskAccess methods!

initialize
  "inizializza le variabili d´istanza"
 (self errors) := Dictionary new.

HandleMessage: aMessage
  "gestore della riflessione"
 |fr r|
 ((aMessage selector) sameAs: ´open:´)
  ifTrue:
   [errors at:(self obj) put:0.
    ((self obj) fixedFileStream) := pathName: (((aMessage arguments) at:1),´.stab´).
    ^(self obj) apply: aMessage.
   ]
  ifFalse:
   [((aMessage selector) sameAs: ´write:´)
    ifTrue:
     [(self obj) apply: aMessage.
      ((self obj) fixedFileStream) nextPut: ((aMessage arguments) at: 1).
     ]
    ifFalse:
     [r := (self obj) apply: aMessage.
      fr := ((self obj) fixedFileStream) next.
      (r == fr)
```

```
        ifTrue:[^r]
        ifFalse:
         [errors at:(self obj) put:((errors at:(self obj))+1).
          ^-1.
         ].
      ].
    ].

Object subclass: #SafeDiskAccess
 instanceVariableNames: 'fileStream fixedFileStream'
 classVariableNames: ''
 poolDictionaries: ''

!SafeDiskAccess methods!

open: aString
  "crea il file"
 fileStream := pathName: aString.

read
  "legge un carattere"
 ^(fileStream next).

write: aChar
  "scrive un carattere sul file"
 fileStream nextPut: aChar.
```

Si supponga di voler aggiungere un metodo alla classe SafeDiskAccess o costruire una classe discendente di quest'ultima con un metodo in più, ad esempio questo potrebbe essere writeString: che scrive sul file una stringa. Continuando sulla strada seguita sinora, cioè adattando il modello a meta-oggetto, la presenza del metodo factotum comporta la modifica di tale metodo, per far si che preveda anche il metodo nuovo (una nuova alternativa o una condizione modificata nell'istruzione condizionale) rompendo così l'encapsulation della classe MetaObj. Questo problema può essere evitato usando un modello con una granularità più fine, quindi sia il modello a materializzazione del messaggio che il modello a materializzazione del canale possono andare bene, però poiché il primo si è già rilevato limitato nel mantenere informazioni tra computazioni riflessive, si usa il modello a materializzazione del canale.
Questo modello, invece che modificare la HandleMessage del meta oggetto, porta a creare una nuova classe discendente di Channel che gestisce la riflessione per il nuovo metodo (caso in cui i metodi sono completamente distinti tra loro, nel modello a meta-oggetto si doveva aggiungere un'alternativa all'istruzione condizionale) oppure al riuso di una classe discendente di Channel già esistente (caso in cui i due metodi richiedono lo stesso comportamento riflessivo, nel modello a meta-oggetto occorreva modificare la clausola di un'istruzione condizionale, in modo che contemplasse anche il nuovo metodo). Nel nostro caso il metodo aggiunto, writeString:, può essere gestito come il metodo write: e quindi viene attivato dalla send:, usando un canale FixedWriteChannel; come si può notare anche l'informazione riflessiva fixedFileStream fa parte dell'entità riflessiva, aumentando l'encapsulation della riflessione.

Il codice dell'esempio poteva essere più semplice, usando una classe comune da cui far discendere le tre classi (FixedOpenChannel, FixedReadChannel e FixedWriteChannel) e contenente le informazioni comuni, ma si è preferito strutturarla in questo modo per far vedere completamente le potenzialità del modello, come la continuità dell'informazione tra computazioni riflessive, la condivisione delle informazioni tra canali diversi, e meta-meta livelli di computazione riflessiva.
Si può obiettare, che l'aggiunta di un metodo completamente distinto nella classe al livello base, porta alla rottura dell'encapsulation della classe CommonChannel come nel modello a meta-oggetto, soltanto che questa classe nel modello a materializzazione del canale può essere progettata scomposta, per evitare tale inconveniente. Di seguito è riportato il codice dell'esempio.

SmallTalk n^o 1.19 *Classe* **SafeDiskAccess** *realizzata usando il Modello a Materializzazione del Canale*

```
Channel subclass: #FixedOpenChannel
 instanceVariableNames: 'fixedFileStream'
 classVariableNames: ''
 poolDictionaries: ''

!FixedOpenChannel methods!

HandleMessage
  "gestore del messaggio"
 (fixedFileStream := pathName: (((message Arguments) at: 1),'.stab').
 ^(self send: 'apply' by: CommonChannel from self).

Channel subclass: #FixedWriteChannel
 instanceVariableNames: 'fixedFileStream'
 classVariableNames: ''
 poolDictionaries: ''

!FixedWriteChannel methods!

HandleMessage
  "gestore del messaggio"
 |return|
 return := self send: 'apply' by: CommonChannel from: self.
 fixedFileStream nextPut: ((message Arguments) at: 1).
 ^return.

Channel subclass: #FixedReadChannel
 instanceVariableNames: 'fixedFileStream errors'
 classVariableNames: ''
 poolDictionaries: ''

!FixedReadChannel methods!

HandleMessage
  "gestore del messaggio"
 |r fr|
 r := self send: 'apply' by: CommonChannel from: self.
 fr := fixedFileStream next.
 (fr == r)
```

```
   ifTrue:[^r]
   ifFalse:
    [errors := errors + 1.
     ^-1.
    ].

Channel subclass: #CommonChannel
  instanceVariableNames: ´fixedFileStream´
  classVariableNames: ´´
  poolDictionaries: ´´

 !CommonChannel class methods!

 getChannel: aBlock
   "recupera il canale che soddisfa a Block"
  self OpenChannel associationsDo:[:each|(aBlock value: each key)
       ifTrue:[^(each value)]].
  ^nil.

 from:aSender to:aReceiver for:aMessage
   "cerca di recuperare un canale il cui sender del sender
    sia uguale e di tipo self se esiste, altrimenti lo crea"
  |channel|
  (OpenChannel isNil)
   ifTrue:[OpenChannel := Dictionary new]
   ifFalse:
    [
     channel := getChannel: [ :k | (((k at:3) = self) and:[(k at:1) == aSender.sender])]
    ].
  (channel isNil)
   ifTrue:
    [channel:=(self new)setSender: aSender ;
              setReceiver: aReceiver ; setMessage: aMessage.
     OpenChannel at:(Array with: self with: aSender with: aReceiver) put: channel.
    ]
    ifFalse:[channel setMessage: aMessage].
  ^channel.

 !CommonChannel methods!

 HandleMessage
   "gestore del messaggio"
  (sender class == FixedOpenChannel)
   ifTrue:[fixedFileStream := sender fixedFileStream]
   ifFalse:[(sender fixedFileStream := fixedFileStream].
  ((sender class == FixedReadChannel) and:[(sender errors) isNil])
   ifTrue:[(sender errors):=0].
  ^(self apply).

 Object subclass: #SafeDiskAccess
  instanceVariableNames: ´fileStream ´
  classVariableNames: ´´
  poolDictionaries: ´´

 !SafeDiskAccess methods!

 open: aString
   "crea il file"
  fileStream := pathName: aString.
```

```
read
  "legge un carattere"
 ^(fileStream next).

write: aChar
  "scrive un carattere sul file"
 fileStream nextPut: aChar.

writeString: aString
  "scrive una stringa sul file"
 fileStream nextPut: aString.
```

1.10 Conclusioni

All'interno di questo capitolo sono state presentate le riflessioni: concetti e caratteristiche; si sono esaminati i principali modelli di implementazione esistenti (modello a meta-classe, modello a meta-oggetto e modello a materializzazione del messaggio) analizzandone pregi e difetti; inoltre si è proposto un nuovo modello (modello a materializzazione del canale) con potenzialità maggiori di quelli precedenti, illustrando con un esempio, i limiti degli altri modelli ed evidenziando come tali limiti vengano superati dal nostro modello.

È interessante osservare cosa accade se anche i messaggi alle classi vengono inviati tramite il metodo send:.
Usando il modello a meta-oggetto o il modello a meta-classe non succede niente, in quanto per nessuno dei due modelli viene a priori associato un meta-oggetto alla classe, ricordo che le classi essendo oggetti ereditano da Object (vedi sezione [1.5.1]), e come tali hanno, a seconda del caso, lo slot reflective o meta. Per monitorare le classi, con questi modelli, è sufficiente inizializzare lo slot reflective o lo slot meta, a seconda del caso, adeguatamente. Usando il modello a materializzazione del messaggio o il modello a materializzazione del canale, viene semplicemente creata un'istanza di Message|Channel ed i messaggi in questione vengono processati come tutti gli altri.

Un'altra questione è come è possibile monitorare la computazione riflessiva.
Nel modello a materializzazione del messaggio questo è normalmente impossibile perché dà origine ad un ciclo infinito di creazioni di messaggi. Nel modello a meta-oggetto è sufficiente munire il meta-oggetto di meta-oggetto; occorre prestare attenzione che si può originare una situazione ciclica in cui un meta-oggetto è meta-oggetto di un meta-oggetto, che è suo meta-oggetto; tale situazione è evitabile modificando il metodo meta: in modo che controlli tali situazioni. Nel modello a meta-classe si deve solo impostare lo slot reflective della classe, per il resto tutti i metodi necessari vengono ritrovati grazie alla particolare gerarchia di SmallTalk (vedere la

figura [1.2] per maggiori chiarimenti). Per quanto riguarda il modello a materializzazione del canale si veda la sezione [1.8] in cui viene presentato un esempio di questa situazione.
A questo punto si può riscrivere l'esempio di riflessione visto nella sezione [1.5.1] sviluppato per uno dei modelli visti, si sceglie il modello a meta-oggetto. In questo caso nessuna modifica deve essere fatta alla classe Point, tutto il controllo è eseguito da un oggetto istanza della classe StatisticMetaObject il cui codice è riportato di seguito. Il seguente programma:

```
|statistica punto puntor|

statistica := (StatisticMetaObject new).
Point send: 'meta:' with: statistica.
puntor := Point send: 'x:y:' with: 0 with: 0.
puntor send: 'meta:' with: statistica.
punto := Point send: 'x:y:' with: 1 with: 1.
punto := Point send: 'x:y:' with: 2 with: 2.
puntor send: 'statistics'.
```

ritorna la seguente riga:

```
Esistono 3 istanze.
```

Da questo programma si può ulteriormente osservare che, rispetto al solo SmallTalk, si è guadagnata la proprietà di trasparenza.

SmallTalk n° 1.20 *classe* **StatisticMetaObject**

```
MetaObject subclass StatisticMetaObject
 instanceVariableNames: 'instace'
 classVariableNames: ''
 poolDictionaries: ''

!StatisticMetaObject!

setInstance: aNumber
  "inizializza instance"
 instance := aNumber.

incInstance
  "incrementa di uno le istanze"
 instance := instance + 1.

HandleMessage: aMessage
  "gestore metodi riflessivi ( override quello presente nella
   sua super classe ) fornisce informazioni statistiche sul
   numero di istanze della classe che monitorizza."
 ((aMessage selector) sameAs: 'statistics')
  ifTrue:
   [Transcript nextPutAll: 'Esistono '.
    instance storeOn: Transcript.
    Transcript nextPutAll: ' istanze' ; cr.
   ]
  ifFalse:
   [((aMessage selector) sameAs: 'x:y:')
```

```
        ifTrue:[ incInstance. ].
        ^(super HandleMessage: aMessage).
    ].
```

2 Riflessioni e Linguaggi Fortemente Tipati

2.1 Introduzione

Nel capitolo [1] sono stati presentati i principali modelli di riflessione, dandone delle possibili implementazioni in un linguaggio interpretato, nel caso specifico SmallTalk. Dei quattro modelli presentati, il modello a meta-oggetto e il modello a materializzazione del canale si sono dimostrati i più flessibili; comunque nella maggioranza delle applicazioni realizzabili con metodologie riflessive (una su tutte la fault tolerance), è richiesta un'efficienza superiore rispetto a quella ottenibile con un linguaggio interpretato.
Scopo di questo capitolo è quello di unire il meccanismo delle riflessioni ad un linguaggio compilato ed in particolare ad un linguaggio compilato e fortemente tipato staticamente. Fermo restando la componente object oriented del linguaggio, i possibili candidati considerati sono C++ ed Oberon.[1]
La trattazione prende in considerazione soltanto i modelli che si sono mostrati più flessibili (modello a meta-oggetto e modello a materializzazione del canale).

2.2 Riflessioni nei Linguaggi Fortemente Tipati

L'intenzione è quella di costruire un ambiente riflessivo sull'architettura che si ha a disposizione e non dentro la stessa, cioè si vuole determinare un insieme di meccanismi, atti a gestire il comportamento riflessivo di un'eventuale applicazione sviluppata con essi, sfruttando i costrutti e la potenza espressiva propria del linguaggio adottato. Procedendo in questo modo è possibile definire un linguaggio di programmazione

[1] Esistono diversi altri linguaggi object-oriented aventi la caratteristica di essere fortemente tipati, come ad esempio EIFFEL, Objective C, Simula etc..., però non è possibile, nell'ambito di questo lavoro, sperimentarli tutti; si è dovuto, quindi, effettuare una scelta, questa è ricaduta sul C++ e Oberon per i seguenti motivi: il C++ attualmente è il linguaggio ad oggetti più utilizzato, sia in ambito universitario che industriale e come tale doveva essere preso in considerazione, per quanto riguarda Oberon, la sua scelta va ad inquadrarsi in un progetto a lungo termine riguardante estensioni di questo linguaggio (vedere EOS nell'appendice [B]).

avente capacità riflessive, senza doverne costruire un compilatore, ma limitandosi ad incapsulare tali capacità riflessive in specifiche librerie usate dai programmatori.
Nel caso generale di un linguaggio ad oggetti fortemente tipato, i problemi per introdurre i meccanismi riflessivi sono dovuti alla presenza dei tipi ed al binding statico. La tipazione forte, in combinazione con il binding statico, da origine al seguente problema: la tipazione forte obbliga la definizione completa dei metodi e dei loro parametri, inoltre per poter gestire la riflessione occorre che ogni metodo conosca l'oggetto riflessivo e quindi il suo tipo, ma è impossibile al momento della stesura del codice che gestisce la riflessione prevedere tutti i tipi dei possibili oggetti riflessivi. Questo problema è in parte risolubile sfruttando le proprietà dell'ereditarietà; se si ha una classe A ed una classe sua discendente B, un oggetto b istanza della classe B può essere assegnato ad una variabile a di tipo A. I metodi in questione, quindi, devono prevedere, come parametro, un oggetto di tipo base da cui tutti i possibili oggetti riflessivi ereditano.
Il binding statico, però per la corretta compilazione dei programmi, concede di attivare solo i metodi definiti per la classe che è tipo statico di una certa entità e non quelli appartenenti alla classe che rappresenta il tipo dinamico di quell'entità, quindi nel caso precedente delle classi A e B, dopo l'assegnazione (a=b) si possono usare esclusivamente i metodi di A e non quelli di B.
Dal momento che come visto per SmallTalk, anche per questa classe di linguaggi, la riflessione si basa su una modifica del meccanismo di attivazione dei metodi, essa (come si vede successivamente durante la disamina dei linguaggi considerati) interferisce significativamente con le regole di binding statico (si osserva, per inciso, che il problema non si pone utilizzando linguaggi object-oriented non tipati come SmallTalk).

2.2.1 Riflessioni in C++

In C++ si può fare "tutto", purtroppo non sempre in modo elegante ed immediato. Di seguito si descrivono i problemi che si incontrano nell'introdurre la riflessione nel C++ e le relative soluzioni, si sottolinea che è possibile costruire un preprocessore atto a gestire automaticamente le soluzioni ai problemi che si descrivono.

***Higher-order methods o metodi di ordine superiore*[2]:** per poter implementare i due modelli, modello a meta-oggetto e modello a materializzazione del canale, usando il C++ si deve supporre di poter passare, tramite puntatore, ad una funzione o ad un metodo anche un altro metodo, oltre le normali funzioni, cioè si suppone che il linguaggio supporti i metodi di ordine superiore.
Questa è un'ipotesi utile solo per rendere più chiara la trattazione, in quanto il passaggio dei metodi e relativa attivazione da parte dell'oggetto interessato possono

[2]Metodo di ordine superiore: è un metodo, che, come dominio, può avere il tipo funzione o sue composizioni, e quindi accetta un metodo come parametro.

essere, comunque, realizzati, usando un raffinato stratagemma, descritto attraverso il seguente esempio.
Supposto di avere una classe person, la quale possiede un metodo print, si costruisce una funzione enc_print (nel seguito, le funzioni definite per tale scopo, verranno dette incapsulatori), che accetta come parametri un oggetto di tipo person e si limita, semplicemente, ad effettuare l'attivazione del metodo print dell'oggetto passatogli. Nel caso il metodo in questione avesse dei parametri, quest'ultimi sarebbero parametri del relativo incapsulatore. Gli incapsulatori, essendo funzioni, possono essere passati ed attivati tramite un puntatore (caratteristica tipica del C presente anche nel C++).

***Inlining higher order methods*:** questo artifizio è utile anche per gestire metodi o funzioni *inline*, il cui codice, in fase di compilazione, viene sostituito a quello della chiamata. Questi metodi o funzioni inline ovviamente, non possiedono un indirizzo da passare come parametro; incapsulandoli invece in un incapsulatore, possono essere trattati come una qualsiasi funzione.
Procedendo in questo modo, però si perde il vantaggio dell'espansione inline, trasformando i metodi inline in normali metodi. Nella costruzione di applicazioni riflessive occorre valutare attentamente la necessità o meno dell'uso della riflessione sui metodi inline, fermo restando che si possono usare sia in modo riflessivo che non riflessivo. Nel seguito comunque si suppone di poter passare ad altre funzioni, i metodi e le funzioni inline come puntatori senza esplicitare sintatticamente l'uso degli incapsulatori. Nel riquadro [2.1] è mostrato il codice relativo al suddetto esempio.

```
#include <stream.h>

class person {
  public:
    char *name;
    int age;
   // constructor
    person( char *n, int a ) : name(n), age(a) {};
   // methods
    void print( void ) {cout << "*** " << name << " " << age << "\n";};
    void incage( void ) {++age;};
  };

void enc_print( person p ) { p.print(); };
```

Codice C++ 2.1: esempio di incapsulatore

***High order methods e Strong Typing*:** altro problema legato al passaggio dei metodi|funzioni come parametri di altri metodi, è dovuto alla definizione del tipo del parametro; il C++ è più restrittivo sul type checking statico dei parametri rispetto

al C, per il quale, non specificando alcun parametro, la funzione può essere attivata con un numero di parametri a scelta, rischiando, al più, un errore in esecuzione. Nel C++ bisogna specificare tutti i parametri ed usarli consistentemente. Nella trattazione si usa una sintassi stile C supponendo inoltre che la chiamata sia attivabile con parametri di tipo *void ** in luogo dei parametri del tipo necessario, lasciando al linguaggio la corretta interpretazione dei tipi (alcuni compilatori si comportano effettivamente così, limitandosi a sollevare un warning, come nel caso del gnu C++, anche se lo standard del C++ prevede una segnalazione d'errore). Anche questo può essere ottenuto tramite gli incapsulatori facendo sì che i parametri siano passati tramite un array di puntatori a *void* e che l'incapsulatore si occupi della chiamata con il numero ed il tipo dei parametri corretto.
Ad esempio, supponiamo che la classe person abbia i suoi due attributi di stato name ed age definiti come privati, cioè accessibili solo tramite dei selettori e che la classe sia dotata dei metodi per inizializzarli set_name e set_age che prevedono come parametri, rispettivamente, una stringa di caratteri ed un intero. Allora gli incapsulatori per set_name (enc_set_name) e per set_age (enc_set_age) gestiscono il cast dei parametri per chiamare i metodi che incapsulano nel modo corretto. Nel riquadro [2.2] è mostrato il codice dei due incapsulatori descritti.

```
void enc_set_name( person p, void **parameter )
{
  p.set_name( (char *)(*parameter) );
}

void enc_set_age( person p, void **parameter )
{
  p.set_age( *((int *)(*parameter) );
}
```

Codice C++ 2.2: esempio di incapsulatori con parametri

***Run-time type identification*:** altra ipotesi necessaria è quella della presenza di meccanismi per effettuare il controllo, al momento dell'esecuzione, del tipo dinamico di un oggetto (RTTI, run-time type identification); tali meccanismi permettono di utilizzare a run-time il binding dinamico in luogo di quello statico, relativamente ad un oggetto, permettendo così di attivare metodi, propri di una classe derivata, tramite il puntatore alla sua classe base.
Si supponga, ad esempio, di derivare da person la classe employee con un attributo salary ed un metodo total_salary per il calcolo dello stipendio annuo, e si consideri una funzione f che prevede un argomento p di tipo person *. Allora se nel corpo della stessa si vuole accedere (sapendo che p in realtà è di tipo employee *) all'attributo salary o attivare il metodo total_salary, questo ci costringe ad usare un cast senza la verifica dei tipi perdendo così la tipazione forte, con possibili inconsistenze nei tipi,

rilevato a run-time, oppure i compilatori non permettono l'accesso a campi non appartenenti alla classe di p, quindi p→salary e p→total_salary() sfociano in un errore. Adottando, invece, metodologie di programmazione RTTI, che permettono l'accesso al tipo dinamico di p nel caso in cui p sia effettivamente un employee, si possono utilizzare tali metodi|attributi. Nel riquadro [2.3] è mostrato il codice del suddetto esempio. Anche l'ipotesi dell'esistenza di un meccanismo per RTTI è facile da

```
#include ''person.h''

class employee : public person {
  public:
    long salary;
   // constructor
    employee( char *n, int a, long s ) : person( n, a ), salary( s ) {};
   // method
    long total_salary() {return( 12*salary );};
   };

void f( person *p )
{
  p.total_salary(); // error
  if ( typeid(p) == Typeid<employee> )
   {
     employee *e = (employee *)p; // downward cast
     e.total_salary();           // correct
   }
 };
```

Codice C++ 2.3: esempio di RTTI

soddisfare, per due buoni motivi: il primo è il fatto che basta inserire nella classe un attributo o una funzione che mantenga|calcoli il tipo dinamico dell'oggetto, potendo così effettuare un cast controllato; il secondo è che lo stesso Stroustrup, creatore del C++, ha proposto un'estensione dello standard del linguaggio che prevede, tra l'altro, meccanismi per gestire il RTTI e che risulta già supportato nei compilatori più recenti come l'ultima release del Borland C++. Per quanto riguarda la sintassi dei costrutti per la gestione del RTTI, si fa riferimento a [Stroustrup-Lenkov 92], in cui Stroustrup e Lenkov hanno proposto dei costrutti per il test sui tipi (typeid(x) e Typeid<x_type>()) ed un operatore di cast controllato o downward cast, che permette di forzare il tipo da quello base ad un tipo suo derivato se e solo se l'oggetto ha come tipo dinamico effettivamente tale tipo (class_Y *y = (class_Y *)x).
Alla proposta di Stroustrup, per i nostri scopi, sarebbe necessario aggiungere un costrutto che permetta di gestire il binding dinamico indipendentemente dal tipo dinamico che ha l'oggetto, cioè, invece di effettuare un cast esplicito in cui occorre conoscere il tipo dinamico, si usa un costrutto (dbind) che permette di sostituire a tempo di esecuzione il tipo statico di un oggetto con il suo tipo dinamico, rendendo così inutile la conoscenza del tipo dinamico.

Nel caso generale per poter accedere ai campi propri del tipo dinamico di un oggetto senza conoscerne il tipo dinamico, occorre un costrutto del linguaggio, come dbind. Una soluzione alternativa nel nostro caso è rappresentata dalla possibilità di passare agli incapsulatori un puntatore ad un oggetto del tipo base (che è ReflObj) come ulteriore argomento dell'incapsualtore, il quale effettua il downward cast al tipo giusto, conosciuto in quanto gli incapsulatori sono legati alla classe derivata. Nel riquadro [2.4] è mostrato l'incapsulatore, enc_total_salary, per il metodo total_salary della classe employee. Per semplificare la trattazione che segue, si fa uso del costrutto dbind piuttosto che esplicitare l'uso del downward cast all'interno degli incapsulatori.

Ricapitolando la presenza dei tipi unita al binding statico del linguaggio, introduce alcuni ostacoli ad un'agevole introduzione della riflessione in C++, in particolare ostacolano l'uso dei metodi come parametri, la loro attivazione differita e l'attivazione dei metodi propri del tipo dinamico di un oggetto, questi problemi come esposto sono evitabili sfruttando il concetto di incapsulatore che è stato introdotto.

Nota Tutte le supposizioni, essendo solo delle semplificazioni permettono di passare dal nostro codice a del codice effettivamente C++, costruendo un semplice preprocessore.

```
long enc_total_salary( person *p )
{
  employee *e = (employee *) p;
  return( e.total_salary() );
}
```

Codice C++ 2.4: incapsulatore per total_salary

2.2.1.1 Modello a Meta Oggetto

In questo modello le entità fondamentali sono due: il meta-oggetto e l'oggetto controllato dal meta-oggetto, che d'ora in poi è chiamato oggetto riflessivo.
Si vede innanzitutto come viene dichiarata una classe per poter consentire alle sue istanze di essere riflessive.
Tutte le classi a cui appartengono gli oggetti che si intende rendere riflessivi devono avere una struttura che permetta alle loro istanze di rispondere ad alcuni metodi necessari alla computazione riflessiva. In particolare devono possedere il riferimento al meta-oggetto (meta), un metodo per attivare la computazione riflessiva (send) ed uno per ripristinare la computazione al livello base (apply). Il modo più immediato in C++ per ottenere tale caratteristica è quello di rendere la classe in questione una classe derivata da una classe che possiede queste caratteristiche (ReflObj). Questo

modo di procedere è molto simile al modo di procedere che si usa in SmallTalk, soltanto che in tale linguaggio, grazie alle sue peculiarità, le modifiche sono localizzate nella classe di sistema (Object) e non in una nuova classe, costruita appositamente (ReflObj).
Si noti che procedendo in questo modo si deve modificare la classe a cui appartengono gli oggetti che si intende rendere riflessivi. La soluzione è quindi applicabile solo a classi di cui si possiede il sorgente (una soluzione a tale problema è presentata nella sezione [2.5]).
Consideriamo a titolo di esempio la classe person presentata nel riquadro [2.1]. Essa viene modificata, in accordo con quanto affermato, per far si che le sue istanze siano riflessive (vedere il codice C++ riportato nel riquadro [2.5] in cui le cui modifiche sono evidenziate con la doppia sottolineatura).

```
#include <stream.h>

class person : public ReflObj {
  public:
    char *name;
    int age;
   // constructor
    person( char *n, int a ) : name(n), age(a) {};
   // methods
    void print( void ) {cout << "*** " << name << " " << age << "\n";};
    void incage( void ) {++age;};
  };
```

Codice C++ 2.5: classe person

Si deve inoltre definire la classe ReflObj che ha un attributo meta, i metodi send, apply e set_meta, quest'ultimo è necessario per poter cambiare dinamicamente l'associazione tra oggetto e meta-oggetto.
Adesso si esamina nel dettaglio tale entità:

---

meta

Questo attributo si limita a mantenere il riferimento al meta-oggetto e come tale è un puntatore al tipo MetaObj descritto in seguito.

---

set_meta

Questo metodo permette di inizializzare e successivamente cambiare il meta-oggetto che monitorizza l'oggetto che lo attiva; la definizione in C++ del metodo set_meta è:

```
void set_meta( MetaObj *m )
```

Si ricorda che questo metodo deve anche mantenere la consistenza della lista dei referenti dei meta-oggetti come spiegato nella sezione [1.6.1.2].

send

Richiede una computazione riflessiva al meta-oggetto; deve conoscere il metodo da elaborare riflessivamente ed i suoi parametri, come discusso in precedenza, (paragrafo relativo ai metodi di ordine superiore) il metodo può essere passato come puntatore; per il passaggio dei parametri, volendo essere il più possibile generali e non sapendo a priori il numero ed il tipo dei parametri che il metodo ha, si ricorre alle caratteristiche del C, insite nel C++, in particolare alle liste variabili di argomenti.
Quindi la definizione in C++ del metodo send è:

```
void *send( void *(*f)(), int n, ... )
```

Questo metodo controlla se l'oggetto che lo attiva possiede un meta-oggetto ed in tal caso richiede la gestione del metodo riferito da *f al meta-oggetto, altrimenti ne attiva la normale computazione tramite apply; i parametri della send sono necessari anche all'elaborazione degli altri metodi facenti parte del meccanismo riflessivo che si sta presentando (apply e HandleMessage), ed occorre passarglieli all'atto dell'attivazione. Dal momento che non si sa quanti siano i parametri ed è impossibile determinarne il numero staticamente, effettuando, così, la chiamata in modo corretto, per ovviare a tale problema, la stessa send si occupa di costruire un messaggio (oggetto istanza della classe message), contenente il numero di parametri ed i parametri stessi, questi ultimi in un campo della classe message (array di puntatori a *void*, si ricorda che in C++ come in C il tipo *void ** è compatibile con qualsiasi tipo puntatore). È proprio questo messaggio che viene passato come parametro ai metodi attivati da send. Nel riquadro [2.6] è presentata la dichiarazione della classe message: era sufficiente dichiararla come *struct*, ma essendo il C++ un linguaggio ad oggetti si è scelto di farne una classe per essere più in linea con questa metodologia.

```
#include <stdio.h>

class message {
  public:
    void **args;
    int n_args;
   // constructor
    message( int n ) : args(NULL), n_args(n) {};
  }
```

Codice C++ 2.6: classe message

apply

Questo metodo esegue il ritorno della computazione al livello sottostante, dopo che si è effettuata la computazione riflessiva. Deve quindi conoscere il metodo da attivare ed i suoi parametri; il primo viene passato tramite un puntatore come già specificato in precedenza, mentre i rimanenti sono passati tramite un oggetto di tipo message (si usa l'operatore & per evitare al compilatore di farne una copia o all'utente di dover usare un puntatore). Quindi la definizione in C++ del metodo apply è:

```
void *apply( void *(*f)(), message &m )
```

Il metodo deve effettivamente attivare il metodo (f), come richiesto dal programma; per poter far ciò è necessario gestire correttamente la chiamata con il tipo dei vari parametri ed il numero degli stessi. La gestione del numero corretto di parametri è effettuata da un costrutto *switch*[3] che per ogni possibile numero di parametri effettua la chiamata corretta, mentre i tipi adeguati si suppone siano ottenuti tramite un cast implicito dal tipo *void ** al tipo corretto (per maggiori ragguagli sull'effettiva implementazione leggere le note introduttive a questa sezione). Determinare tipo e numero dei parametri non è sufficiente per poter effettivamente attivare il metodo desiderato, in quanto il tipo statico dell'oggetto è ReflObj e non è detto che questa classe sia effettivamente in grado di attivare il metodo richiesto; in realtà si deve usare il tipo dinamico e quindi dbind per usare il metodo e mantenere la genericità. Usare invece il downward cast proposto da Stroustrup, è possibile solo se si conosce il tipo dinamico e questo vuol dire rendere il metodo apply dipendente dalla classe che si vuole rendere riflessiva. Occorrerebbe, allora, renderlo virtuale in ReflObj e ridefinirlo in ognuna di tali classi (ad esempio in person), aumentando così le informazioni da aggiungere alla classe le cui istanze sono riflessive. Con il costrutto dbind è invece possibile scrivere un metodo apply "generico" dentro ReflObj e limitare così le informazioni da far fornire nelle varie classi dal programmatore. Nel riquadro [2.7] è mostrato il codice della classe ReflObj.

La seconda entità necessaria per costruire applicazioni riflessive secondo il modello a meta-oggetto è la classe MetaObj. Questa è la classe base di qualsiasi meta-oggetto, cioè ogni classe, le cui istanze possono essere usate come meta-oggetto di qualche oggetto riflessivo, deve essere una classe derivata dalla classe MetaObj.
Questa, come si può osservare dal codice del metodo send, deve definire il metodo

[3]Per coerenza con quanto sinora affermato, il metodo apply è presentato come uno *switch* sul numero dei parametri, però in realtà l'attivazione del metodo con il numero ed il tipo dei parametri corretto è lasciato all'incapsulatore. L'implementazione effettiva è quindi:

```
void *ReflObj::apply( void *(*f)( ReflObj *, int, void ** ), message &m ) { return( f( this, m.n, m.args ) ); }
```

```
#include ''MetaObj.h''

class ReflObj {
  public:
    MetaObj *meta;
   // constructors
    ReflObj( void ) : meta(NULL) {};
    ReflObj( MetaObj *m );
   // methods
    void set_meta( MetaObj *m )
    void *send( void *(*f)(), char *fmt, ... );
    void *apply( void *(*f)(), message &m );
  };

ReflObj( MetaObj *m ) : meta(m) { m→add_ref( &this ) };


void ReflObj::set_meta( MetaObj *m ) {
  if (meta)
    meta→remove_ref(&this);
  meta = m;
  meta→add_ref(&this)
}

void *ReflObj::send( void *(*f)(), int n, ... ) {
  message m( n );
  va_start( va_list va, n );
  m.args = new void*[n];
  for( int i=0; i<n; i++ ) m.args[i] = va_arg( va, void * );
  va_end( va );
  if (meta) return( meta→send( meta→HandleMessage, 3, this, f, m ) );
  return( apply( f, m ) );
}

void *ReflObj::apply( void *(*f)(), message &m ) {
  dbind( this );
  switch( m.n_args ) {
    case 0: return( f() );
    case 1: return( f( m.args[0] );
    case 2: return( f( m.args[0], m.args[1] ) );
        ⋮
  }
}
```

Codice C++ 2.7: classe ReflObj

HandleMessage. Inoltre, siccome un meta-oggetto può controllare più oggetti riflessivi, deve mantenere i riferimenti a tali oggetti nell'attributo referents, e può avere a sua volta un meta-oggetto che lo controlla. La classe MetaObj può, quindi, essere a sua volta riflessiva, cioè deve essere una diretta discendente della classe ReflObj. Nel seguito si esamina nel dettaglio tale classe ed in particolare il suo unico metodo.

```
class MetaObj : public ReflObj {
  private:
    ReflObj **referents; void *free; int n;
  public:
   // constructor
    MetaObj( void ) : free(NULL), n(0), referents(NULL) {};
   // methods
    void *HandleMessage( ReflObj *ro, void *(*f)(), message &m );
    void add_ref( ReflObj *ro );
    void remove_ref( ReflObj *ro );
  };


void MetaObj::remove_ref( ReflObj *ro ) {
  for( int i=0;i<n;i++ )
    if ( ro == *(referents+i) )
      break;
  *(referents+i) = free;
  free = - (*(referents+i));
}

void MetaObj::add_ref( ReflObj *ro ) {
  if ( referents ) {
    referents = (ReflObj **)calloc( ++n, sizeof(ReflObj *) );
    *(referents+0) = ro;
  }
  if ( free ) {
    void *aux = -free;
    free = *aux;
    *aux = ro;
  } else {
    referents = (ReflObj **)realloc( sizeof(ReflObj *)*(++n) );
    *(referents+(n-1)) = ro;
  }
}

void *MetaObj::HandleMessage( ReflObj *ro, void *(*f)(), message &m ) {
  return( ro→apply( f, m ) );
}
```

Codice C++ 2.8: classe MetaObj

---

referents

Si mantiene il riferimento agli oggetti riflessivi controllati dal meta-oggetto in una struttura (**referents**) gestita tramite due metodi: il primo, **add_ref**, inserisce i referenti all'interno della struttura; il secondo, **remove_ref**, elimina un referente dalla struttura. La struttura è implementata tramite un array dinamico di puntatori ad oggetti riflessivi. A questa struttura è affiancata una variabile (**free**) contenente il riferimento ad un elemento disponibile dell'array.
Per evitare costosi ricompattamenti dell'array, i referenti cancellati non vengono

effettivamente eliminati dalla struttura, ma solo resi disponibili per successivi inserimenti, attraverso una gestione a "lista libera".

HandleMessage

Questo metodo è responsabile della computazione riflessiva, e come tale è dipendente dal tipo di meta-oggetto. Questo significa che ogni classe discendente da MetaObj deve implementare la propria HandleMessage; per questo motivo si sarebbe potuto definire HandleMessage come virtuale, però siccome si vuole che abbia un comportamento di default (quello di tornare al livello sottostante), si è scelto di non rendere MetaObj una classe astratta. Per poter effettuare il ritorno in modo corretto la HandleMessage deve conoscere l'oggetto riflessivo, il metodo da attivare ed i suoi parametri; l'oggetto riflessivo, ovviamente, è un puntatore ad un oggetto di tipo ReflObj mentre metodo e parametri vengono passati, come già spiegato per il metodo apply, tramite un puntatore ed un oggetto di tipo message, la definizione in C++ del metodo HandleMessage è quindi

```
void *HandleMessage( ReflObj *ro, void *(*f)(), message &m )
```

Nel riquadro [2.8] è riportato il codice della classe MetaObj.

2.2.1.2 Modello a Materializzazione del Canale

Questo modello, come già sottolineato nel capitolo [1], non utilizza alcun meta-oggetto, bensì la nuova entità canale[4]. L'oggetto riflessivo, come già nel caso del modello a meta-oggetto, oltre a poter effettuare le proprie attività, può richiedere l'esecuzione dell'attività riflessiva (passaggio al meta-livello) ed il ripristino della computazione al livello sottostante (metodi send e apply); come già detto il modo più immediato per ottenere tale comportamento, nel caso generale, è quello di modificare la classe che si vuole rendere riflessiva facendola ereditare da una classe, astratta o meno, che abbia tali caratteristiche (ReflObj).
La prima peculiarità saliente da poter osservare nella classe ReflObj, è l'assenza dell'attributo meta e quindi la mancanza di un riferimento all'entità riflessiva, con un'ovvia linearizzazione della definizione della classe riflessiva (non c'è più il riferimento alla classe MetaObj, la quale ereditava dalla classe ReflObj). Si attende di descrivere i metodi della classe ReflObj dopo aver descritto i canali e la loro implementazione.
I canali, come già detto nella sezione [1.8], hanno la caratteristica di sopravvivere alla comunicazione e di essere riutilizzati ogniqualvolta sia attivato un metodo che necessita di una computazione riflessiva dello stesso genere e che coinvolge gli stessi

[4] Per semplicità non si prendono in considerazione modelli misti.

oggetti. Quindi tali canali sono distinguibili per gli oggetti che coinvolgono ed il genere di computazione riflessiva che effettuano.
Occorre, quindi, introdurre un'entità in grado di mantenere le informazioni necessarie per il recupero, ad esempio un oggetto istanza della classe dictionary (OpenChannel).
I canali vengono classificati in una gerarchia di classi (in base al genere), che ha come radice la classe channel, ed ognuno di essi differisce per il comportamento riflessivo delle sue istanze. Per ottenere le prime due caratteristiche occorre mantenere le informazioni in modo che qualsiasi tipo di canale sia in grado di reperirle. Il modo migliore è quello di avere una variabile condivisa da ogni possibile classe della gerarchia dei canali (è quindi inserita in channel e definita static, questo in C++ permette di condividere tale variabile tra le istanze della classe), che mantenga tali informazioni ed i metodi per reperirle. Per questa trattazione non è particolarmente rilevante l'organizzazione di tale struttura (lista, dizionario, hash-table, ...), quindi ci si limita a fornirne la dichiarazione (riquadro [2.9]). Nella sezione [2.6] si descrive un modo efficiente per implementare tale struttura. Ovviamente, per attenersi alla metodologia object-oriented, questa struttura è una classe (dictionary) che prevede i metodi necessari per l'inserimento (insert), la cancellazione (delete) e la ricerca|recupero (get).

```
class dictionary {
  private:
    ...
  public:
   // methods
    void get( ReflObj &sender, &receiver, char *tag_type, channel **ch );
    void insert( ReflObj &sender, &receiver, char *tag_type, channel *ch );
    void delete( channel *ch );
  }
```

Codice C++ 2.9: classe dictionary (incompleta)

I metodi necessari al canale sono uno per la creazione del canale (il costruttore channel) ed un altro per la gestione della computazione riflessiva ed il ritorno al livello sottostante (HandleMessage). Ora si esamina nel dettaglio tali metodi.

---

channel

Esso gestisce la creazione del canale e mantiene le relative informazioni (gestione dell'attributo OpenChannel). Il costruttore deve conoscere gli oggetti che il canale

deve collegare, il genere di computazione riflessiva richiesta[5] ed il metodo da eseguire una volta effettuata la computazione riflessiva, con i relativi parametri. Quest'ultimo gruppo di informazioni viene fornito sotto forma di un'istanza della classe message, pressoché uguale a quella presentata nel riquadro [2.6], con l'aggiunta di un campo contenente il metodo da eseguire una volta tornati al livello sottostante, vedere riquadro [2.10].
La definizione in C++ dei costruttori è quindi[6]:

```
channel( ReflObj &s, &r, char *tt, message &m )
```

```
channel( ReflObj &s, &r, char *tt )
```

Quando il costruttore viene attivato tenta di recuperare un canale già esistente ed adatto alla computazione richiesta, in caso tale canale non esista, ne crea effettivamente uno nuovo e lo inserisce nella struttura.

```
#include <stdio.h>

class message {
  public:
    void *args;
    int n_args;
    void *(*selector)();
   // constructor
    message( int n, void *(*s)() ) : args(NULL), selector(s), n_args(n) {};
  }
```

Codice C++ 2.10: classe message

HandleMessage

Oltre all'operazione di creazione|recupero, il canale deve essere in grado di effettuare la computazione riflessiva, di questo si occupa il metodo HandleMessage, che differisce in ogni classe discendente da channel, e nel caso di quest'ultima si limita ad effettuare

[5] Essendo ogni genere di computazione riflessiva rappresentata da un canale di tipo diverso, questa informazione può considerarsi implicita nel costruttore dei canali. Nel caso del C++ si sceglie di esplicitarla.

[6] Viene usato il costruttore o meglio una coppia di costruttori, in quanto sono gli unici metodi della classe (confrontare con i metodi di classe di SmallTalk), che possono essere attivati indipendentemente dall'esistenza di un'istanza della classe e che vengono richiamati automaticamente quando si crea una nuova istanza della classe, sia che avvenga all'atto della definizione di una variabile (channel ch(s, r, tt, m)) che tramite un'operazione di *new* (ch = new(s, r, tt, m)). Se ne usano due, che differiscono nella lista dei parametri per la presenza o meno del messaggio, uno effettua il tentativo di recupero, l'altro invece crea effettivamente il canale.

il ritorno al livello sottostante attivando il metodo appropriato.
Il metodo HandleMessage deve conoscere l'oggetto riflessivo ed il messaggio; siccome tali informazioni sono insite nella struttura del canale, il metodo HandleMessage non ha parametri, quindi la sua definizione in C++ è:

```
void *HandleMessage( void )
```

Quest'ultimo si limita ad effettuare il ritorno al livello sottostante (comportamento per difetto). Nel riquadro [2.11] è riportato il codice della classe channel.

```
#include "ReflObj.h"

class channel {
  private:
    static dictionary OpenChannel;
  public:
   reflObj sender, receiver;
   char *tag_type;
   message msg;
   // constructors
    channel( ReflObj &s, &r, char *tt, message &m );
    channel( ReflObj &s, &r, char *tt ) : sender(s), receiver(r), tag_type(tt) {};
   // method
    void *HandleMessage( void ) { return( receiver.apply(msg) ); };
  };

channel::channel( ReflObj &s, &r, char *tt, message &m )
{
  channel ch = OpenChannel.get( s,r, tt );
  if (ch)
    {
      ch = new( s, r, tt );
      OpenChannel.insert( s, r, tt, ch );
    }
  msg = m;
  return ch;
}
```

Codice C++ 2.11: classe channel

Una volta vista l'implementazione dei canali, si può trattare l'implementazione degli oggetti riflessivi (ReflObj), questi, come già detto, devono essere in grado di rispondere al metodo send ed al metodo apply; vediamoli nel dettaglio.

send

```
class ReflObj {
  public:
   //constructor
    ReflObj(){};
   //methods
    void *send( ReflObj &r, char *tt, channel (*c)(), void *(*f)(), int n, ... );
    void *apply( message &m );
  };

void *ReflObj::send( ReflObj &r, char *tt, channel (*c)(), void *(*f)(), int n, ... )
{
  message m(n,f);
  va_start( va_list va, n );
  m.args = new void*[n];
  for( int i=0;i<n;i++ ) m.args[i] = va_args( va, (void *));
  va_end( va );
  channel ch = (*c)( this, r, tt, m );
  return(ch.HandleMessage());
}

void *ReflObj::apply( message &m );
{
  dbind( this );
  switch( m.n_args ) {
    case 0 : return( m.(*selector)() );7
    case 1 : return( m.(*selector)( m.args[0] );
    case 2 : return( m.(*selector)( m.args[0], m.args[1] ) );
       ⋮
}
```

Codice C++ 2.12: classe ReflObj

Gestisce l'attivazione della computazione riflessiva. Dovendo conoscere il genere di computazione riflessiva che deve essere effettuata, **send** necessita di un parametro che definisca il genere della computazione riflessiva richiesta, una possibilità è rappresentata dal costruttore del canale desiderato. Inoltre occorre passare come parametro informazioni come il sender (informazione implicita, in quanto il sender è l'oggetto che attiva il metodo e quindi è rappresentato dalla keyword ***this***), il receiver, il metodo da eseguire ed i suoi parametri, quest'ultimi, come nel caso modello a meta-oggetto, vengono passati sfruttando le liste variabili di argomenti. La sua definizione in C++ è:

```
void *send( ReflObj &r, char *tt, channel (*ch)(), void *(*f)(), int n, ... )
```

[7] In realtà m.(*selector)() sta per m.(*selector)(this) dove selector è l'incapsulatore del metodo in questione.

La send effettua il passaggio al meta-livello dopo aver costruito il messaggio contenente il metodo da elaborare riflessivamente ed i suoi parametri, a questo punto crea|riutilizza un canale e comincia effettivamente la computazione riflessiva.

---

apply

Gestisce il ritorno della computazione al livello sottostante dopo un'esecuzione riflessiva; dal momento che deve attivare il metodo invocato dall'oggetto riflessivo, deve conoscere quest'ultimo ed i parametri da passargli, che sono contenuti nel messaggio costruito da send; la sua definizione in C++ è:

```
void *apply( message &m )
```

Come già osservato nel modello a meta-oggetto, per attivare il metodo desiderato, si deve usare il binding dinamico e determinare il numero ed il tipo corretto dei parametri da passargli. Nel riquadro [2.12] è mostrato il codice della classe ReflObj.

Nota Nelle porzioni di codice presentato in questo paragrafo si sono adottate le ipotesi fatte all'inizio della sezione.

Nota Se si volessero effettuare delle meta-meta computazioni occorre rendere il canale un oggetto riflessivo, e quindi farlo ereditare dalla classe ReflObj; per maggiori dettagli su come effettuare questo tipo di computazione, vedere la sezione [1.8].

2.2.2 Riflessioni in Oberon

Oberon non è un linguaggio ampiamente noto, come il C++ tuttavia in questa sezione, a titolo di paragone, si esaminano i passi necessari per implementare i due modelli riflessivi considerati in questo linguaggio.
Per agevolare la comprensione gli esempi presentati vengono annotati, comunque per maggiori dettagli si può consultare l'appendice [B].

Oberon nonostante non presenti dei meccanismi per associare le operazioni ai tipi e per ereditare tali operazioni nei sotto-tipi (come tale non è classificabile come object-oriented, almeno secondo la classificazione dettata da Wegner [Wegner 87]), può, comunque, essere usato per programmare secondo il paradigma object-oriented sfruttando l'estensione dei tipi e le variabili di tipo procedura. Quindi in Oberon la programmazione ad oggetti è ottenuta come segue:

- Un oggetto è rappresentato da un puntatore ad un descrittore di record che, oltre le informazioni di stato, contiene dei campi di tipo procedura, i cui valori definiscono le operazioni che possono essere applicate all'oggetto. Ad esempio nel riquadro [2.13] è mostrato il codice Oberon per un tipo Person che ha come attributi i campi Age e Name e due variabili di tipo procedura (Print e IncAge), che d'ora in poi vengono riferite, impropriamente, con il termine metodi.

```
TYPE Person* = POINTER TO PersonDesc;[8]
        PrintType* = PROCEDURE( p : Person );[9]
        IncAgeType* = PROCEDURE( p : Person );
        PersonDesc* =
          RECORD
            Name* : ARRAY 25 OF CHAR;
            Age* : INTEGER;
            IncAge* : IncAgeType;
            Print : PrintType;
          END;

PROCEDURE IncAge*( p : Person );
BEGIN
  INC( p.Age );
END IncAge;

PROCEDURE PrintPerson*( p : Person );
BEGIN
  MYTEXTS.WriteString( "*** " ); MYTEXTS.WriteString( p.Name );
  MYTEXTS.WriteString( " " ); MYTEXTS.WriteInt( p.Age );
  MYTEXTS.WriteString( " ***" ); MYTEXTS.WriteLn;
END PrintPerson;
```

Codice Oberon 2.13: Classe Person

- L'ereditarietà è ottenuta estendendo il tipo base dell'oggetto; ad esempio se si vuole derivare dal tipo Person il tipo Employee con in più il campo Salary ed un metodo per il calcolo dello stipendio annuale (Total_Salary), si deve rendere quest'ultimo un estensione del tipo Person come si può vedere nel riquadro [2.14].

- l'overriding è ottenuto assegnando ai campi di tipo procedura una procedura differente, mantenendo comunque il tipo corretto; questo provoca dei problemi nel recuperare i campi propri del tipo esteso, è quindi necessaria una guardia dinamica sui tipi per concedere l'accesso a variabili di stato definite nell'estensione del tipo base. Nell'esempio, riquadro [2.15], è mostrato il codice

[8] Normalmente le entità definite all'interno di un modulo sono private al modulo stesso e non possono essere utilizzate dai moduli che lo importano, per renderle esportabili è necessario marcarle con un asterisco.

[9] Una limitazione della programmazione object-oriented in Oberon è dovuta alla necessità di inserire un parametro nei metodi rappresentante il self dell'oggetto. Questo problema è stato risolto nella successiva evoluzione del linguaggio Oberon-2 [Mössenböck 91] ed in una estensione di Oberon chiamata EOS, definita in [Ancona-Moreno 94], in cui sono state introdotte le procedure legate al tipo (***Type-Bound procedure***) vedi appendice [B].

```
TYPE Employee* = POINTER TO EmployeeDesc;
     TotSalaryType* = PROCEDURE( e : Employee ) : INTEGER;
     EmployeeDesc* =
       RECORD
         Salary* : INTEGER;
         Total_Salary : TotSalaryType;
       END;

PROCEDURE Total_Salary*( e : Employee ) : INTEGER;
BEGIN
  RETURN( 12*e.Total_Salary );
END Total_Salary;
```

Codice Oberon 2.14: classe Employee

dell'operazione di stampa relativa al tipo Employee che contempla, rispetto a quella per il suo tipo base (Person), anche la visualizzazione dello stipendio.

```
PROCEDURE PrintEmployee( p : Person )[10]
BEGIN
  WITH p : Employee DO
    MYTEXTS.WriteString( "*** " ); MYTEXTS.WriteString( p.Name );
    MYTEXTS.WriteString( " " ); MYTEXTS.WriteInt( p.Age );
    MYTEXTS.WriteString( " *** " ); MYTEXTS.WriteInt( p.Salary );[11]
    MYTEXTS.WriteLn;
  END (* WITH *);
END PrintPerson;
```

Codice Oberon 2.15: metodo PrintEmployee

- L'inizializzazione degli elementi di un oggetto, nei linguaggi object-oriented, viene fatta tramite un metodo speciale detto costruttore, facente anch'esso parte dell'oggetto. In Oberon non è stata prevista la possibilità di avere un costruttore, e non è neanche possibile, in un programma, aggiungere un metodo con questa funzione perché, essendo un oggetto un puntatore ad un record, occorre allocargli dello spazio prima di poter accedere ai suoi metodi, ed essendo questi delle variabili, occorre inizializzarle prima di poterle usare. Risulta evidente che se il costruttore fosse uno di questi metodi, non potrebbe essere attivato per poter costruire l'oggetto di cui farebbe parte. Il costruttore deve quindi essere una procedura esterna progettata dal programmatore che si

[10]La procedura PrintEmployee deve mantenere il tipo PrintType per poter essere assegnata al campo Print ereditato dal tipo Person e quindi ha un parametro p di tipo Person e non di tipo Employee.

[11]Senza il costrutto *WITH* questa attivazione di metodo avrebbe dato errore perché Salary non appartiene al tipo statico dell'oggetto, ma al tipo dinamico dello stesso.

deve occupare di inizializzare tutti i campi compresi quelli di tipo procedura. Nell'esempio (riquadro [2.16]) sono mostrati i costruttori per i tipi Person e Employee. Come si può osservare, questi costruttori si limitano ad inizializzare i campi degli oggetti senza riservargli dello spazio, in pratica senza effettivamente costruirli (manca la chiamata all'istruzione *NEW*). Il motivo di ciò è che Oberon è un linguaggio con binding tipicamente statico e quindi il far creare l'oggetto a InitPerson impediva la possibilità di renderlo un oggetto di tipo Employee, farlo creare sia a InitPerson che a InitEmployee vorrebbe dire rinunciare all'impiego di InitPerson dentro InitEmployee, pena la doppia creazione dell'oggetto, e quindi con una scarsa riusabilità del codice.

```
PROCEDURE InitPerson*( p : Person; name : ARRAY OF CHAR; age : INTEGER );
BEGIN
  STRING.Copy( name, p.Name );
  p.Age := age;
  p.IncAge := IncAge;
  p.Print := PrintPerson;
END InitPerson;

PROCEDURE InitEmployee*( e : Employee; name : ARRAY OF CHAR; age : INTEGER );
BEGIN
  InitPerson( e, name, age );
  e.Salary := salary;
  e.Total_Salary := Total_Salary;
END InitEmployee;
```

Codice Oberon 2.16: Costruttori per le classi Person ed Employee

Nel caso del C++ si sono dovute fare molte ipotesi sulla presenza o meno di meccanismi per gestire le informazioni sui tipi e sulle possibilità di poter passare o meno delle funzioni come parametri. Nel caso di Oberon queste supposizioni non sono necessarie perché tali caratteristiche sono già intrinseche nel linguaggio (il costrutto *WITH* già visto negli esempi ed i metodi del prim'ordine); quindi una volta stabilito come possono essere utilizzate le metodologie ad oggetti resta solo da chiarire come possono essere passati un numero variabile di parametri ad una funzione.
Il passaggio di un numero variabile di parametri è fatto sfruttando il concetto di messaggio e gestore dei messaggi fornito da Wirth in [Reiser-Wirth 92].
Esso consiste nel definire un tipo Message come un record vuoto e successivamente definire suoi sotto-tipi che estendono quest'ultimo con i campi contenenti gli argomenti della funzione da gestire. Occorre poi dotare la classe che li può ricevere di un metodo in grado di interpretarli detto handler, che utilizza il messaggio ed il relativo tipo per attivare il metodo corretto.
Questo meccanismo presenta i seguenti vantaggi:

- Il messaggio può essere memorizzato in una variabile e riutilizzato per invii successivi.

- È possibile inviare ad un oggetto un messaggio che esso non è in grado di "capire". L'oggetto allora può ignorare il messaggio o delegare il trattamento ad un altro oggetto, questo comportamento è essenziale per effettuare il passaggio dal livello base al meta-livello. Con le variabili di tipo procedura questo non è possibile perchè il compilatore controlla se il messaggio può essere capito dall'oggetto o meno, segnalando eventualmente l'errore e quindi arrestando la compilazione.
- l'handler può essere rimpiazzato a run-time cambiando in un'unica soluzione il comportamento dell'oggetto; questo perchè i metodi non sono altro che delle variabili di tipo procedura e quindi modificabili con delle assegnazioni.
- I messaggi possono essere dichiarati in moduli diversi, in modo da poter aggiungere nuovi messaggi ad un tipo, quando un nuovo modulo è codificato.
- Lo stesso messaggio può essere facilmente distribuito a più oggetti (broadcasting dei messaggi). Consideriamo il caso in cui si abbia una lista di persone e dipendenti e si voglia stampare la scheda di ognuno. Usando le procedure, il richiedente avrebbe dovuto attraversare la lista delle persone ed inviare il messaggio print ad ogni persona.

  ```
  p := firstPerson; WHILE p # NIL DO p.print(p); p := p.next END
  ```

  La struttura della lista di persone deve, quindi, essere conosciuta al richiedente, cosa non sempre possibile. Il codice per l'attraversamento della lista viene duplicato ogni volta che si vuole eseguire un'azione analoga, ma che coinvolge un metodo diverso. Con gli handler si può implementare l'attraversamento in una procedura (BroadCast) alla quale viene passato come parametro il messaggio che deve essere trasmesso:

  ```
  PROCEDURE (l:List)BroadCast( VAR m : Message )
    VAR p : Person;
  BEGIN
    p := l.first;
      WHILE p # NIL DO
        p.handle(p,m);
        p := p.next
     END;
  END BroadCast;
  ```

  Questo consente di nascondere la struttura della lista e tenere il codice necessario per l'attraversamento solo in un posto.

Il metodo presenta anche degli svantaggi:

- Non è immediatamente chiaro quali messaggi appartengono ad un tipo, cioè quali messaggi un oggetto "capisce". Per determinarli occorre sapere quale handler viene attivato a run-time e come questo viene implementato.

- Gli errori di tipo sintattico relativi all'attivazione di un metodo non possono più essere rilevati in fase di compilazione, ma solo in fase di esecuzione.
- I messaggi sono interpretati dall'handler a run-time ed in ordine sequenziale. Questo è più lento di qualsiasi meccanismo di binding dinamico.

Nonostante abbia dei difetti, uno dei quali ci porta a preferirgli il linguaggio C++, questo è il meccanismo migliore offerto da Oberon per le nostre necessità. Nel riquadro [2.17] è mostrato il codice dell'handler, per la classe Person descritta prima, che gestisce i metodi Print e IncAge.

```
TYPE Message = RECORD END;
       PrintMessage = RECORD(Message) p : Person END;
       IncAgeMessage = RECORD(Message) p : Person END;

PROCEDURE Handler( p : Person; m : Message );
BEGIN
  IF ( m IS PrintMessage ) THEN
    WITH m : PrintMessage DO
      m.p.Print( m.p );
    END;
  ELSIF m IS IncAge THEN
    WITH m : IncAge THEN
      m.p.IncAge( m.p );
    END;
  ELSE
    MYTEXTS.WriteString( 'Messaggio di tipo sconosciuto !' );
  END;
END Handler;
```

Codice Oberon 2.17: Handler e messaggi

2.2.2.1 Modello a Meta Oggetto

In questo modello le entità fondamentali sono due: il meta-oggetto e l'oggetto riflessivo; tutte le classi che si intende rendere riflessive devono avere una struttura che permetta alle loro istanze di rispondere ad alcuni metodi, necessari alla computazione riflessiva.

Innanzitutto serve il riferimento al meta-oggetto rappresentato dallo slot meta, un metodo per attivare la computazione riflessiva (send) ed uno per ripristinare la computazione al livello base (apply); come già visto usando il C++ il modo più immediato per dotare una classe di queste caratteristiche è quello di farla ereditare da una classe che le possiede. Quindi si costruisce la classe[13] ReflObj dotata della variabile meta e

[12] Equivalente in C++ a meta→send(meta→HandleMessage, 3, this, f, m)

[13] Qui, come nel seguito, si usano impropriamente i termini classe, oggetto, metodo ed ereditarietà in luogo di record, puntatore a record, procedura ed estensione del tipo, per rendere l'esposizione più aderente alla metodologia ad oggetti.

```
MODULE REFLOBJ
IMPORT METAOBJ;
TYPE ReflObj* = POINTER TO ReflObjDesc;
        Message* = POINTER TO MessageDesc;
        MessageDesc* = RECORD sender : ReflObj; END;
        SendType* = PROCEDURE( ro : ReflObj; m : Message );
        ApplyType* = PROCEDURE( ro : ReflObj; m : Message );
        SetMetaType* = PROCEDURE( ro : ReflObj; mo : METAOBJ.MetaObj );
        ReflObjDesc* =
          RECORD
            meta* : MetaObj.MetaObj;
            send* : SendType;
            apply* : ApplyType;
            set_meta* : SetMetaType;
          END;

PROCEDURE set_meta*( ro : ReflObj; mo : METAOBJ.MetaObj )
BEGIN
   IF ( ro.meta # NIL ) THEN ro.meta.remove_ref( ro.meta, ro ); END;
   IF ( mo.add_ref( mo, ro ) ) THEN ro.meta := mo;
   ELSE ro.meta := NIL;
   END;
END set_meta;

PROCEDURE send*( ro : ReflObj; m : Message )
VAR mm : METAOBJ.MetaHandleMessage;
    mo : ReflObj;
BEGIN
   IF ( ro.meta # NIL ) THEN
      NEW( mm );
      mm.sender := ro.meta;
      mo := ro.meta;
      mo.send( mo, mm );12
   ELSE ro.apply( ro, m );
   END;
END send;

PROCEDURE apply*( ro : ReflObj; m : Message )
BEGIN
   IF ( m IS METAOBJ.MetaHandleMessage ) THEN
      WITH m : METAOBJ.MetaHandleMessage DO
         ro.HandleMessage( ro, m.m );        (* ro è un meta-oggetto *)
      END;
   ELSE
     :
     :                    (* cambia con il cambiare degli oggetti riflessivi *)
   END;
END apply;

PROCEDURE InitReflObj*( VAR ro : ReflObj );
BEGIN
   ro.send := send;
   ro.meta := NIL;
   ro.apply := apply;
   ro.set_meta := set_meta;
END InitReflObj;


END ReflObj.
```

Codice Oberon 2.18: Classe ReflObj

dei metodi send e apply, da cui eredita qualsiasi classe che si vuole rendere riflessiva. Esaminiamo il dettaglio dell'implementazione Oberon di tale classe.

meta e set_meta

meta è semplicemente una variabile che mantiene il riferimento all'entità riflessiva e come tale è un oggetto di tipo MetaObj, mentre set_meta è il metodo che durante l'esecuzione permette di cambiare il meta-oggetto; quest'ultimo ha la seguente definizione in Oberon:

```
PROCEDURE set_meta( ro : ReflObj; mo : MetaObj )
```

È compito di quest'ultimo metodo mantenere consistente la lista dei referenti dei meta-oggetti. Come si vede in seguito, per ragioni legate al linguaggio e per semplificare l'implementazione non è detto che questo metodo abbia sempre successo, in caso di fallimento meta è posto a NIL.

send

Questo metodo si occupa di richiedere al meta-oggetto una computazione riflessiva; deve conoscere, oltre l'oggetto stesso, per le limitazioni di Oberon citate prima, il metodo da elaborare riflessivamente ed i suoi parametri. Tali conoscenze sono ottenute sfruttando l'espediente suggerito da Wirth (vedere inizio sezione per maggiori ragguagli), per cui si passa al metodo send un oggetto derivato da Message contenente i parametri del metodo ed in base al tipo dinamico di quest'ultimo si è in grado di risalire al metodo in questione. Quindi la definizione in Oberon del metodo send è:

```
PROCEDURE send( ro : ReflObj; m : Message )
```

Questo metodo verifica la presenza di un meta-oggetto che monitorizza l'oggetto che lo attiva ed in tal caso passa a tale meta-oggetto la gestione del metodo in questione, altrimenti passa il controllo al metodo apply per la sua esecuzione.

apply

Questo metodo sovraintende il ritorno della computazione al livello sottostante, dopo che si è effettuata la computazione riflessiva; deve quindi conoscere, oltre l'oggetto riflessivo, il metodo da attivare ed i suoi parametri. Questi vengono passati come nel caso del metodo send tramite un oggetto di tipo Message. Quindi la sua definizione in Oberon è:

```
PROCEDURE apply( ro : ReflObj; m : Message )
```

Esso deve ricavare dal tipo dinamico del messaggio il metodo ed attivarlo effettivamente, come richiesto dal programma.

Nota Sempre tramite il messaggio si possono gestire i valori di ritorno dei metodi, prevedendo un campo aggiuntivo del messaggio in cui inserirli durante l'esecuzione del metodo apply e passando il messaggio per riferimento. I parametri per riferimento sono ottenuti sfruttando il side-effect offerto dai puntatori.

Occorre notare che in Oberon si deve definire una funzione apply per ogni classe riflessiva differente, in quanto ognuna riconosce metodi differenti propri della classe; questo è dovuto alla gestione dei tipi statici del linguaggio ed alla mancanza di un costrutto per la gestione di un numero variabile di parametri. C'è un modo per aggirare parzialmente il problema ed è quello di usare gli *ARRAY OF BYTE*, questi sono equivalenti ai *void ** del C++. Abbinandoli al messaggio introdotto da Wirth si possono classificare i metodi in base al numero dei parametri, avere tipi procedura "generici" con parametri di tipo *ARRAY OF BYTE* e scrivere apply, similmente al caso del C++, come un *case* sul numero dei parametri. L'uso degli *ARRAY OF BYTE* però fa perdere la possibilità di utilizzare il type checking fornito da Oberon. Per evitare di appesantire la trattazione e siccome gli *ARRAY OF BYTE* non sono gestiti nello stesso modo da tutti i compilatori si considera apply come una procedura variante da classe a classe. Nel riquadro [2.18] è mostrato il codice della classe ReflObj.

Nota Sia in C++ che in Oberon si è affrontato il problema del type-checking statico sull'attivazione dei metodi, sfruttando delle soluzioni che permettono di aggirare l'ostacolo (ad esempio i *void ** in C++) senza effettivamente risolverlo. Fermo restando che l'intento di questo capitolo è quello di introdurre le riflessioni in linguaggi fortemente tipati senza modificare il linguaggio stesso, questo tipo di soluzione è una delle più adatte. Tale soluzione presenta, comunque, il problema di rendere non safe il type checking effettuato in compilazione, introducendo la possibilità che si verifichino errori sui tipi a run-time.
Una soluzione migliore, che però implica l'estensione dei linguaggi adottati, concerne l'estensione oculata dei meccanismi di RTTI, in modo da poter effettuare dei controlli anche su queste situazioni.

La seconda entità necessaria per costruire applicazioni riflessive secondo il modello a meta-oggetto è la classe MetaObj. Questa è la classe base di qualsiasi meta-oggetto, cioè ogni classe, le cui istanze possono essere usate come meta-oggetto di qualche oggetto riflessivo, deve essere una classe derivata dalla classe MetaObj.
La classe MetaObj ha il metodo HandleMessage necessario per gestire la computazione riflessiva, e siccome un meta-oggetto può controllare più oggetti riflessivi, tiene i riferimenti a tali oggetti; inoltre il meta-oggetto può avere a sua volta un meta-

oggetto che lo controlla e quindi essere a sua volta riflessivo, questo significa che la classe MetaObj deve essere una diretta discendente della classe ReflObj. Vediamo più dettagliatamente tale classe ed in particolare il suo unico metodo.

referents

La gestione degli oggetti riferiti da un meta-oggetto è effettuata in maniera simile al caso del C++, la principale differenza è rappresentata dall'impossibilità di avere array di lunghezza variabile in Oberon.[14] Si hanno altre due possibilità: fissare il numero massimo di referenti oppure definire la struttura referents come una lista.
Si è optato sulla prima soluzione innanzitutto per la semplicità della struttura ed anche perché per le nostre necessità il numero dei referenti di un meta-oggetto è piuttosto esiguo. Quindi si hanno due metodi: add_ref che inserisce i referenti nella struttura, remove_ref che elimina un referente dalla struttura.

HandleMessage

Questo metodo tratta la computazione riflessiva, che è dipendente dal tipo di meta-oggetto; questo significa che ogni classe discendente da MetaObj deve implementare la propria HandleMessage, il cui codice per il comportamento (default) di tornare al livello sottostante, è mostrato nel riquadro [2.19].
Per poter effettuare il ritorno in modo corretto la HandleMessage deve conoscere l'oggetto riflessivo, il metodo da attivare ed i suoi parametri; tali informazioni sono contenute nel messaggio come già spiegato per la send e l'apply. La sua definizione in Oberon è quindi:

```
PROCEDURE HandleMessage( ro : ReflObj; m : Message )
```

Nel riquadro [2.19] è riportato il codice della classe MetaObj.

2.2.2.2 Modello a Materializzazione del Canale

Anche per il linguaggio Oberon valgono le osservazioni di carattere generale fatte per il modello a materializzazione del canale nel caso del C++, cioè non esistendo più un meta-oggetto, che è sostituito dalla materializzazione della comunicazione in un canale, con un'ovvia semplificazione della definizione della classe degli oggetti

[14] Oberon-2 fornisce la possibilità di avere variabili di tipo array, dimensionabili tramite un'operazione di inizializzazione (*NEW*) a run-time. Però l'uso degli array aperti per gestire un numero variabile di referenti non è così essenziale da giustificare il passaggio da Oberon a Oberon-2.

[15] Con Oberon si è costretti a limitare il numero dei referenti di un meta-oggetto.

```
MODULE METAOBJ
IMPORT REFLOBJ;
TYPE MetaObj* = POINTER TO MetaObjDesc;
        HandleMessageType* = PROCEDURE( mo : MetaObj; m : REFLOBJ.Message );
        AddRefType* = PROCEDURE( mo : MetaObj; ro : REFLOBJ.ReflObj ) : BOOLEAN;
        RemoveRefType* = PROCEDURE( mo : MetaObj; ro : REFLOBJ.ReflObj );
        MetaObjDesc* =
          RECORD( REFLOBJ.ReflObjDesc )
            HandleMessage* : HandleMessageType;
            referents* : ARRAY 25 OF REFLOBJ.ReflObj;[15]
            add_ref : AddRefType;
            remove_ref : RemoveRefType;
          END;

PROCEDURE HandleMessage*( mo : MetaObj; m : REFLOBJ.Message )
BEGIN
  m.sender.apply( m.sender, m );
END HandleMessage;

PROCEDURE add_ref*( mo : MetaObj; ro : REFLOBJ.ReflObj ) : BOOLEAN
VAR aux : REFLOBJ.ReflObj;
BEGIN
  IF ( mo.referents[0] # NIL ) THEN
    aux := -mo.referents[0];
    mo.referents[0] := mo.referents[aux];
    mo.referents[aux] := ro;
    RETURN TRUE;
  END;
  RETURN FALSE;
END add_ref;

PROCEDURE remove_ref*( mo : MetaObj; ro : REFLOBJ.ReflObj )
VAR i : INTEGER;
BEGIN
  i := 1;
  WHILE ( i <= 24 ) DO
    IF ( ro = mo.referents[i] ) THEN
      mo.referents[i] := mo.referents[0];
      mo.referents[0] := -i;
    ELSE INC( i );
    END;
  END;
END remove_ref;

PROCEDURE InitMetaObj*( mo : MetaObj )
VAR i : INTEGER;
BEGIN
  i := 0;
  WHILE ( i <= 24 ) DO
    mo.referents[i] := -(i+1);  INC( i );
  END;
  mo.referents[i] := NIL;
  mo.HandleMessage := HandleMessage;
  mo.add_ref := add_ref;
  mo.remove_ref := remove_ref;
END InitMetaObj;

END MODULE.
```

Codice Oberon 2.19: Classe MetaObj

riflessivi, e l'oggetto riflessivo deve essere in grado di richiedere una computazione riflessiva e di ripristinare la normale computazione.
Questa capacità è ottenuta incapsulando la classe dell'oggetto che si vuole rendere riflessiva in una classe (ReflObj) che ha questa capacità.
Cominciamo con il descrivere l'implementazione dei canali in Oberon. Come già detto più volte i canali hanno la caratteristica di sopravvivere alla comunicazione e di essere riutilizzati ogniqualvolta sia attivato un metodo che necessita di una computazione riflessiva dello stesso genere e che coinvolge gli stessi oggetti; quindi i canali sono distinguibili per gli oggetti che coinvolgono e per il genere di computazione riflessiva che effettuano.
Deve esistere un'entità in grado di mantenere le informazioni necessarie per assicurare ai canali tale caratteristica. Questa entità, detta dizionario dei canali, è un oggetto istanza della classe Dictionary. I canali vengono classificati in una gerarchia di classi (in base al tipo), che ha come radice la classe channel, ed ognuno di essi differisce per il comportamento riflessivo delle sue istanze; per ottenere le caratteristiche descritte, occorre mantenere le informazioni in modo che qualsiasi tipo di canale sia in grado di reperirle. Quando si è esaminato il linguaggio C++ si è osservato che il modo migliore per ottenere tale caratteristica è quello di usare una variabile condivisa da ogni possibile classe della gerarchia dei canali; l'idea nel caso del C++ si poteva avvalere del costrutto static per essere implementata. Oberon invece non è dotato di tale od equivalente costrutto e per la natura particolare degli oggetti in Oberon, questi non possono condividere le informazioni.
Un primo modo per risolvere il problema è quello di definire nel modulo in cui si definisce la classe Channel una variabile (OpenChannel), con la funzione di dizionario dei canali avente le procedure per gestirla, di marcarla come esportabile ed inoltre di inizializzarla nella parte di inizializzazione del modulo; procedendo così tutti i canali istanza di tale classe condividono tale sorgente di informazioni ed i moduli, in cui si definiscono altri tipi di canali, dovendo importare questo modulo (ovvio per poter estendere il tipo channel) e condividono, anch'essi, il dizionario dei canali.
In alternativa, potendo utilizzare, invece di Oberon la sua estensione EOS, si può sfruttare la clausola *SHARE* offerta da EOS e quindi inserire, come già spiegato per il linguaggio C++, tale variabile nella classe channel stessa. Ovviamente questa seconda soluzione risulta più elegante e compatta e come tale preferibile. Quindi nel seguito della trattazione il codice viene espresso in EOS, precisando se è possibile o meno, caso per caso, anche una specifica in Oberon.
Seguendo la filosofia di programmazione del paradigma object-oriented, la variabile OpenChannel è definita di tipo dictionary, che è una classe dotata di un metodo per l'inserimento (insert), uno per il recupero (get) ed uno per la cancellazione (delete), il cui approfondimento è rimandato al paragrafo [2.6].
I metodi necessari al canale sono uno per la gestione della computazione riflessiva ed il ritorno al livello sottostante (HandleMessage), ed un altro per la creazione del canale (il "costruttore" InitChannel).

InitChannel

Come si è visto in più occasioni, il mantenimento del dizionario dei canali è gestito dalla procedura di creazione|inizializzazione dei canali stessi, in questo caso la procedura Init_Channel.
La procedura Init_Channel, conoscendo le entità tra cui instaurare un canale ed eventualmente il genere del canale, verifica se esiste già un canale con tali requisiti, in caso affermativo lo attiva altrimenti ne crea uno nuovo. Si è detto eventualmente perché, dato che ogni genere di canale ha una propria operazione di inizializzazione, questa informazione può essere implicita nel codice della procedura stessa, nel caso di Oberon, in contrapposizione al caso del C++ si adotta quest'ultima tecnica. La definizione in Oberon è:

```
PROCEDURE InitChannel( r, s : ReflObj; m : Message ) : Channel
```

Ovviamente InitChannel deve conoscere anche il messaggio che gli oggetti si scambiano, cioè il metodo ed i parametri attuali di quest'ultimo che il sender richiede di attivare al receiver. Essi vengono comunicati tramite un'istanza della classe Message.

Nota La definizione di questa procedura è identica per Oberon e per EOS perchè non è legata a nessun tipo (procedura non record-bound).

HandleMessage

Il canale deve essere in grado di effettuare la computazione riflessiva vera e propria, questa viene effettuata tramite il metodo HandleMessage. Ovviamente siccome le classi di canali differiscono per il comportamento riflessivo delle proprie istanze (genere), il metodo HandleMessage è differente in ogni classe, in quanto gestore del comportamento riflessivo. Nel caso della classe Channel, base di tutta la gerarchia di classi, il metodo HandleMessage si limita ad effettuare il passaggio dal meta-livello al livello inferiore.
Il metodo in questione deve conoscere il sender, il receiver ed il messaggio che si scambiano, informazioni che sono però implicite nel canale stesso, quindi, per le limitazioni più volte citate, la definizione in Oberon di tale metodo è:

```
PROCEDURE HandleMessage( ch : Channel )
```

In EOS, il metodo HandleMessage è una procedura legata al tipo e non una semplice procedura, la sua definizione è:

```
PROCEDURE HandleMessage [self : Channel] ()
```

```
MODULE CHANNEL;
IMPORT REFLOBJ, DICTIONARY;
DEFS ReflObj = REFLOBJ.ReflObj;
        ReflObjDesc = REFLOBJ.ReflObjDesc;
        Message = REFLOBJ.Message;
        Dictionary = DICTIONARY.Dictionary;
TYPE Channel* = POINTER TO ChannelDesc;
        HandleMessageType* = PROCEDURE [self : Channel] ();
        InitChannelType* = PROCEDURE ( s, r : ReflObj; m : MEssage ) : Channel;
        ChannelDesc* =
          RECORD( ReflObjDesc )
            sender*, receiver* : ReflObj;
            message* : Message;
            tag_type* : ARRAY 25 OF CHAR;
            HandleMessage* : HandleMessageType;
            SHARE OpenChannel* : Dictionary;
          END;

PROCEDURE InitChannel*( s, r : ReflObj; m : Message ) : Channel;
VAR ch : Channel;
BEGIN
  ch := NIL;
  IF ( Channel.OpenChannel # NIL ) THEN ch := Channel.OpenChannel.get( s, r, ``CHANNEL'' ) END;
  IF ( ch = NIL ) THEN
    NEW( ch );
    InitReflObj( ch );
    ch.sender := s;
    ch.receiver := r;
    ch.message := m;
    ch.HandleMessage := HandleMessage;
    ch.OpenChannel.add( ch );
  END;
END InitChannel;

PROCEDURE HandleMessage* : HandleMessageType;
BEGIN
  self.receiver.apply( self.message );[16]
END HandleMessage;

BEGIN Channel.OpenChannel.Init;

END CHANNEL;
```

Codice EoS 2.20: Classe Channel

Nel riquadro [2.20] è riportato il codice EOS della classe Channel.

Una volta vista l'implementazione dei canali, si può trattare l'implementazione degli oggetti riflessivi (ReflObj); questi come già detto, devono essere in grado di rispon-

[16] Nel caso di OBERON|EoS non è stata considerata l'eventualità di valori di ritorno dall'attivazione dei metodi riflessivi, questa situazione può essere facilmente gestita passando il messaggio per variabile e non per valore e prevedendo un apposito slot da riempire con il valore restituito.

dere al metodo send ed al metodo apply.

---

send

Il metodo send si occupa di attivare la computazione riflessiva. Oltre al sender, al receiver, al metodo da eseguire riflessivamente ed i parametri con cui deve essere attivato, il metodo send deve conoscere il genere di comportamento riflessivo che è richiesto. Tale comportamento è automaticamente determinato dall'inizializzatore utilizzato per costruire|recuperare il canale che sovraintende la computazione. Ovviamente le informazioni riguardanti il metodo da attivare ed i suoi parametri sono rappresentati da un messaggio costruito seguendo l'idea di Wirth esposta in precedenza. Tramite il tipo del messaggio si è in grado di risalire al metodo da attivare. Quindi la sua definizione in Oberon è:

```
PROCEDURE send( s, r : ReflObj; init : InitChannelType; m : Message )
```

Siccome, per quanto spiegato all'inizio della sottosezione, si usa il linguaggio EOS invece di Oberon, in realtà la definizione usata della send è:

```
PROCEDURE send[self : ReflObj]( r : ReflObj; init : InitChannelType; m : Message )
```

---

apply

Sovraintende il ritorno della computazione al livello sottostante dopo la computazione riflessiva; dal momento che deve attivare effettivamente il metodo riflessivo, deve essere in grado di determinarlo e conoscere i parametri su cui attivarlo. Le informazioni necessarie gli sono fornite per mezzo di una istanza della classe Message e tramite il tipo dinamico di quest'ultimo può risalire e quindi attivare il metodo corretto. La definizione di questo metodo nel caso di Oberon|EOS è:

```
PROCEDURE apply[self:ReflObj]( m : Message )
```

Come già visto nel caso del modello precedente questo metodo è dipendente dalla classe che si intende rendere riflessiva, in particolare il codice di questo metodo è dipendente dai metodi che deve essere in grado di attivare.

Nota La dipendenza del metodo apply dalla classe che si rende riflessiva rappresenta un ostacolo alla generalità del meccanismo delle riflessioni obbligando il progettista di una nuova classe riflessiva a definire il proprio metodo apply.

Nel riquadro [2.21] è mostrato il codice EOS relativo alla classe ReflObj.

[17] Perché una procedura o metodo possa essere assegnata ad una variabile in EOS devono essere definite dello stesso tipo e non di tipo anonimo.

[18] Per quanto riguarda i parametri di ritorno delle operazioni applicate in "ritardo" vedere la nota numero 16

```
MODULE REFLOBJ
IMPORT CHANNEL;
DEFS InitChannelType = CHANNEL.Init;
        Channel = CHANNEL.Channel;
TYPE ReflObj* = POINTER TO ReflObjDesc;
        Message* = POINTER TO MessageDesc;
        SendType* = PROCEDURE [self : ReflObj]( r : ReflObj; i : InitChannelType; m : Message );
        ApplyType* = PROCEDURE [self : ReflObj]( m : Message );
        InitReflObjType* = PROCEDURE( VAR ro : ReflObj );
        MessageDesc* = RECORD END;
        ReflObjDesc* =
          RECORD
            send* : SendType;17
            apply* : ApplyType;
          END;

PROCEDURE send* : SendType; 17
VAR ch : Channel;
BEGIN
  ch := i( self, r, m );
  ch.HandleMessage();
END send;

PROCEDURE apply* : ApplyType;18
BEGIN
  IF ( m IS <MethodMessageType1 > ) THEN
    WITH m : <MethodMessageType1 > DO
      self.<MethodWithParameters1 >
    END
  ELSE
    :
    :
  END;
END apply;

PROCEDURE InitReflObj : InitReflObjType;
BEGIN
  ro.send := send;
  ro.apply := apply;
END InitReflObj;

END REFLOBJ.
```

Codice EoS 2.21: Classe ReflObj

<u>Nota</u> Per quanto riguarda la classe ReflObj l'uso del linguaggio EOS invece del linguaggio Oberon non porta grossi benefici, però siccome si è visto che è utile per la condivisione del dizionario dei canali e che quindi lo si è adottato per la classe Channel, si è scelto di usarlo anche per questa classe.

<u>Nota</u> La classe Dictionary viene tralasciata in quanto la definizione è completamente analoga a quella fornita per il C++ ed inoltre viene più esaustivamente trattata nella sezione [2.6].

2.3 Riflessione Strutturale

La riflessione strutturale, come già accennato, è un tipo particolare di computazione riflessiva che riguarda informazioni sulla struttura delle entità coinvolte, in particolare nozioni come il nome dei metodi attivati, i nomi delle variabili di un oggetto o il numero di istanze di una certa classe, e la possibilità di poterle alterare, alterando così la struttura stessa dell'oggetto.

```
class struct_person : public person {
  public:
    struct var_env {
      char *tag_name;
      void *value;
    } var_env[2];

    struct meth_env {
      char *tag_name;
      void *(*f)();
    } meth_env[2];

   //constructor
    struct_person( char *n, int a ) : person(n,a)
      {
        strcpy(var_env[0].tag_name, ''name''); strcpy(var_env[1].tag_name,''age'');
        var_env[0].value = &name; var_env[1].value = &age;
        strcpy(meth_env[0].tag_name, ''incage''); strcpy(meth_env[1].tag_name,''print'');
        meth_env[0].f = incage; meth_env[1].f = print;
      }
  }
```

Codice C++ 2.22: classe struct_person

Questo tipo di elaborazione viene anche riferita come "autocoscienza", in quanto il programma che la effettua, gestisce delle informazioni su se stesso di cui normalmente non è "conscio".
Questa proprietà risulta particolarmente utile se si intende costruire un'applicazione che monitorizzi il proprio comportamento. Un'applicazione che esemplifica i concetti, oltre al classico meta-debugger, è il sistema operativo ed il suo meccanismo di gestione delle limitazioni dei diritti degli utenti. Un esempio può essere rappresentato dal sistema UNIX che consente di vietare l'uso di alcuni programmi a certi utenti (come ad esempio i servizi di rete, stile ftp), restrizione che può essere facilmente aggirata copiando o ricompilando gli eseguibili in questione nella propria home directory e rinominandoli. Questo perchè la protezione è basata sul nome dell'applicazione inibita, ed una volta rinominata, il sistema non è più in grado di riconoscerla.
Sfruttando la riflessione strutturale, si può risolvere un problema di questo tipo introducendo, ad esempio, un livello di inibizione più restrittivo, basato sulle chiamate di sistema: la protezione non si basa più sul nome del programma, ma sulle

chiamate che questo effettua. Il sistemista, nel caso dei servizi di rete, dovrebbe inibire le chiamate alla gestione dei socket per quell'utente; il sistema operativo ad ogni chiamata effettua una meta-computazione che controlla se l'utente ha i diritti di eseguire tale system call, ed in caso negativo blocca l'accesso, fornendo così un livello di protezione a prova di ridenominazione.
La riflessione strutturale nella maggioranza dei modelli (fa eccezione solo il modello a meta-classe, in cui la classe coincide con la meta-entità) è lasciata alla classe degli oggetti. Solitamente le informazioni necessarie per questo tipo di computazione sono contenute nella classe, ma sono poi gestite dalle meta-entità. Al contrario, la riflessione computazionale, è gestita completamente dalle meta-entità.
I linguaggi di programmazione adottati (C++ e Oberon) non forniscono la possibilità di accedere ad informazioni strutturali (al contrario di SmallTalk), come tale, questo tipo di computazione deve essere implementata ex-novo.
Una tecnica adottabile in C++ [20] e valida per entrambi i modelli (modello a meta-oggetto e modello a materializzazione del canale) consiste nel costruire per ogni classe che si intende rendere riflessiva una classe discendente da tale classe riflessiva, che mantenga le informazioni sulla struttura della classe, in particolare i valori attuali degli attributi (tramite puntatori) ed il loro nome ed analogamente per i metodi (ricordo che queste sono solo alcune delle possibili informazioni necessarie per effettuare delle computazioni di tipo strutturale, comunque più che sufficienti per la spiegazione).
Vediamo di chiarire la tecnica proposta su un esempio: si consideri la classe person (mostrata nel riquadro [2.5]) e si supponga di voler conservare informazioni strutturali sul nome ed il contenuto degli attributi della classe (name e age). Si aggiungono allora due array; uno funziona da ambiente per questa classe, e contiene in ogni slot oltre al nome della variabile il puntatore alla variabile stessa (in modo che sia possibile ottenerne il contenuto, modificarlo e che le modifiche effettuate dai metodi siano immediatamente accessibili tramite questa struttura). L'altro array contiene il nome ed il puntatore ad ogni metodo (o meglio all'incapsulatore relativo ad ogni metodo) della classe.
A questo punto l'entità che effettua la computazione riflessiva può accedere ad informazioni strutturali tramite gli "ambienti" (ovviamente usando debitamente dbind). Nel riquadro [2.22] è riportato il codice in C++ della classe struct_person come esempio di implementazione della riflessione strutturale.

[20] La discussione è effettuata, per comodità, riferendosi al linguaggio C++, ma può essere comunque riferita ad Oberon senza grossi cambiamenti.

Nota Circa l'efficienza di questa tecnica di implementazione, si osserva che l'array è una struttura dati idonea per i seguenti motivi:

- Il numero ed il nome dei metodi della classe sono noti al momento della compilazione.
- Una classe non ha tantissimi attributi e metodi, quindi le dimensioni degli array sono contenute e le ricerche sono discretamente efficienti, nonostante la semplicità dell'implementazione.
- Se fosse necessario rendere più veloci le ricerche, si può costruire facilmente una funzione di hash.

Come osservato SmallTalk per la sua particolare natura (linguaggio object-oriented puro, interpretato e con binding dinamico) fornisce un ottimo supporto per la riflessione strutturale, rappresentato dall'esistenza di oggetti rappresentanti le classi degli oggetti e contenenti tutte le informazioni necessarie per effettuare la riflessione strutturale. Per dotare un linguaggio fortemente tipato di una gestione della riflessione strutturale più estesa di quella presentata occorre dotarlo di un supporto run-time simile a quello offerto da SmallTalk a tale proposito si veda [Vines-Kishimoto 93] in cui gli autori descrivono un metodo per riprodurre l'ambiente SmallTalk in C++. Una trattazione esaustiva della riflessione strutturale esula dallo scopo di questa tesi, pertanto nel seguito ci si limita a sfruttare la riflessione strutturale offerta dagli "ambienti" ogni qualvolta si necessiti tale comportamento.

2.4 Riflessione sulla Creazione degli Oggetti

I linguaggi fortemente tipati, che sono stati considerati in questo capitolo, si differenziano da SmallTalk per la loro visione delle entità con cui operano; in C++ ed Oberon non tutte le entità sono considerate degli oggetti, ed in particolare la classe è solamente un'entità sintattica che descrive la struttura delle sue istanze.
Una simile visione preclude la possibilità di effettuare azioni riflessive durante la creazione degli oggetti. Ad esempio non è possibile contare in modo trasparente, al contrario di quanto è stato mostrato nelle conclusioni del capitolo [1], il numero di istanze di una classe.

2.4.1 Può avvenire un'attività riflessiva durante la creazione di un oggetto?

Poiché le azioni riflessive vengono effettuate richiamando il metodo send dell'oggetto in questione e successivamente il metodo apply, risulta evidente che se la classe è solo un costrutto sintattico che descrive le sue istanze e non un oggetto, non può

possedere dei metodi da attivare prima che una sua istanza venga creata.
Il modo più immediato per ovviare a questo inconveniente è quello di separare i suddetti metodi send e apply dalla classe e renderli quindi delle procedure autonome. Questa soluzione presenta alcuni inconvenienti:

- L'uso di procedure che non sono dei metodi (cioè che non sono operazioni di nessuna classe) non è supportato da tutti i linguaggi object-oriented, un esempio su tutti EIFFEL; questo linguaggio prevede che ogni procedura sia incapsulata all'interno di una classe, perfino lo stesso programma principale è una classe anche se in effetti non rappresenta alcun tipo di dato astratto.

- La separazione del metodo send dalla classe preclude l'override di tale metodo; quindi se si volesse distinguere il comportamento del metodo send o del metodo apply da classe a classe si dovrebbe definire una procedura send e apply diversa per ogni classe.

- L'introduzione di metodi, separati dalla classe di cui dovrebbero far parte, snatura la concezione propria della metodologia ad oggetti.

Una risoluzione "pulita" del problema, risulta essere dipendente dal linguaggio ad oggetti adottato. Si considerino i linguaggi finora esaminati:

- C++: Questo linguaggio fornisce la classe di memorizzazione *static* che permette, ad un metodo dichiarato come appartenente a questa classe, di essere attivato indipendentemente dalla presenza di un'istanza della classe a cui appartiene.
  Definire i metodi send e apply come *static*, li rende equivalenti ai metodi di classe di SmallTalk e quindi attivabili tramite il costrutto :: (ReflObj::send(...)) oltre che con la solita dot-notation (ro.send(...)). Si osservi che in realtà i metodi *static* sono qualcosa di più dei metodi di classe, in quanto non solo possono essere attivati dalla classe ma anche dalle sue istanze. Questo comportamento li rende più simili ai metodi d'istanza della classe Object di SmallTalk, che sono sia metodi d'istanza delle istanze di una classe che metodi d'istanza dell'oggetto classe, che come tutti gli oggetti in SmallTalk, eredita i metodi dalla classe Object. Poiché invece in C++ non esiste la gerarchia di classi presente in SmallTalk, questi metodi possono essere usati solo da quegli oggetti che discendono direttamente o indirettamente dalla classe in cui i metodi send e apply sono inseriti.
  La classe di memorizzazione *static* offre la possibilità di una soluzione semplice ed elegante al nostro problema.

- Oberon: Il caso di questo linguaggio è più delicato, in quanto non solo la classe è un entità sintattica, ma i metodi sono variabili che richiedono di essere inizializzate prima di poter essere utilizzate, e quindi occorre creare un'istanza

della classe prima di utilizzarli. Come già si fa per l'operazione di creazione degli oggetti, l'unica possibilità è l'uso di procedure esterne.
Una possibilità più elegante invece è offerta dalla sua naturale estensione Oberon-2 (vedi [Mössenböck 91] e l'appendice [B]) che rende il linguaggio un vero e proprio linguaggio object-oriented introducendo il concetto di Type-Bound procedure. Le procedure in questo modo sono rese più simili al concetto di metodo di quanto non fossero le variabili di tipo procedura di Oberon. La stessa estensione fa parte della proposta EOS, (vedere [Ancona-Moreno 94] e l'appendice [B]) che alle procedure legate al tipo derivate da Oberon-2, aggiunge il concetto di Shared Method, fornendo quindi la possibilità di condividere effettivamente fra oggetti i metodi delle classi e permettendo quindi di attivarli indipendentemente dalla presenza o meno di un'istanza della classe.
Quindi in Oberon la soluzione rimane quella di far diventare i metodi send e apply delle procedure separate dalla classe oppure di utilizzare un linguaggio suo derivato nel caso specifico EOS.

2.4.2 Attività riflessiva durante la creazione di un oggetto: i modelli

Si è già spiegato che l'attività riflessiva può essere in un modo o nell'altro svolta anche durante la creazione di un oggetto; come viene svolta ed i problemi che si incontrano svolgendola dipendono dal modello adottato.

Nota nel nostro caso l'oggetto riflessivo è una classe, si continua però ad usare il termine oggetto riflessivo e non classe riflessiva perché con quest'ultimo termine si è soliti indicare una classe le cui istanze sono degli oggetti riflessivi e non la capacità propria della classe di compiere azioni riflessive.

Si considerino i due modelli esaminati in questo capitolo.

Modello a Meta-Oggetto: in questo modello il problema è quello di individuare il meta-oggetto responsabile della computazione riflessiva sulla creazione dell'oggetto, cioè quando non esiste ancora l'oggetto.
In SmallTalk si può associare un meta-oggetto ad una classe, in quanto essa stessa è un oggetto ed esiste a run-time. Essendo un oggetto, essa ha un suo slot meta (ereditato dalla classe Object), distinto da quello delle sue istanze, che mantiene il riferimento al meta-oggetto corrispondente, e che quindi esegue le computazioni riflessive sulla creazione delle istanze.
In C++ ed Oberon la classe non è un oggetto e come tale non esiste a run-time. Quindi non si può mantenere nella classe il riferimento al suo meta-oggetto.
In C++ si può definire lo slot meta come *static*, ma se questo è dichiarato nella classe ReflObj, ogni oggetto istanza di una classe discendente dalla classe ReflObj condividerebbe lo stesso meta-oggetto, facendo così collassare il modello a meta-oggetto nel

modello a meta-classe.
L'alternativa consiste nel lasciare inalterato il riferimento meta al meta-oggetto nella classe ReflObj (che è usato per le azioni riflessive richieste da un oggetto) e nell'inserire il riferimento cmeta nella classe da rendere riflessiva, definendolo *static* in modo che appartenga alla classe e non agli oggetti sua istanza (così lo si inizializza prima dell'inizio del main, in fase di creazione dell'ambiente).
Bisogna a questo punto anche dotare i metodi send e apply della capacità di discernere se la computazione è stata richiesta da un oggetto o da una classe, cioè si effettua un test per determinare l'identità del chiamante (un possibile test potrebbe coinvolgere il segno del numero dei parametri).
Questa soluzione, ovviamente, rappresenta un ulteriore punto di rottura per l'encapsulation della classe, che è comunque risolubile come nei casi precedenti (vedi la sezione [2.5]). Inoltre è necessario accedere tramite send e apply alla variabile cmeta contenuta in una discendente della classe ReflObj, ciò si ottiene dotando la classe ReflObj di un metodo fittizio (get_cmeta), che viene ridefinito nella classe discendente in modo che ritorni il valore corretto di cmeta. Ovviamente anche la HandleMessage deve essere in grado di gestire la situazione in cui l'oggetto riflessivo è una classe, in modo analogo a send e ad apply. Discorso analogo può essere fatto per Oberon solo che allo stato dell'arte, come detto, conviene usare EOS invece di Oberon.

Modello a Materializzazione del Canale: in questo modello l'oggetto riflessivo non mantiene alcun riferimento con il meta-oggetto, in quanto questo "non esiste" e come tale il fatto che la classe sia un'entità sintattica puramente descrittiva non è un problema rilevante, anzi non ci tocca affatto.
Un problema comunque permane ed è quello dei riferimenti al sender ed al receiver mantenuti dal canale e conseguentemente come vengono attivati i metodi dopo l'azione riflessiva. Questo è un problema minore in quanto sia sender che receiver in questo caso vengono a coincidere con la classe ed i metodi che possono essere attivati "in ritardo" sono solo i costruttori. Il costruttore non deve essere attivato da un oggetto, ma come se fosse una procedura, sia in C++ che in Oberon. L'unico reale inconveniente che rimane è la necessità, per adeguarsi al caso generale, di identificare il sender ed il receiver, problema irrisorio che può essere risolto nel modo seguente: si definiscono delle entità particolari, dette <u>target</u>, rappresentanti dei sender e dei receiver fittizi. Questi target devono avere i seguenti requisiti:

- ⋆ Non devono essere dei veri e propri oggetti, è inutile creare un oggetto che non viene poi effettivamente utilizzato.
- ⋆ I target devono avere le stesse caratteristiche dei riferimenti ad oggetti, ma devono comunque essere distinguibili da quest'ultimi.
- ⋆ I target di classi diverse devono essere differenti.

Siccome gli oggetti sono rappresentati da puntatori, cioè da degli indirizzi e che questi sono degli interi, i candidati migliori per i target sono proprio gli interi. Per distinguere i target dagli indirizzi si possono usare quei valori che non possono essere degli indirizzi, cioè se l'architettura della macchina usa solo indirizzi positivi i target saranno interi negativi, se l'architettura della macchina specifica gli indirizzi come interi con il segno, per i target si possono usare quei valori appartenenti agli interi lunghi e non agli interi (ovviamente si modifica anche il tipo del riferimento al sender ed al receiver) e così via.
Quindi ogni classe ha una costante che definirà il suo target. Ovviamente si devono modificare HandleMessage e apply in modo da riconoscere i target ed agire di conseguenza; è sufficiente un test analogo a quello descritto nella trattazione del modello a meta-oggetto, soltanto che è fatto sui sender e sui receiver per determinare se sono oggetti o target.

Si conclude confrontando le implementazioni della creazione riflessiva nei due modelli sopra esposti.
Nel modello a materializzazione del canale le modifiche non di carattere generale sono confinate quasi interamente nella meta entità, mentre nel modello a meta-oggetto le modifiche maggiori sono nell'entità del livello base. Altra osservazione è che le modifiche proprie del modello a materializzazione del canale sono indipendenti dal linguaggio usato, mentre quelle necessarie al modello a meta-oggetto sono invece dipendenti dal linguaggio adottato.

2.5 Modularità delle Riflessioni

Programmazione modulare una volta significava costruzione di programmi come montaggio di piccoli pezzi, generalmente "subroutine". Ma una simile tecnica non può portare reali benefici in termini di estensibilità e riusabilità a meno che i pezzi risultanti (i moduli) non siano autonomi, coerenti ed organizzati in architetture robuste. Qualunque definizione completa della modularità deve affrontare questi problemi.
Da ciò risulta che una singola definizione sarebbe insufficiente; si deve guardare alla modularità da più di una prospettiva. Bertrand Meyer in [Meyer 88] per definire nel suo complesso la modularità ha adottato cinque "criteri" e cinque "principi". Si riporta brevemente quanto affermato da Meyer, per fornire le basi su cui determinare e risolvere i punti di minore modularità delle soluzioni finora presentate e per aggiungere la riflessione a dei linguaggi di programmazione object oriented.
I criteri sono indipendenti: è possibile soddisfarne uno violando invece gli altri. I principi sono logica conseguenza dei criteri.
I cinque criteri aiutano la valutazione dei metodi di progetto dal punto di vista della modularità:

- Decomponibilità Modulare: consiste nella possibilità di scomporre un dato problema in vari sottoproblemi la cui soluzione può essere ricercata separatamente. Criterio importante per lo lo sviluppo modulare di sistemi non banali.
- Componibilità Modulare: consiste nella costruzione di moduli che possono essere combinati liberamente tra loro per poter produrre nuovi sistemi. Criterio importante per la riusabilità.
- Comprensibilità Modulare: consiste nel produrre moduli che possono essere compresi da persone diverse dall'autore. Criterio importante dal punto di vista della manutenzione.
- Continuità Modulare: consiste nel fatto che piccole modifiche apportate al problema si ripercuotono in modifiche a pochi moduli.
- Protezione Modulare: consiste nel fatto che, in architetture che godono di tale criterio, l'effetto di una condizione anormale che capita in un modulo in fase di esecuzione rimane confinato entro il modulo stesso.

A partire dai criteri sopra esposti, seguono i principi che devono essere osservati per garantire un'appropriata modularità:

- Unità Modulari Linguistiche: i moduli devono corrispondere ad unità sintattiche nel linguaggio usato. Questo principio deriva da diversi criteri:
  - Decomponibilità: volendo dividere lo sviluppo dei sistemi in attività separate, allora ciascuna di queste attività deve portare ad un'unità sintattica chiaramente delimitata.
  - Comprensibilità: come si possono comporre unità che non sono unità del linguaggio?
  - Protezione: si può sperare di controllare la portata degli errori solo se i moduli sono delimitati sintatticamente.

  Questo principio toglie ogni speranza di poter realizzare in pratica una buona modularità senza il supporto di un linguaggio adeguato.

- Poche interfacce: è importante limitare il numero totale di canali di comunicazione tra i moduli in un'architettura software. Ogni modulo deve comunicare con quanti meno moduli possibile. La comunicazione può avvenire tra i moduli in una varietà di modi. Questo principio limita il numero di tali collegamenti. Tale principio segue dai criteri di continuità e protezione modulare: se ci sono troppe correlazioni tra i moduli, l'effetto di un cambiamento o di un errore può propagarsi a molti altri moduli.

- Interfacce Piccole (Accoppiamento Debole): se due moduli comunicano tra loro, essi devono scambiare meno informazioni possibile. Questo requisito deriva in particolare dai criteri della continuità e di protezione.

- Interfacce Esplicite: quando due moduli A e B comunicano tra di loro, ciò deve essere evidente dal codice di A, di B o di entrambi. Dietro questo principio ci sono i criteri di decomponibilità e della componibilità, tutti i collegamenti con l'esterno devono essere marcati chiaramente), della continuità e della comprensibilità.

- Occultamento dell'informazione: tutta l'informazione concernente un modulo non deve essere accessibile dall'esterno, a meno che non sia esplicitamente dichiarata come pubblica. L'applicazione di questo principio presuppone che tutti i moduli siano noti al resto del mondo attraverso una qualche descrizione ufficiale, o interfaccia. La motivazione fondamentale che sta dietro questo principio è il criterio di continuità.

Un linguaggio orientato agli oggetti, con la definizione di classe, fornisce un ottimo meccanismo per modularizzare i programmi.
Nel caso delle riflessioni, si può osservare che, sia nel modello a meta-oggetto che nel modello a materializzazione del canale, le capacità riflessive, perlomeno per come sono state implementate sinora, vengono integrate nella classe (tramite l'ereditarietà dalla classe ReflObj) le cui istanze si vuole rendere riflessive, come tale, le entità riflessive e quelle non riflessive vengono a coincidere inscindibilmente (non si può avere due istanze, una riflessiva e l'altra priva delle capacità riflessive, nello stesso programma)[21] abbassando la modularità di queste entità.
Il degrado della modularità si manifesta con delle violazioni ai cinque principi nel seguente modo:

- L'integrazione delle due classi è in chiaro contrasto con il principio di occultamento dell'informazione, in quanto tutte le peculiarità della classe ReflObj sono a disposizione della classe riflessiva.

- Essendo integrati uno nell'altro; i due moduli presentano un accoppiamento molto forte.

e porta i seguenti inconvenienti:

- Rottura dell'incapsulamento della classe con conseguente impossibilità di applicare la riflessione a classi di cui non si possiede il codice sorgente, e quindi applicabile solo ad entità proprie.

[21] Osservazione: essere privo delle capacità riflessive è differente dall'avere la possibilità di avvalersi delle capacità riflessive; come offerto dalla filosofia del send la differenza consiste che nel primo caso l'oggetto non è in grado di rispondere al messaggio send, mentre nel secondo anche se normalmente non viene usato può rispondervi tranquillamente.

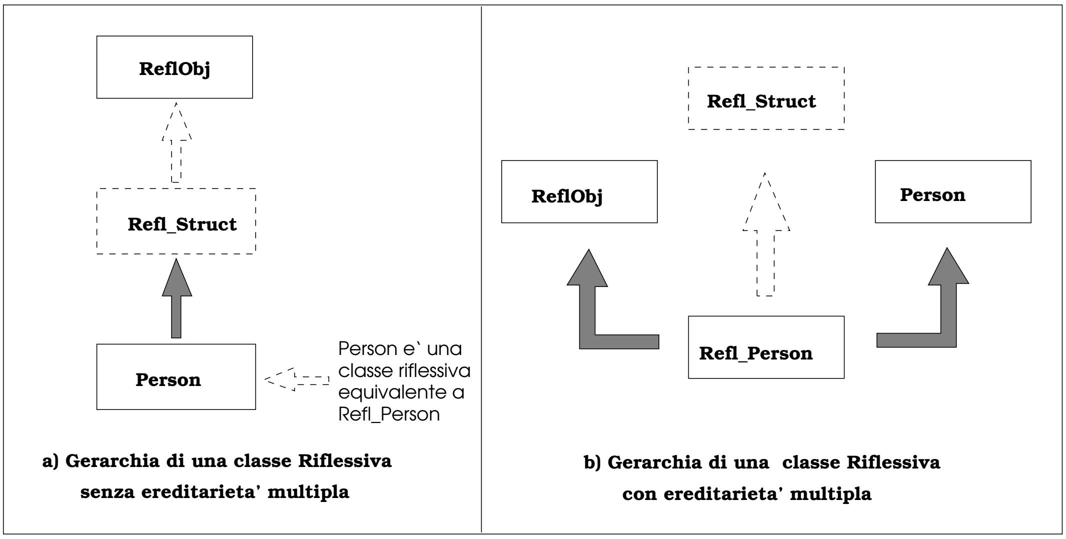


Figura 2.1: **Gerarchia di ereditarietà delle riflessioni**

- Impossibilità di avere oggetti di una classe riflessiva che non dispongono delle caratteristiche riflessive (tutti gli oggetti istanza della stessa classe sono riflessivi o non lo sono).

In Oberon ed in SmallTalk, perlomeno usando gli standard di questi linguaggi, non è possibile realizzare l'architettura riflessiva in modo differente, in quanto non possiedono l'ereditarietà multipla; discorso differente va fatto per il C++, questo possedendo l'ereditarietà multipla, offre un'altra possibilità di implementazione.
Invece di modificare la classe X, che si vuole rendere riflessiva, facendola ereditare dalla classe ReflObj, si costruisce una terza classe Refl_X, che eredita sia dalla classe X che da quella ReflObj, ed in cui si possono inserire le informazioni strutturali senza dover modificare ulteriormente la classe X, al contrario di quanto visto nel paragrafo [2.3].
Addirittura si può farla ereditare da una terza classe Refl_Struct, contenente la definizione degli "ambienti", che vengono inizializzati in quest'ultima. Come risulta evidente, procedendo in questo modo, viene preservato l'incapsulamento della classe X come pure quella della classe ReflObj e si può, non dovendo modificarla, rendere riflessiva anche una classe di cui non si dispone dei sorgenti, inoltre, essendo X e Refl_X due classi distinte, è possibile avere oggetti non riflessivi istanza della classe X e oggetti riflessivi istanza della classe Refl_X con le stesse caratteristiche base. Un'altra caratteristica è la diminuzione dei punti di contatto tra i moduli, con conseguente incremento della modularità. In figura è rappresentata la gerarchia delle classi usata finora (figura [2.1.a]) e quella proposta adesso (figura [2.1.b]).
Si consideri ad esempio la classe person. Sinora renderla riflessiva era equivalente

a modificarne il codice, facendola ereditare dalla classe ReflObj. Questo metodo è scarsamente riusabile. Per questo si costruisce, ora, una nuova classe Refl_Person (riquadro [2.23]) che eredita sia dalla classe ReflObj che dalla classe Person e che contiene anche le informazioni strutturali (vedere sezione [2.3]) ereditandole dalla classe ReflStruct.
Come detto nella sezione [2.4] anche i meccanismi per gestire la riflessione sulla creazione degli oggetti offrono un ulteriore punto di rottura dell'encapsulation della classe dell'oggetto riflessivo.
Sfruttando l'ereditarietà multipla, questo problema viene risolto semplicemente, introducendo le informazioni necessarie nella classe Refl_X.

Nota Siccome queste ultime modifiche sono dipendenti dal modello adottato, il codice presentato nel riquadro [2.23], non le riporta.

Nota Questo aumento della modularità ha, ovviamente, un dazio da pagare ed è rappresentato dall'ambiguità semantica fornita dall'ereditarietà multipla ed al problema del clash dei nomi di metodi ereditati da due classi distinte, per maggiori approfondimenti si rimanda a [Meyer 88].

2.6 Implementazione del Dizionario

Adottando il linguaggio SmallTalk per presentare il modello a materializzazione del canale, per la costruzione del dizionario dei canali si è utilizzata la libreria di classi predefinita di SmallTalk ed in particolare la classe Dictionary. Questa scelta è dovuta ad un insieme di fattori: SmallTalk è adatto alla costruzione rapida di un prototipo sfruttando le classi già esistenti; non essendo un linguaggio particolarmente efficiente non era significativo scrivere una nuova classe, sviluppata appositamente per ottimizzare gli accessi a tale struttura, ed inoltre era preferibile chiarire l'idea su cui il modello si basa che fornirne una implementazione ottimale.
L'obiettivo di questo capitolo è quello di sfruttare la maggiore efficienza dei linguaggi compilati, per poter implementare la riflessione. A tale scopo è d'obbligo utilizzare una struttura dati, che non si riveli essere un collo di bottiglia nella nuova architettura.
Per poter costruire una struttura dati efficiente occorre esaminare le operazioni che essa deve implementare, determinando così quelle usate più frequentemente e quindi progettarle nel modo migliore.
Le operazioni da implementare sono sostanzialmente tre:

- ricerca (get)
- inserimento (insert)
- cancellazione (delete)

```
class refl_person : public person, public ReflObj, public refl_struct {
  public:
    refl_person(char *n, int a) : person(n,a), age(n), name(n),
              refl_struct(2,2,{``name'',``age''},{&name,&age},{``incage'',``print''},{incage,print}){}
  }

class refl_struct {
  public:
    struct var_env {
      char *tag_name;
      void *value;
    } *var_env;

    struct meth_env {
      char *tag_name;
      void *(*f)();
    } *meth_env;

   //constructor
    refl_struct(int n, m, char **vn, void **vv, char **mn, void **(*mf)());
  };

refl_struct::refl_struct(int n, m, char **vn, void **vv, char **mn, void **(*mf)())
{
   var_env = struct var_env new[n];
   meth_env = struct meth_env new[m];
   for(int i=0;i<n;i++) {
     strcpy( *(vn+i), var_env[i].tag_name );
      *(var_env+i).value = *(vv+i);
   }
   for( int i=0;i<n;i++ ) {
     strcpy( *(mn+i), meth_env[i].tag_name );
      *(meth_env+i).f = mf[i];
   }
 }
```

Codice C++ 2.23: classe Refl_Person

La ricerca viene usata ad ogni attivazione[22] di send. Questo metodo demanda la computazione di un altro metodo ad un canale (entità riflessiva). Prima di crearne uno nuovo, verifica la possibilità di usarne uno già esistente; quindi si deve effettuare una ricerca per determinare se esiste o meno un canale con i requisiti richiesti, e se questo non esiste lo si deve creare effettivamente, ed inserirlo nel dizionario per poterlo successivamente riutilizzare. Nel caso peggiore, rappresentato da richieste di computazioni riflessive con necessità sempre diverse e che quindi non permettono mai di riutilizzare i canali, si effettuano tante ricerche quanti inserimenti (al momento dell'attivazione della send si effettua un tentativo di recupero da un canale, questo fallisce ed allora si crea quello nuovo da dover inserire nel dizionario).
Discorso a parte va fatto per l'operazione di cancellazione, questa può essere usata

[22] Si ricerca un canale ad ogni attivazione del metodo send perché non è detto che ogni metodo sia gestito dallo stesso canale ed è quindi necessario ritrovare nel dizionario il canale adatto.

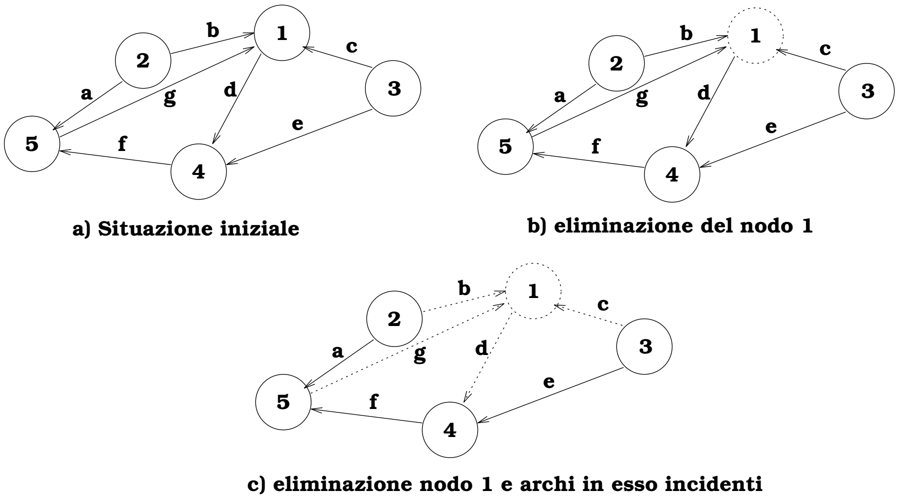


Figura 2.2: **SnapShot del dizionario dei canali**

in diverse occasioni. Un canale può essere eliminato:

- Esplicitamente: il canale non serve più o non si vuole più usarlo allora lo si chiude esplicitamente; in questo caso, conoscendo il canale si hanno tutte le informazioni necessarie (sender, receiver, genere).

- Implicitamente: un oggetto che è sender o receiver di almeno un canale viene distrutto. In questo caso si devono eliminare tutti i canali instaurati con quell'oggetto.

Per chiarire questa affermazione ci si avvale del grafo in figura [2.2.a]; questo grafo esemplifica una ipotetica situazione degli oggetti e canali esistenti (i nodi sono gli oggetti mentre gli archi sono i canali ed il loro orientamento sottolinea qual è il sender e quale il receiver), se in tale situazione si decide di cancellare il canale g, lo si cancella senza coinvolgere altre entità; se invece l'oggetto etichettato con 1 cessa di esistere, si hanno dei canali "appesi" a cui manca uno degli oggetti comunicanti (figura [2.2.b]), quindi devono essere chiusi i canali che hanno l'oggetto 1 come sender (d), ma anche quelli che lo hanno come receiver (b, c, g).
Ricapitolando: le operazioni di inserimento e di ricerca sono quelle usate più spesso e come tali devono essere progettate il più efficientemente possibile; la cancellazione è un'operazione usata più raramente, ed eventualmente implementabile in modo differito rispetto al momento in cui la risorsa è "cancellata", quindi la sua efficienza è meno importante, ma non per questo trascurabile.

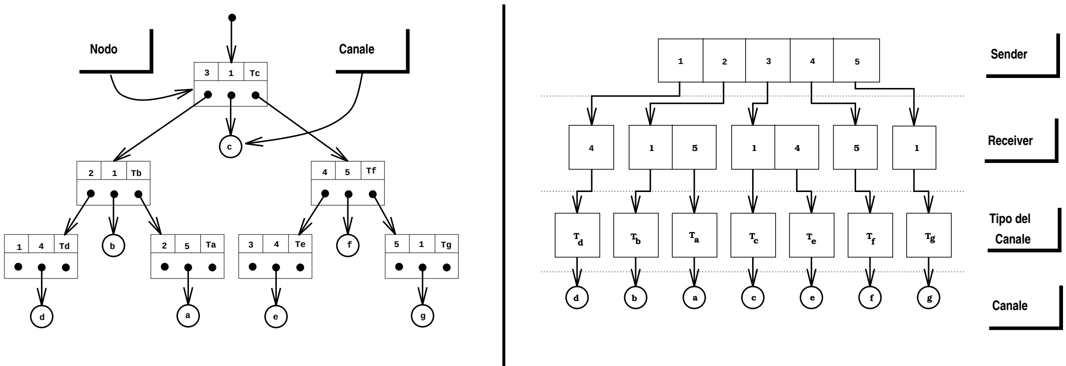


Figura a: Dizionario dei Canali Strutturato ad Albero

Figura b: Dizionario dei Canali Ottimizzato

Figura 2.3: **Confronto delle Strutture per il Dizionario dei Canali**

2.6.1 Struttura ad Albero

I canali vengono classificati in base al sender, al receiver ed al tipo di computazione riflessiva effettuata; quindi l'operazione di ricerca, e conseguentemente quella di inserimento, utilizza queste entità come chiavi per i confronti. La prima struttura dati che viene in mente è un albero di ricerca, meglio se bilanciato o AVL, che usa come chiave per l'ordinamento un record avente come campi questi tre attributi (ordinamento lessico-grafico, prima il sender, poi il receiver ed infine il tipo). Questa struttura garantisce una complessità per l'operazione di ricerca in $\mathcal{O}(\log(n))$, dove con n si intende il numero di nodi dell'albero e quindi dei canali presenti nel dizionario. L'inserimento prevede una ricerca necessaria per il posizionamento nel punto corretto dell'albero ed un eventuale ribilanciamento dello stesso e quindi anche questa operazione è effettuabile con una complessità in $\mathcal{O}(\log(n))$, nel caso dell'AVL (per approfondimenti e conferme su questi risultati si possono consultare i seguenti volumi [Wirth 76] e [Horowitz-Sahni-Andersen–Freed 93]). Più difficoltosa risulta essere la cancellazione. Nel caso della cancellazione esplicita è anch'essa in $\mathcal{O}(\log(n))$, perché si sa qual è il canale lo si cerca e lo si cancella, invece nel caso della cancellazione implicita, occorre determinare tutti i canali instaurati con l'entità che è stata cancellata, e ciò comporta una visita completa dell'albero, per cui l'operazione di cancellazione ha complessità in $\mathcal{O}(n)$ (sia nel caso di una visita depth-first che breadth-first). Chiariamo questa affermazione su un esempio, si consideri la situazione di figura [2.2.a], in cui si tralascia il genere dei canali; si hanno sette canali, i cui valori sono mostrati nella seguente tabella:

sender	receiver	genere	canale
1	4	T_d	d
2	1	T_b	b
2	5	T_a	a
3	1	T_c	c
3	4	T_e	e
4	5	T_f	f
5	1	T_g	g

e la struttura che si ottiene è rappresentata schematicamente in figura [2.3.a]. Se si cancella il nodo 1, come già detto, devono essere chiusi i canali b, c, d, g, ma questi sono situati ovunque all'interno dell'albero e quindi occorre una visita completa per poterli cancellare, da cui la complessità risulta $\mathcal{O}(n)$. È da osservare che comunque occorre visitare tutto l'albero per sincerarsi che tutti i canali con l'oggetto eliminato siano cancellati, e quindi anche nel caso migliore la complessità rimane $\mathcal{O}(n)$, da cui segue che essa è $\Theta(n)$.

2.6.2 Ottimizzazioni al Dizionario dei Canali

La struttura ad albero appena presentata risulta essere particolarmente adatta per essere utilizzata in applicazioni riflessive che usano pochi oggetti riflessivi e soprattutto pochi canali. Se l'applicazione richiede un numero elevato di canali, i continui accessi al dizionario ed il tempo necessario per realizzare le operazioni di ricerca ed inserimento, incidono sensibilmente sulle prestazioni complessive del sistema.
Nel caso si sia interessati a costruire applicazioni riflessive che richiedono un grosso numero di canali, come ad esempio un debugger o applicazioni che utilizzano oggetti persistenti, si possono effettuare due ottimizzazioni. La prima ottimizzazione consiste nell'utilizzare una struttura, per il dizionario, che consenta di realizzare le operazioni di ricerca ed inserimento in tempo costante anche a scapito di una occupazione di memoria maggiore. La seconda ottimizzazione consiste nel progettare il metodo **send** in modo che acceda al dizionario solo quando è effettivamente necessario.

Struttura Alternativa per il Dizionario dei Canali

In questa sezione si esamina una seconda struttura per il dizionario dei canali, con operazioni di ricerca ed inserimento in tempo costante.
Si comincia con lo scomporre la chiave di ricerca in tre chiavi distinte: la prima basata sul sender, la seconda sul receiver e l'ultima sul genere del canale; successivamente si riorganizza l'albero in quattro livelli: nel quarto livello si ha l'insieme dei sender a cui è associato almeno un canale, ognuno di questi sender è collegato con il livello inferiore, nel quale ad ogni sender è associato l'insieme dei receiver per cui esiste un canale instaurato tra quel sender e quel receiver; da ogni receiver si

dipana un collegamento verso il livello sottostante, nel quale ad ogni collegamento è associato l'insieme dei generi dei canali instaurati tra quel sender e quel receiver ed ogni tipo è collegato direttamente, nel primo livello, ai canali. Per chiarire la descrizione si rimanda alla figura [2.3.b] in cui si visualizza come viene immagazzinata la situazione rappresentata in figura [2.2.a] nella struttura presentata.
Analizziamo la complessità delle operazioni di ricerca, inserimento e cancellazione, che si ottengono usando questa struttura.
Inizialmente consideriamo costante la complessità delle operazioni di ricerca, inserimento e cancellazione di un elemento all'interno di un insieme.
Per recuperare un canale è necessario navigare attraverso quattro puntatori, quindi la ricerca di un canale, sotto l'ipotesi che la ricerca della chiave sia fatta in tempo costante, è effettuabile con complessità $\mathcal{O}(1)$. L'operazione di inserimento nel caso peggiore, che consiste nell'inserire un canale per un sender per cui non esistono altri canali, deve aggiungere un sender e la catena che da esso si deriva, con l'ipotesi fatta è ottenibile in tempo costante e quindi con complessità $\mathcal{O}(1)$. Per quanto riguarda l'operazione di cancellazione, quella esplicita è analoga ad un inserimento e come tale ha complessità in $\mathcal{O}(1)$, quella implicita, invece, prevede l'eliminazione completa dell'entità distrutta dall'insieme dei sender (con tutti i canali ad essa collegati) e questo è ottenibile con complessità costante (pongo semplicemente a NULL il puntatore e lo cancello dall'insieme, lasciando la liberazione della memoria al garbage collector del linguaggio, sempre presente nei moderni linguaggi object-oriented come ad esempio Oberon e SmallTalk) e prevede inoltre la cancellazione di tale entità per ogni insieme di receiver, che sono uno per ogni sender e quindi in tempo $\mathcal{O}(\mathrm{k})$, dove k è il numero dei sender nel dizionario ed è minore od al più uguale al numero dei canali, questo trascurando la complessità di garbage-collection per recuperare lo spazio libero.
Considerando il solito esempio la cancellazione del nodo 1 si ripercuote nella cancellazione dei canali b, c, d, g, ed usando la struttura descritta si procede come segue:

- Cancellazione del sender 1 dal primo insieme. $\mathcal{O}(1)$
- Per ogni sender rimasto nell'insieme. $\mathcal{O}(\mathrm{k})$
    - Si controlla se nel relativo insieme dei receiver c'è l'oggetto 1. $\mathcal{O}(1)$
    - In caso affermativo lo si elimina, ponendo a NULL il puntatore e, se era l'unico elemento dell'insieme, si elimina anche il sender relativo. $\mathcal{O}(1)$

Finora si è supposto che il ritrovamento delle chiavi fosse fatto in tempo costante, adesso si esamina la struttura necessaria per ottenere tale complessità; l'idea si basa sulle tecniche di hash ed in particolare sul lavoro di Fagin e soci [Fagin 79], che hanno proposto una forma di hashing dinamico (hashing estendibile), usato per la gestione delle basi di dati.

Hashing Estendibile e Dizionario dei Canali

In questa sezione si descrive brevemente l'hashing estendibile, spiegandone l'impiego all'interno del dizionario dei canali. Per una breve presentazione delle tecniche di hash in generale si può consultare la sezione [C.1] e per un approfondimento si può consultare [Horowitz-Sahni-Andersen–Freed 93].

L'hashing estendibile è un tipo particolare di hashing dinamico in cui alla struttura rappresentante l'insieme si affianca una struttura ausiliaria (index) ed una variabile r, oltre alla funzione di hash (h_r). La variabile r stabilisce il numero di elementi necessari per l'agevole mantenimento dell'insieme. La funzione di hash restituisce una codifica della chiave, detta pseudo-chiave, dipendente dal valore di r, una delle tecniche più adatta consiste nel prendere la chiave, codificarla in binario e restituire i primi r bit come pseudo-chiave.
La struttura ausiliaria è un array indiciato direttamente sulle pseudo-chiavi, avente quindi in ogni momento 2^r elementi che puntano all'elemento dell'insieme, corrispondente alla chiave; in caso di trabocchi si aumenta r di uno e conseguentemente si raddoppia lo spazio dedicato all'indice. I nuovi elementi dell'indice puntano agli elementi da cui si sono sdoppiati, fatta eccezione per l'elemento che ha dato origine al trabocco.

Si esamina ora, nel dettaglio, una possibile implementazione del dizionario dei canali che sfrutta l'hashing estendibile.
Si hanno tre tipi distinti di insiemi da considerare: l'insieme che adotta come chiave di ricerca i sender, quello che adotta i receiver e quello indiciato sul genere dei canali. Siccome sia i sender che i receiver sono rappresentati da puntatori e quindi da locazioni di memoria, la funzione di hash non si basa sui primi r bit, bensì sugli ultimi r, questo perché essendo locazioni di memoria è facile che abbiano la stessa radice e quindi generino troppe collisioni.
Un indirizzo di memoria è rappresentato come un intero lungo (32 - 64 bit a seconda della macchina), il genere dei canali è un intero (16 - 32 bit) facilmente mappabile in un intero lungo, M è una costante rappresentante il numero dei bit occupato da un intero lungo e la funzione di hash che rende gli ultimi r bit è quella che scorre a sinistra di M-r bit, cancellandone così gli M-r bit non interessanti e poi riportando a posto gli r bit che realmente servono.
La funzione di hash adottata in tutti e tre i casi è:

$$\boxed{h_r(k) = ((k \ll (M\text{-}r)) \gg (M\text{-}r))}$$

La ricerca del valore di una chiave è abbastanza semplice e si limita all'applicazione del valore determinato da h_r all'indice (index) che si è costruito

$$\boxed{\text{get(k)} = \text{index}[h_r(k)]}$$

entrambe queste funzioni sono calcolabili con complessità costante. Bisogna osservare che l'operazione di ricerca è usata per determinare se un elemento è contenuto o meno in un'insieme, e siccome può capitare che un elemento non nell'insieme abbia la stessa pseudo-chiave di uno presente nell'insieme, oltre le pseudo-chiavi si confrontano anche le chiavi.
Discorso leggermente più complesso deve essere fatto per l'operazione di inserimento. Si hanno diverse situazioni:

- Situazione iniziale in cui `index` non è ancora stato inizializzato; lo si inizializza come un'array di due locazioni (le uniche pseudo-chiavi che si possono avere con `r` =1 sono 0 e 1) e si inserisce il valore nel posto corretto.
- L'indice è stato creato, ed è stata selezionata una pseudo-chiave non ancora occupata, allora si inserisce il valore nello slot corrispondente.
- Lo slot individuato dalla funzione di hash è già stato occupato e l'indice non ha subito raddoppi; allora si incrementa `r`, si sdoppia l'indice e si ridistribuiscono le chiavi, che hanno dato origine al trabocco, tra la pagina esistente ed una appositamente creata.
- Lo slot individuato dalla funzione di hash è già stato occupato, però l'indice è già stato sdoppiato; allora si ridistribuiscono le due chiavi.

Le prime due situazioni hanno banalmente complessità costante, gli altri due casi, meritano un maggior approfondimento. Lo sdoppiamento, in C++, è effettuato con una chiamata alla funzione *realloc* e la copia dei puntatori è effettuata con un'altra chiamata di sistema *bcopy*; queste essendo delle chiamate al sistema operativo sono considerate di complessità costante. La funzione di distribuzione si limita a riordinare le chiavi che hanno originato la collisione. Nel caso in cui non ci siano ulteriori collisioni durante la ridistribuzione questa è ottenibile in tempo costante.
Questo però non è l'unico caso e purtroppo nemmeno il più frequente e come tale va anche considerata la possibilità di successive collisioni: il metodo per la ridistribuzione tiene conto di tale eventualità ridistribuendo solo le chiavi già presenti nell'insieme e ritentando di inserire la chiave che ha provocato la collisione. Vediamo questa situazione cosa comporta dal punto di vista della complessità.
Nel nostro caso le entità che collidono sono degli indirizzi o comunque valori equiparabili a quest'ultimi. Il caso peggiore è rappresentato da due indirizzi che differiscono solo per il bit meno significativo, ad esempio 111010101 e 011010101, in questo caso, che rammento è il caso peggiore, si avrebbe un numero di collisioni pari al numero di bit che compongono la chiave, che è poi la pseudo-chiave, tale numero è pari a $\log_2(\mathrm{k})$, dove k è l'indirizzo più grande che si può avere. k è una costante dipendente dal sistema, ma pur sempre una costante, come tale anche il suo logaritmo è una costante che indichiamo con K (K= $\log_2(k)$) perciò la complessità nel caso peggiore della ridistribuzione è in $\mathcal{O}(\mathrm{K})$, coincidente con $\mathcal{O}(1)$.

Quindi anche l'operazione di inserimento è in $\mathcal{O}(1)$ nel caso peggiore.
L'operazione di cancellazione esplicita è complementare a quella di inserimento e come tale non viene trattata, e comunque ha la stessa complessità.
A questo punto l'unica cosa che rimane da vedere è come vengono legati i tre insiemi su cui si effettua l'hashing.
Il legame è effettuato sul significato del campo informazione della coppia (chiave, valore). Quando la chiave è un sender o un receiver l'informazione è un insieme, rispettivamente, di receiver o generi e come tale è una struttura hash, se invece la chiave è basata sul genere del canale l'informazione è il canale stesso. Nei riquadri [2.24] e [2.25], è mostrata l'implementazione C++ della classe dictionary.

```
class dictionary {
  private:
    ehash d;
  public:
   // methods
    void get( ReflObj &sender, &receiver, char *tag_type, channel **ch );
    void insert( ReflObj &sender, &receiver, char *tag_type, channel *ch );
    void delete( channel *ch );
  }

void dictionary::get( ReflObj &sender, &receiver, char *tag_type, channel **ch ) {
  ehash *eh;
  *ch = NULL;
  if (!(eh = d.get((long)sender)))
    if (!(eh = eh→get((long)receiver))
      *ch = eh→get((long)tag_type);
}

void dictionary::insert( ReflObj &sender, &receiver, char *tag_type, channel *ch ) {
  ehash *eh, *eh1;
  while( eh = d.get((long)sender) )
    d.insert((long)sender);
  while( eh1 = eh→get((long)receiver) )
    eh→insert((long)receiver);
  eh1→insert((long)tag_type, ch );
}

void dictionary::delete( channel *ch ) {
  ehash *eh;
  if (!(eh = d.get((long)ch→sender)))
    if (!(eh = eh→get((long)ch→receiver)))
      eh→delete( ch→tag_type );
}
```

Codice C++ 2.24: classe dictionary (completa)

La struttura presentata ha due difetti: lo spazio occupato e la complessità della cancellazione implicita in assenza del garbage-collector che è in $\mathcal{O}(n^2)$, dove n è il numero di canali presenti nel dizionario. Maggiori ragguagli a questo proposito sono presentati, rispettivamente, nei paragrafi [C.2] e [C.3].

```
class page {
  private:
    long key;  int r;
    void *value;
  public:
   // constructor
    page( long k, int r, void *v ) : key(k), r(r), value(v) {};
   // methods
    int is_key( long k ) { return( key == k ) };
    long key() { return( key ) };
    void *value() { return( value ) };
    int level() { return( r ) };
    void set_level( r1 ) { r = r1 };
  }

class ehash {
  private:
    const M = sizeof(long);
    page **index;  int r;
    void redistribuite( long k, k1 );
  public:
   // constructor
    ehash() : r(1), index(NULL) {};
   // methods
    void *get( long k );
    int hr( long k, int level = r ) { return((( k<<(M-level))>>(M-level)) };
    void insert( long k );
 }

void *ehash::get( long k ) {
  page *p = *(index+hr(k));
  return( p->is_key(k)?p->value(): NULL );
}

void ehash::insert( long k, void *v = NULL ) {
  if (index == NULL)
    index = (page **)calloc( 2, sizeof(page *));
  if ( !((*index)+hr(k)) )
    ((*index)+hr(k)) = &page( k, hr( k ), 1, ( (v) ? ehash() : v ) );
  else {
    if ( ((*index)+hr(k))->level() == r ) {
      index = (page **)realloc( exp( 2, ++r)*sizeof(page *), index );
      bcopy( index, *(index+exp(2,r-1)), exp(2,r-1)*sizeof(page *) );
    }
    redistribuite( ((*index)+hr(k))->key, k, v );
  }
}

void ehash::redistribuite( long k, k1, void *v ) {
  page *p = ((*index)+hr(k));
  int level = p->level();
  ((*index)+hr(k,level++)) = NULL;
  int pk = hr( k, level );
  p->set_level( level+1 );
  ((*index)+pk) = p;
  insert( k1, v );
}
```

Codice C++ 2.25: hashing estendibile

Quindi con la struttura presentata si sono riuscite ad implementare le operazioni di inserimento, di ricerca e di cancellazione esplicita in tempo costante a scapito dell'aumento dello spazio occupato. Inoltre si è visto che la cancellazione implicita può essere effettuata in due modi; nel primo, in cui si lascia gran parte del lavoro alla routine di garbage-collector del sistema, la complessità è in $\mathcal{O}(n)$ come con l'altra struttura, nel secondo modo in cui la cancellazione fa parte dell'algoritmo, la complessità rispetto alla prima struttura degrada da $\Theta(n)$ a $\mathcal{O}(n^2)$.

Applicazione del Principio di Località

Il dizionario dei canali è il punto nevralgico del modello a materializzazione del canale, però la ricerca del canale per ogni attivazione di un metodo riflessivo è inefficiente, in particolare in ambiente distribuito in cui ogni richiesta viene tradotta in scambi di messaggi.
Il primo passo per migliorare l'efficienza è quello di determinare quando è effettivamente necessario ricercare il canale da usare.
È ovvio che la ricerca è necessaria nel momento in cui il canale da usare non è noto all'oggetto riflessivo interessato. Per determinare quando e come tali informazioni sono note all'oggetto riflessivo, l'ideale sarebbe dotare l'oggetto riflessivo (o meglio ancora il metodo **send**) di "memoria" in modo da conservare le informazioni necessarie sul canale tra due attivazioni successive (sono sufficienti il riferimento al receiver, il genere del canale ed il riferimento al canale, non serve il riferimento al sender perché è implicito nell'oggetto riflessivo in questione, *this*).
In questo modo prima di iniziare la ricerca nel dizionario dei canali si verifica se il precedente canale utilizzato soddisfa anche la richiesta riflessiva attuale. In caso affermativo si ha già il canale da usare e si evita l'accesso al dizionario, altrimenti si determina il canale da usare ricercandolo nel dizionario dei canali.
Il procedimento descritto non è altro che l'applicazione del <u>principio di località</u>.
L'introduzione della "memoria" appena descritta può essere effettuata sfruttando la dichiarazione *static* che permette tra un'attivazione di un metodo e quella successiva di mantenere il contenuto degli identificatori locali dichiarati *static*, quindi si modifica come segue il metodo **send**:

```
void *ReflObj::send( ... ) {
  static ReflObj static_receiver;
  static char *static_kind;
  static channel static_channel = NULL;
      :
  if ( ( static_channel == NULL ) || ( !strcmp( static_kind, tt ) ) || ( static_receiver != r ) ) {
    static_channel = (*c)( this, r, tt, m );
    static_receiver = r;
    static_kind = tt;
  }
  return( static_channel.HandleMessage() );
}
```

L'introduzione della "memoria" sull'ultimo canale usato, suggerisce l'estensione della memoria del metodo send a tutti i canali usati eliminando così completamente il dizionario; non si procede in questo modo perché una tale distribuzione dei canali complicherebbe l'operazione di cancellazione implicita coinvolgendo tutti gli oggetti riflessivi alla ricerca dei canali di cui l'oggetto in questione svolgeva il ruolo di receiver. Questa stuazione risulterebbe molto inefficiente in ambiente distribuito dove ogni richiesta è fatta tramite scambio di messaggi.

Ricapitolando, le applicazioni riflessive costruite seguendo il modello a materializzazione del canale necessitano di una struttura che mantenga traccia dei canali attivati (dizionario dei canali).
Se i canali utilizzati sono in numero esiguo si può utilizzare una struttura più semplice, come ad esempio la struttura ad albero presentata, altrimenti, per motivi di efficienza, si può utilizzare la struttura basata sull'hashing che è stata presentata.
Comunque una miglioria applicabile ad entrambi i dizionari, come visto, è rappresentata dall'applicazione del principio di località.

2.7 Conclusioni

In questo capitolo si è visto come dotare di architettura riflessiva alcuni linguaggi tipati staticamente. In particolare si sono determinati i meccanismi necessari per aggiungere la riflessione, secondo il modello a meta-oggetto e il modello a materializzazione del canale, nei linguaggi C++ e OBERON|EOS.
Dalla trattazione è risultato che tali meccanismi sono costituiti da strutture dipendenti dal modello implementato (i meta-oggetti ed i canali) e da una struttura comune ai due modelli con cui dotare le classi le cui istanze si vuole rendere riflessive e che consenta le tre fasi principali della computazione riflessiva: shift-up, passaggio del flusso computazionale dal livello base al meta-livello, effettuato dal metodo send, computazione riflessiva vera e propria, effettuata dal metodo HandleMessage; e shift-down, passaggio del flusso computazionale dal meta-livello al livello base, effettuato dal metodo apply.
Il confronto tra i due linguaggi scelti (C++ e Oberon), ha dimostrato che con alcuni aggiustamenti sono entrambi adeguati a supportare il comportamento riflessivo.
Tra i due comunque è preferibile usare il C++ per una serie di ragioni, alcune marginali, come la maggior flessibilità sulle liste di argomenti (dal punto di vista del programmatore è più comodo attivare un metodo con una lista variabile di parametri che essere obbligati ad avere un numero fisso di parametri e dover costruire un messaggio contenente i parametri) o la possibilità di sviluppare un metodo apply indipendente dalle classi riflessive che si costruiscono. Ma le ragioni più importanti, per scegliere il C++ sono la sua maggiore disponibilità e la presenza dell'ereditarietà multipla (utile

per ottenere una maggiore modularità delle applicazioni, separando completamente la parte riflessiva da quella non riflessiva). Altre ragioni vengono ulteriormente trattate quando suddetto meccanismo per implementare la computazione riflessiva viene portato in un ambiente parallelo.

3 Riflessioni in Ambiente Concorrente

3.1 Introduzione

Nel capitolo [2] si è determinato il meccanismo necessario (cioè la terna di metodi send-HandleMessage-apply più le meta-entità) ad un linguaggio object-oriented fortemente tipato per poter dotare le applicazioni codificate in tale linguaggio di comportamento riflessivo.
I linguaggi sperimentati sono Oberon e C++; proprio quest'ultimo si è rivelato essere la scelta più adatta, soprattutto grazie alla presenza dell'ereditarietà multipla.
Diverse applicazioni riflessive, come la fault tolerance, sono pensate per ambienti concorrenti. In questo capitolo si esaminano i passi necessari per adattare il meccanismo determinato nel precedente capitolo, per entrambi i modelli riflessivi (il modello a meta-oggetto ed il modello a materializzazione del canale), per poter essere usato in ambiente concorrente.
L'esistenza di diversi linguaggi di programmazione concorrente basati sul C++ come COOL, Presto, Act++ e Parmacs, e di tool come PVM usati per aggiungere al C++ la possibilità di costruire applicazioni concorrenti, fornisce un ulteriore motivazione per preferire il linguaggio C++ al linguaggio Oberon per il quale esiste solamente una versione concorrente, implementata da Lalis [Lalis-Sanders 94], che non è conosciuta e diffusa su diverse piattaforme come nel caso dei linguaggi che estendono il C++.

3.2 Terminologia

Le parole concorrente, parallelo e distribuito generano spesso confusione, infatti, rappresentano per tutti diverse sfaccettature dello stesso concetto, ma non è ben chiaro quale particolare aspetto ognuno di tali termini rappresenti. Per evitare ulteriori ambiguità, generando confusione nel seguito, questi tre termini vengono usati con il seguente significato:

Concorrenza: con il termine concorrenza si intende la possibilità di effettuare azioni "contemporanee", astraendo da come queste vengono effettivamente realizzate, cioè astraendo dall'architettura hardware sottostante.

Parallelismo: con il termine parallelismo si intende una possibile realizzazione della concorrenza su architetture multi-processore, come possono essere gli ipercubi ed i transputer.

Distribuzione: con il termine distribuzione si intende una possibile realizzazione della concorrenza su un'architettura a scambio di messaggi, come può essere una rete di workstation.

Il significato adottato è il significato utilizzato in [Ben-Ari 90].

3.3 Programmazione e Linguaggi Concorrenti ad Oggetti: Aspetti Generali

La programmazione concorrente ad oggetti è un tipo di programmazione evolutasi dalla programmazione ad oggetti tenendo conto della presenza di attività multiple e comunicazioni simultanee.
Questo tipo di programmazione risponde particolarmente bene a due necessità della programmazione attuale e futura:

1. Il bisogno di metodologie per poter gestire la necessaria decomposizione dei problemi e dei programmi, la cui complessità va via via crescendo, così che si possa gestire la loro continua evoluzione ("modularizzare" il problema in problemi più semplici basati sui dati piuttosto che sulle funzioni).

2. Lo sfruttamento delle prospettive di efficienza crescente offerta dalle nuove macchine ad archittettura parallela.

La programmazione concorrente ad oggetti generalizza i principi e le possibilità della programmazione ad oggetti sequenziale, e si basa sull'integrazione della nozione di attività con quella di oggetto. È naturale identificare gli oggetti come unità di attività concorrente (un'applicazione sviluppata secondo la metodologia object-oriented può essere pensata come un insieme di oggetti cooperanti concorrentemente) ed associare la sincronizzazione di queste attività con la trasmissione di messaggi agli altri oggetti.
Esistono, comunque, diversi problemi aperti nella programmazione ad oggetti concorrente come la difficoltà di conciliare la concorrenza con l'ereditarietà e l'incapsulation [Snyder 86, Briot-Yonezawa 87, Kafura-Lee 89, Matsuoka-Yonezawa 92].
Questa unione preserva molti dei vantaggi offerti dalla programmazione ad oggetti tradizionale, rinforzando inoltre l'autonomia degli oggetti.

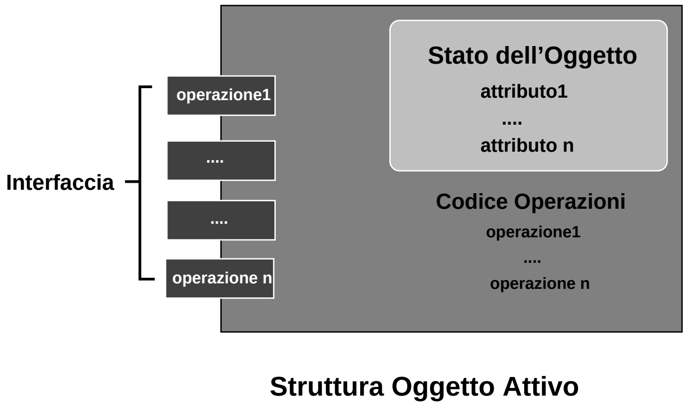


Figura 3.1: **Schematizzazione degli oggetti**

Gli oggetti dotati di attività autonoma sono detti <u>oggetti attivi</u>. La comunicazione tra oggetti attivi avviene attraverso lo scambio di messaggi.

I messaggi interpretabili da un oggetto attivo sono definiti da un'interfaccia, che è costituita dalle operazioni pubbliche dell'oggetto. Quindi un oggetto attivo (schematizzato in figura [3.1]), concettualmente, può essere visto come un'entità autonoma, che possiede un proprio stato e comunica con gli altri oggetti, influenzando così il proprio ed il loro stato, tramite l'interfaccia.

Prima di entrare nell'analisi delle riflessioni in ambiente concorrente si descrivono le principali caratteristiche dei linguaggi di programmazione concorrente ad oggetti, per una loro trattazione esaustiva si rimanda a [Wyatt-Kavi-Hufnagel 92] e [Chin-Chanson 91].

I linguaggi di programmazione concorrente ad oggetti sfruttano i benefici della programmazione orientata agli oggetti, in ambiente concorrente.
Questo tipo di linguaggi possono essere sviluppati seguendo i seguenti criteri:

- Integrando un linguaggio object-oriented con un sistema operativo distribuito come nel caso di Emerald [Andrew-Hutchinson-Jul-Levy 86].

- Inserendo dei costrutti per gestire la concorrenza in un linguaggio object-oriented già esistente come nel caso dei derivati di SmallTalk (Concurrent SmallTalk [Yokote-Tokoro 86, Yokote-Tokoro 87] e la sua più recente evoluzione Concurrent SmallTalk-90 [Okamura-Tokoro 90] o Distribuited SmallTalk [Bennett 87] od AcTalk [Briot 94]).

- Inserendo dei costrutti per supportare la metodologia ad oggetti in un linguaggio concorrente come nel caso di POOL-T [America 87], di ABCL [Yonezawa 90] e tutti i linguaggi di programmazione concorrente ad oggetti basati sul modello degli attori [Agha 86].

Nel seguito vengono espressamente tralasciati aspetti specifici della programmazione distribuita come il bilanciamento del carico, o la migrazione dei processi, ed altri aspetti pur importanti ma non essenziali per il fine di questo lavoro, come la struttura degli oggetti e la loro rappresentazione in memoria, e la sicurezza e l'affidabilità degli stessi. Ci si concentra, invece, maggiormente su aspetti, sia linguistici che implementativi, dei linguaggi e dei sistemi di programmazione concorrente ad oggetti che sono utili per lo sviluppo di architetture riflessive.

3.3.1 Oggetti Attivi: Oggetti e Processi Gestori

Un sistema software risulta composto da un insieme di oggetti concorrenti (detti attivi) che cooperano tra loro.
Gli oggetti sono considerati come unità di concorrenza del sistema. Un oggetto è composto da diversi thread[1] (attività) ed informazioni incapsulate. I thread costituiscono l'unità di concorrenza interna degli oggetti. Le informazioni vengono condivise dai thread ed accedute in modo consistente dai diversi thread. Il modo più semplice di mantenere la consistenza delle informazioni locali ad un oggetto è quello di assegnare ad ogni oggetto un unico thread (altrimenti occorre introdurre dei meccanismi di sincronizzazione interni all'oggetto per gestire la consistenza dello stato di oggetti multithread). La gestione dell'attività di un oggetto, cioè l'interazione dell'oggetto attivo con altri oggetti attivi tramite l'interfaccia, è effettuata da processi gestori.

[1] Un thread è l'unità di base di utilizzo della CPU e consiste di un program counter, un insieme di registri ed uno stack. Esso condivide con i thread ad esso associati le sezioni codice e dati, oltre che le risorse fornite dal sistema operativo, quali i file aperti ed i segnali. L'insieme dei thread e dell'ambiente da essi condiviso prende il nome di task. Un processo tradizionale equivale ad un task dotato di un unico thread. L'alto grado di condivisione fa si che la creazione dei thread ed il passaggio del controllo della CPU da un thread ad un altro all'interno dello stesso task siano operazioni meno costose del context switch tra processi tradizionali, in quanto non coinvolge alcuna operazione legata alla gestione della memoria.
Sotto molti punti di vista, il comportamento dei thread rispecchia quello dei processi. Un thread può trovarsi in uno dei seguenti stati: *ready, blocked, runnig* o *terminated.* Analogamente ai processi i thread condividono la CPU, e può essere attivo un solo thread alla volta. Ogni thread all'interno di un task viene eseguito in modo sequenziale e possiede i propri stack e program counter. I thread possono generare altri thread e possono sospendere la propria esecuzione in attesa del completamento di una system call; quando si verifica una situazione di questo genere, un altro thread può essere eseguito. A differenza dei processi, i thread non sono tra loro indipendenti; difatti la condivisione dello spazio degli indirizzi all'interno del task consente ad un thread di leggere o scrivere sullo stack di uno qualsiasi degli altri thread del task. Tale struttura non garantisce protezione che, tra l'altro, non dovrebbe essere necessaria, perchè al contrario dei processi i thread appartengono allo stesso utente e dovrebbero essere progettati per cooperare tra loro.

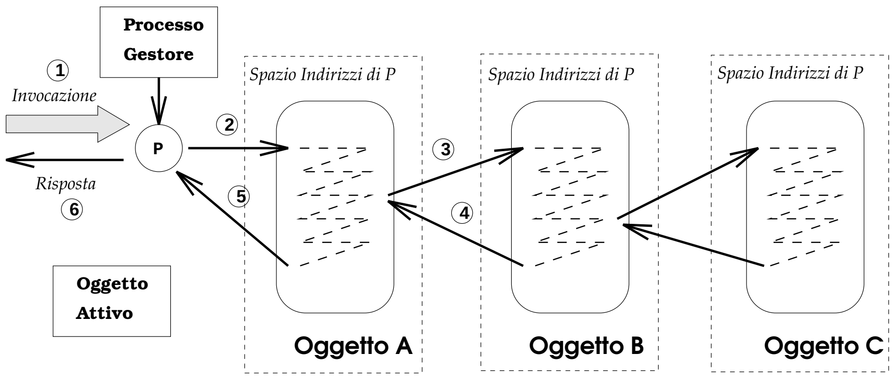


Figura 3.2: **Oggetto Attivo secondo il modello passivo**

La relazione tra i processi gestori e gli oggetti in un sistema object-oriented concorrente caratterizza la composizione degli oggetti attivi cioè come questi ultimi sono definiti e possono interagire tra di loro.
I processi gestori possono essere separati dagli oggetti e legarsi loro solo temporaneamente quando è necessaria la loro unione per gestire un'azione, oppure possono essere accoppiati permanentemente all'oggetto di cui gestiscono la computazione. Questi due approcci corrispondono rispettivamente al modello passivo degli oggetti, ed al modello attivo degli oggetti.
Sia il *modello attivo* che quello *passivo* caratterizzano la composizione degli oggetti attivi. Nel seguito il termine oggetto attivo viene usato per indicare gli oggetti che sono dotati di attività autonoma, trascendendo da come questa attività viene gestita (cioè trascendendo dal modello adottato). In caso di affermazioni valide o caratteristiche solo per un modello, questo viene espressamente indicato e, per evitare ulteriori confusioni, viene distinto dall'uso del corsivo.

3.3.1.1 Modello Passivo degli Oggetti

Nel *modello passivo degli oggetti*, i processi gestori e gli oggetti di un sistema sono entità completamente separate. Un processo gestore non è legato ad un solo oggetto né tantomeno limitato a gestirne solo uno. Un singolo processo gestore è usato per elaborare tutte le operazioni richieste per soddisfare un'azione. Quindi un processo gestore può coinvolgere, durante il suo ciclo di vita, diversi oggetti che vengono di volta in volta mappati nel suo spazio indirizzi.
Quando il flusso computazionale di un'azione passa da un oggetto ad un altro, cioè quando per il corretto espletamento di un'azione intrapresa da un certo oggetto è necessaria la partecipazione dell'elaborazione di un altro oggetto, (passo 3 in figura

[3.2]) il processo gestore sospende[2] momentaneamente la gestione della computazione del primo oggetto per poter gestire la computazione del secondo oggetto coinvolto nell'azione. Nel momento in cui il secondo oggetto termina la computazione richiesta si ha un ritorno del flusso computazionale al primo oggetto coinvolto, il processo gestore allora si occupa di riattivare la computazione di quest'ultimo dal punto in cui era stato interrotto (passo ④). Un vantaggio offerto dal modello passivo degli oggetti è che non ci sono restrizioni[3] sul numero di processi che possono essere legati ad ogni oggetto. Il principale svantaggio è rappresentato dal costo e dalla difficoltà di rimappare gli oggetti attivi nello spazio indirizzi del processo gestore. In questo modello la composizione dell'oggetto attivo è costituito dall'associazione dinamica tra processo gestore e l'oggetto che, in ogni momento, è mappato nello spazio indirizzi di quel processo. Di conseguenza un oggetto attivo perde la propria identità di oggetto.

3.3.1.2 Modello Attivo degli Oggetti

Nel *modello attivo degli oggetti* diversi processi gestori sono creati ed assegnati ad un oggetto per gestire le richieste che può ricevere. Ogni processo gestore è legato ad un particolare oggetto per il quale viene creato. Quando un oggetto viene distrutto con esso vengono distrutti i processi gestori ad esso legati.
Nel modello attivo, l'attivazione di un metodo di un oggetto da parte di un altro oggetto viene gestita come segue: l'oggetto richiedente inoltra la richiesta ad un processo gestore legato all'oggetto che fornisce il servizio, il quale gestisce l'attivazione di tale servizio. In particolare tra i due oggetti attivi si instaura una relazione client-server che si esterna seguendo i seguenti passi (i numeri sono riferiti alla figura [3.3]):

① L'oggetto attivo (client) inoltra una richiesta di invocazione con la lista degli argomenti ad un processo gestore (server) legato all'oggetto attivo invocato.

② Il processo gestore (server) accetta la richiesta, quindi gestisce la computazione dell'operazione richiesta.

③ Se durante l'esecuzione dell'operazione richiesta l'oggetto richiede i servizi di un altro oggetto attivo allora l'oggetto attivo inoltra la richiesta ad un processo gestore dell'oggetto attivo interessato ed attende il risultato. I passi ①-③ vengono quindi iterati.

④ Quando l'elaborazione dell'operazione è completa il processo gestore (server) invia il risultato all'oggetto attivo richiedente (client).

[2] La sospensione è effettuata memorizzando le informazioni sullo stato dell'oggetto necessarie per il ripristino e rilasciando lo spazio indirizzi per la rimappatura del nuovo oggetto.
[3] In pratica l'unico limite è imposto dalla memoria disponibile.

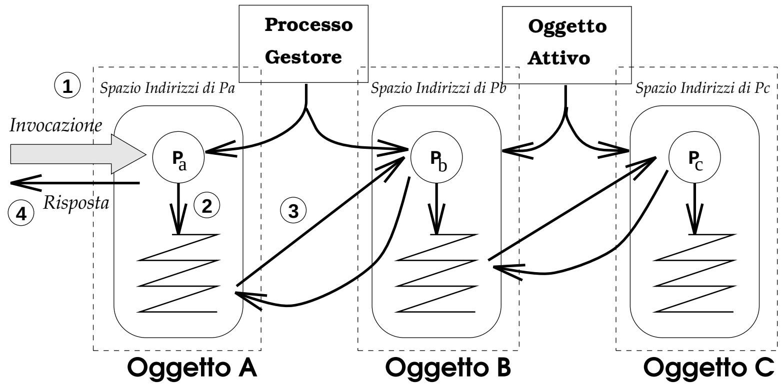


Figura 3.3: **Oggetti Attivi secondo il modello attivo**

Con questo approccio, diversi processi gestori possono essere coinvolti nell'elaborazione di una singola azione (vedi figura [3.3]). Un oggetto attivo è composto dall'oggetto più i suoi processi gestori. Un problema che si incontra adottando il modello attivo degli oggetti è il *deadlock*. Il deadlock occorre se un oggetto non ha un numero sufficiente di processi ad esso associati per gestire le richieste che gli vengono inoltrate.

3.3.2 Gruppi di Processi e di Oggetti

I gruppi di processi sono stati introdotti da Birman e Joseph in [Birman-Joseph 87] e ne hanno fatto uno dei concetti principali del sistema ISIS [ISIS 90].

def. Gruppo di Processi è un insieme di processi raggruppati per cooperare nel fornire un servizio.

Successivamente Liang, Chanson e Neufeld in [Liang-Chanson-Neufeld 90] hanno ripreso la definizione di gruppo di processi e l'hanno estesa agli oggetti.

def. Gruppo di Oggetti è un insieme di oggetti che condividono una o più caratteristiche, interagendo tra loro e coordinandosi per fornire un'interfaccia esterna uniforme. Dall'esterno un gruppo di oggetti appare come un unico oggetto indivisibile.

I gruppi sono un'astrazione utile per:

- Astrarre le caratteristiche comuni dei membri del gruppo, ed i servizi che offrono.
- Incapsulare lo stato interno e nascondere le interazioni tra i membri del gruppo, fornendo un'interfaccia uniforme al resto del mondo.

- Definire un'unità di "costruzione" per costruire sistemi più "grossi" a partire da entità più piccole.

In [Liang-Chanson-Neufeld 90], i gruppi vengono classificati in base alla struttura ed al comportamento, di seguito si riporta brevemente tale classificazione:

- Classificazione strutturale Le operazioni ed i dati manipolati dal gruppo sono gli elementi che determinano il servizio che un gruppo fornisce. È ovvio quindi classificare i gruppi in base a queste due entità.
  - Dati ed Operazioni omogenei (DOO): ogni membro di un gruppo che ricade in questa categoria (gruppo DOO) mantiene una replica completa dello stato del gruppo di oggetti di cui è membro ed implementa un identico insieme di operazioni su questi oggetti. Questi gruppi sono usati per incrementare l'affidabilità di certe elaborazioni.
  - Solo Operazioni Omogenee (SOO): in un gruppo SOO, lo spazio degli oggetti è partizionato tra i membri del gruppo quindi ogni membro mantiene solo una parte dello stato globale del gruppo. Ogni membro supporta un identico insieme di operazioni sulla sua porzione di gruppo di oggetti. Questi gruppi sono usati per distribuire il carico di lavoro.
  - Solo Dati Omogenei (SDO): i membri di un gruppo SDO condividono un insieme di oggetti nello stesso spazio indirizzi. Ogni membro supporta un insieme di operazioni sugli oggetti. Queste operazioni possono o meno essere identiche a quelle degli altri membri del gruppo. Questi gruppi solitamente sono usati per fornire un servizio di gruppo tramite processi cooperanti.
  - Eterogenei: le operazioni e gli oggetti che ogni membro di un gruppo eterogeneo mantiene ed implementa possono essere differenti (eterogenei).

- Classificazione Comportamentale: in accordo con il loro comportamento esterno i gruppi di processi possono essere classificati in due categorie principali: deterministico e non deterministico.
  - Un gruppo di processi è deterministico se ogni membro deve agire su ogni richiesta. Questo comportamento richiede un coordinamento tra i membri del gruppo. Questo tipo di gruppo è principalmente usato per replicare dati e servizi, per ottenere un aumento dell'affidabilità dell'applicazione.
  - Un gruppo di processi è non deterministico se i membri possono reagire in modo diverso ad una richiesta inoltrata al gruppo. Non è necessario un coordinamento tra i membri del gruppo. Questo tipo di gruppo è usato principalmente per distribuire i dati ed il carico di lavoro tra diversi processi, per aumentare la disponibilità dell'informazione e la coordinazione delle risorse.

Inoltre i gruppi di oggetti possono essere statici o dinamici. I gruppi di oggetti sono statici se il numero dei membri ed i membri stessi sono definiti al momento della creazione del gruppo e non possono più variare. I gruppi di oggetti sono dinamici se il numero dei membri ed i membri stessi possono variare durante l'esecuzione. I gruppi di oggetti statici sono supportati dai linguaggi tramite una primitiva di creazione ed una di distruzione che, rispettivamente, creano e distruggono il gruppo (non necessariamente agiscono anche sui membri). I gruppi dinamici sono supportati dai linguaggi tramite una primitiva di arruolamento (*join*) ed una di dissociazione (*disjoint*), che si occupano rispettivamente di arruolare un oggetto attivo in un gruppo (e di crearlo se questo non esiste) e di fare abbandonare il gruppo all'oggetto (e di distruggerlo se non contiene più membri).
Un gruppo di oggetti è considerato come un'unica entità, quindi le richieste vengono inoltrate al gruppo e non ad ogni singolo membro del gruppo. Normalmente la comunicazione con un gruppo si rivela essere una comunicazione in *broadcast* con ogni membro del gruppo. Alcuni linguaggi o sistemi di programmazione, che supportano i gruppi, forniscono la possibilità di comunicare anche solo con un membro del gruppo (come ad esempio PVM [PVM3 94]). Ricevuta una richiesta, il gruppo la gestisce in base alla sua struttura ed al suo comportamento, ad esempio ogni membro del gruppo può reagire alla richiesta ed i membri compiono tutti la stessa azione (gruppo DOO o SOO con comportamento deterministico) o solo alcuni membri reagiscono alla richiesta e compiono azioni diverse (gruppo SDO o eterogeneo con comportamento non deterministico). Le comunicazioni con entità esterne al gruppo sono dette <u>inter-gruppo</u>. I membri del gruppo interagiscono tra loro nel solito modo (come se il gruppo non esistesse). Le comunicazioni tra entità interne al gruppo sono dette <u>intra-gruppo</u>.

3.3.3 Concorrenza Interna degli Oggetti Attivi

Gli oggetti attivi internamente possono essere sequenziali, quasi-concorrenti o completamente concorrenti.
Un oggetto attivo è sequenziale quando è composto da un unico thread che può essere in corso di esecuzione oppure essere in attesa di essere attivato da una richiesta di esecuzione.
Un esempio di esecuzioni quasi-concorrenti è illustrato dai monitor [Hoare 74], che hanno al più un solo thread attivo, ma possono avere thread sospesi in una o più code d'attesa. L'abilità di sospendere i thread internamente fornisce flessibilità, mentre il fatto che al più un thread possa essere attivo garantisce accessi mutuamente esclusivi ai dati locali dell'oggetto attivo. A causa di questi vantaggi, linguaggi di programmazione concorrente ad oggetti, come ABCL, sono basati sulla quasi-concorrenza, anche se il loro protocollo per passare i messaggi trascende il meccanismo di client|server dei monitor.
Gli oggetti attivi completamente concorrenti hanno diversi thread in esecuzione con-

temporaneamente, questi non vengono sincronizzati al momento dell'attivazione come accade negli oggetti attivi quasi-concorrenti, ma nel momento in cui si accede ai dati, ad esempio usando dei semafori o con meccanismi più ad alto livello.
Un esempio di concorrenza completa è rappresentato dai *guardians* di ARGUS (vedere [Liskov-Scheifler 83]). I guardians non sincronizzano i thread, ma si limitano solo ad incrementare il numero dei thread attivi. La sincronizzazione avviene in modo *lazy*, nel momento in cui un thread tenta di accedere a dei dati condivisi con altri thread.

3.3.4 Concorrenza tra Oggetti Attivi

Gli oggetti attivi sono degli oggetti il cui comportamento è caratterizzato da un'attività autonoma. Gli oggetti attivi sono assimilabili ai processi, hanno un proprio spazio degli indirizzi, un loro program counter ed un loro stack pointer.
L'evoluzione di un oggetto attivo è dovuta alla cooperazione con altri oggetti attivi; le attività di due oggetti attivi, quando non cooperano, evolvono concorrentemente. Se l'attività di un oggetto attivo evolve in seguito alla cooperazione con un altro oggetto attivo, allora l'attività dei due oggetti attivi per evolvere correttamente deve essere sincronizzata. Un sistema basato sugli oggetti attivi può essere statico o dinamico. Si parla di un sistema statico quando tutti gli oggetti attivi componenti il sistema vengono creati nel momento in cui si attiva l'applicazione e successivamente non se ne possono creare altri. Si parla di un sistema dinamico quando gli oggetti attivi possono essere creati e distrutti durante l'esecuzione dell'applicazione.
La creazione di un oggetto attivo può avvenire implicitamente od esplicitamente.
Per poter creare esplicitamente un oggetto attivo, il linguaggio di programmazione adottato deve fornire appositi costrutti, equivalenti a quelli per creare dei processi, come ad esempio *pvm_spawn* di PVM. La creazione è implicita quando è il risultato di particolari azioni, ad esempio nei linguaggi basati sul modello degli attori ogni azione da origine ad un nuovo oggetto attivo (attore).

3.3.5 Attivazione dei Metodi e Localizzazione degli Oggetti Attivi

Quando un cliente inoltra una richiesta di servizi ad un oggetto attivo, il sistema è responsabile dei passi necessari per consegnare la richiesta al processo gestore dell'oggetto interessato e per ritornare i risultati al cliente. Come il sistema gestisce le attivazioni dei metodi dipende interamente dal modello degli oggetti adottato. I principali schemi di attivazione sono lo schema <u>a scambio di messaggi</u> e lo schema <u>a chiamata od attivazione diretta</u>.

3.3.5.1 Scambio di Messaggi

Un sistema progettato basandosi sul *modello attivo degli oggetti* tipicamente adotta lo schema a scambio di messaggi per gestire le interazioni tra oggetti attivi.

Quando un cliente richiede un servizio (attivazione di un metodo) ad un oggetto, si effettuano i seguenti passi:

- I parametri della chiamata vengono impacchettati in un messaggio di richiesta.
- Il messaggio di richiesta è inviato ad un processo gestore associato all'oggetto invocato.
- Il processo accetta il messaggio, spacchetta i parametri ed effettua le operazioni necessarie.
- Terminata l'elaborazione, i risultati vengono impacchettati in un messaggio di ritorno che è inviato al cliente.

Queste schema è particolarmente adatto per le comunicazioni di tipo richiesta|risposta tipico di questi sistemi, però impone un'overhead per le comunicazioni tra gli oggetti attivi residenti sulla stessa macchina: esso risulta fondamentale in ambiente distribuito.

3.3.5.2 Attivazione Diretta

Un sistema progettato basandosi sul *modello passivo degli oggetti* tipicamente adotta lo schema a chiamata diretta per gestire le interazioni tra oggetti attivi.
Nel *modello passivo degli oggetti* un solo processo gestore è responsabile di effettuare tutte le operazioni associate ad un'azione. Quindi tale processo migra da operazione ad operazione e da oggetto ad oggetto, quando durante l'azione si effettua una chiamata.
Quando un oggetto richiede un servizio ad un altro oggetto, residente sulla stessa macchina, vengono intrapresi i seguenti quattro passi (figura [3.4.a])

① Lo stato del processo gestore e l'oggetto che sta gestendo vengono memorizzati sullo stack del processo stesso.

② I parametri della chiamata sono messi anch'essi sullo stack.

③ L'oggetto attivato viene caricato in memoria, mappandolo nello spazio indirizzi del processo gestore e quindi si effettua l'elaborazione richiesta.

④ Quando l'operazione termina i risultati sono ritornati al cliente ed il processo gestore viene ripristinato allo stato in cui era prima della chiamata.

Quando un processo invoca un oggetto che risiede su un'altra macchina, allora, oltre ai precedenti, si devono effettuare i seguenti tre passi, i primi due dopo il passo ① ed il terzo dopo il passo ③ (figura [3.4.b])

α I parametri della chiamata vengono impacchettati in un messaggio che viene inviato alla macchina su cui risiede l'altro oggetto.

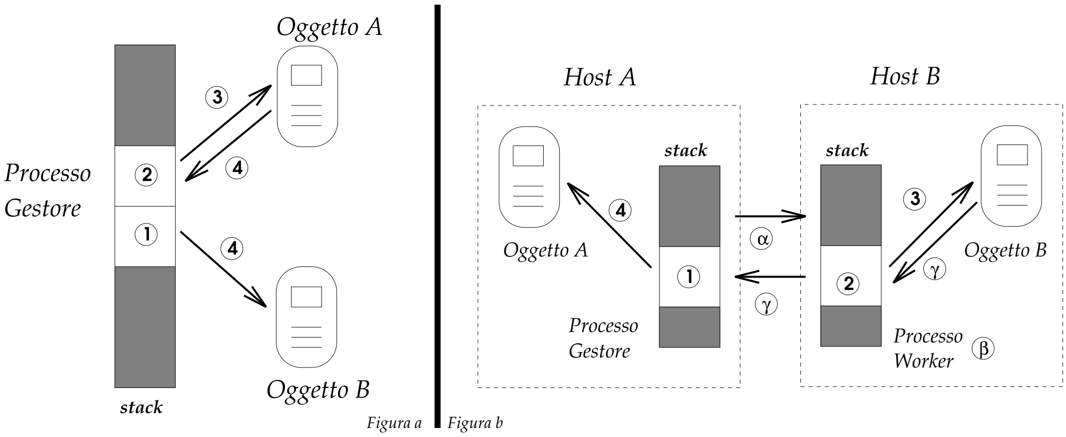


Figura 3.4: **Attivazione dei Metodi: Schema a Chiamata Diretta - a) Caso di Attivazione che Coinvolge Due Oggetti Attivi Residenti sullo Stesso Host b) Caso di Attivazione che Coinvolge Due Oggetti Attivi Residenti su Host Diversi**

β La macchina che riceve questo messaggio crea un processo gestore, detto worker, appositamente per elaborare le azioni richieste.

γ Quando l'operazione è conclusa, i risultati sono impacchettati in un messaggio di ritorno che è ritornato alla macchina del cliente ed il worker creato su quest'ultima viene ucciso.

Questo approccio ha un minore overhead in quanto lo scambio di messaggi viene usato solo quando più macchine sono coinvolte, però in questo caso si ha un costo aggiuntivo dovuto alla creazione e distruzione del processo worker sulla seconda macchina.

Quando durante un'azione viene richiesto un servizio ad un oggetto, il sistema deve localizzare l'oggetto in questione per poter effettuare i passi necessari ad espletare l'operazione richiesta. La localizzazione deve essere trasparente, cioè il cliente non deve essere consapevole della locazione fisica dell'oggetto da attivare. Il sistema assegna un identificatore unico ad ogni oggetto. L'identificatore può essere stabilito seguendo diversi schemi:

- Un approccio è quello di codificare la locazione dell'oggetto attivo entro l'identificatore d'oggetto. Al momento della richiesta il sistema esamina l'appropriato campo dell'identificatore d'oggetto per determinare la macchina su cui l'oggetto risiede. Questo è un approccio diretto ed efficiente però presenta lo svantaggio che gli oggetti non possono migrare su altre macchine pena la perdita dell'identità (la migrazione implica la modifica dell'identificatore d'oggetto).

- Un secondo approccio consiste nel mettere sulle diverse macchine delle entità (processi od oggetti), detti *name server* che, cooperando tra loro, mantengono informazioni aggiornate sulla localizzazione di ogni oggetto nel sistema. Questo approccio consente un'agevole migrazione degli oggetti, però l'aggiornamento di tutti i name server richiede tempo e può creare delle situazioni inconsistenti.

3.3.6 Sincronizzazione ed Attivazione

Tralasciando il tipo di chiamata, cioè se si sta effettuando un'attivazione diretta o tramite scambio di messaggi, in questa sotto sezione si esaminano i possibili approcci di sincronizzazione sulla chiamata tra chiamante ed chiamato. I principali approcci sono tre:

- Sincrono: il chiamante una volta inoltrata la propria richiesta all'oggetto invocato sospende la propria attività in attesa che l'oggetto chiamato espleti il servizio richiesto, ritornando i risultati all'oggetto chiamante, che così può riprendere la propria attività. Questo approccio è simile a quello dei task Ada, [USDoD 80]. Nel momento in cui un task attiva un altro task (rendez-vous), il primo viene sospeso in attesa che il secondo esegua il codice relativo alla richiesta per poi ritornare i risultati al primo che così può riprendere la propria attività.

- Asincrono: il chiamante una volta inoltrata la propria richiesta di servizi ad un altro oggetto, prosegue la propria esecuzione senza attendere il risultato della richiesta inoltrata. Ovviamente è necessaria una qualche forma di sincronizzazione se la computazione dell'oggetto chiamante è basata sul risultato della computazione dell'oggetto chiamato.

- Basato su variabili future: una variante dell'attivazione asincrona è rappresentata dai *futures* o *variabili future*. Un risultato calcolato in modo asincrono viene gestito tramite una struttura dati detta future creata al momento dell'attivazione. I futures consentono la sincronizzazione di operazioni nel momento in cui si usano effettivamente i valori che sono il risultato di computazioni asincrone. Chiariamo il concetto con un esempio: si consideri la richiesta di un servizio a (in modo asincrono) ad un oggetto attivo $\mathcal{A}$, che fornisce un risultato (x_1). Il risultato viene usato successivamente all'attivazione:

  $\mathcal{A}$`.a(x) `x_1`(futures(reply));`

  $\vdots$

  `x := futures(`x_1`);`

  In questo frammento di codice il risultato del servizio viene associato ad x_1 che è una variabile futura; in virtù dell'asincronicità della richiesta l'azione

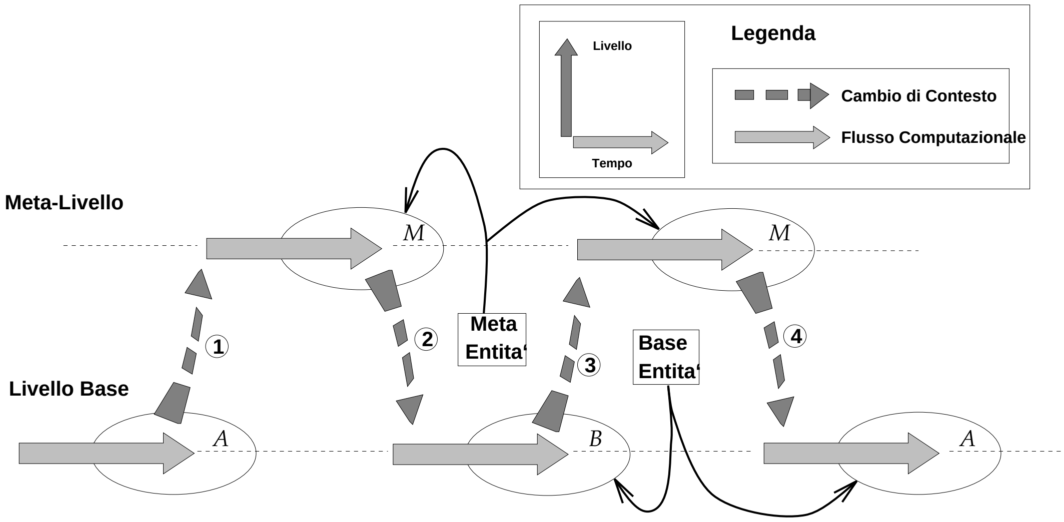


Figura 3.5: **Flusso Computazionale in Ambiente Sequenziale**

continua fino al momento in cui si incontra la prima occorrenza della variabile x_1 che rappresenta il valore di ritorno del servizio. Solo, a questo punto, se il contenuto di x_1 non è ancora disponibile, si attende il completamento dell'azione a per poi proseguire la computazione. Le variabili future sono un costrutto per gestire la concorrenza, utilizzato in prevalenza nei linguaggi concorrenti derivati dal Lisp, come ad esempio Multi-Lisp [Barker-Hewitt 77], Spur-Lisp [Zorn-Ho-Larus-Semenzato-Hilfinger 89] e Q-Lisp [Goldman-Gabriel 89], e nelle derivazioni del Lisp orientate agli oggetti come ABCL [Yonezawa 90].

3.4 Riflessioni e Concorrenza

Nei precedenti capitoli sono stati studiati diversi modelli di computazione riflessiva considerando esclusivamente il loro comportamento nell'ambito di sistemi sequenziali; in questo capitolo ed, in particolare, in questa sezione si è interessati ad introdurre i meccanismi determinati sinora in ambiente concorrente.
Per poter determinare e risolvere i problemi che si hanno nella trasposizione del meccanismo in ambiente concorrente, è necessario analizzare il sistema sequenziale dal punto di vista del flusso computazionale.

3.4.1 Riflessioni in Ambiente Sequenziale

In ambiente sequenziale, in ogni istante solo un oggetto può essere attivo. Vediamo cosa significa dal punto di vista del flusso computazionale, con l'ausilio della figura [3.5].

Qualsiasi modello di riflessione si implementi, nel momento in cui l'oggetto $\mathcal{A}$ del livello base richiede una computazione riflessiva (attivazione del metodo send, passo ① della figura) la sua esecuzione viene interrotta e si attiva l'esecuzione dell'entità $\mathcal{M}$ presente nel meta-livello[4] supervisionante $\mathcal{A}$. La meta-entità $\mathcal{M}$ richiede l'attivazione effettiva del metodo (esecuzione del metodo apply, passo ② in figura) e quindi la sua esecuzione viene interrotta per attivare quella di un'altra entità del livello base ($\mathcal{B}$) che effettua la computazione richiesta. Terminata tale esecuzione l'oggetto $\mathcal{B}$ viene sospeso e si riprende l'esecuzione della meta-entità $\mathcal{M}$ (passo ③ in figura) che completa la gestione riflessiva per poi nuovamente sospendersi per riattivare l'oggetto di partenza (passo ④ in figura).
Come già detto solo un oggetto può essere attivo in ogni momento e soprattutto solo un oggetto del livello base alla volta può richiedere delle computazioni riflessive ad una meta-entità. In particolare in ogni momento il flusso di controllo di un'applicazione riflessiva può essere al livello base o nel meta-livello, ma non in entrambi.

3.4.2 Riflessioni in Ambiente Concorrente

In ambiente concorrente la situazione è ben più complessa. In ogni istante più oggetti, sia del livello base che del meta-livello, possono essere attivi.
Questo vuol dire che un'entità operante nel meta-livello, sia essa un meta-oggetto od un canale, può ricevere richieste di computazioni riflessive da diversi oggetti del livello sottostante. Inoltre potendo più oggetti agire contemporaneamente non è più necessario che l'attivazione di una meta-entità sospenda l'esecuzione dell'entità corrispondente del livello base dando così origine a problemi di sincronizzazione. La possibile distribuzione degli oggetti (meta-entità ed oggetti riflessivi) su diverse piattaforme richiede di scegliere un modello di comunicazione tra oggetti, siano essi dello stesso livello o di livelli differenti, tra quelli visti nelle sezioni precedenti. Inoltre non è più chiaro se in un certo istante il flusso computazionale dell'applicazione sia al livello base o ad un meta-livello.
Nel seguito si esaminano nel dettaglio i diversi aspetti della riflessione in ambiente concorrente.

3.4.2.1 Sincronizzazione delle meta-entità

Nell'applicazione riflessiva che si costruisce sono presenti diverse entità, con scopi e filosofie diverse, interagenti tra loro. Questo porta a considerare diversi livelli di concorrenza tra queste entità [Kono-Tokoro 91]. In particolare si pone la questione di quali entità possono agire "concorrentemente" senza problemi.

[4] In questa sezione si presentano situazioni che interessano entrambi i modelli di riflessione usati, quindi si trascende da quest'ultimi, e si usa il termine generico di meta-entità o entità operante nel meta-livello in luogo di meta-oggetto e|o canale.

Le entità in questione sono sostanzialmente di due tipi: entità riflessive (oggetti operanti nel livello base) e meta-entità (oggetti operanti nei meta-livelli) ed entrambe queste entità possono essere rappresentate da oggetti attivi.
Per la peculiare struttura del paradigma riflessivo l'applicazione viene sviluppata in più strati|livelli (si veda la torre riflessiva presentata nel capitolo [1]) ed ogni entità operante in un certo livello è ignara della presenza e dell'operato dell'entità operanti nei livelli superiori. In base a questa osservazione le entità operanti nello stesso livello sia esso base o meta, possono operare concorrentemente con le restrizioni imposte dal problema. Questo significa che due entità dello stesso livello base possono agire contemporaneamente finché una delle due entità non necessita dei servizi offerti dall'altra entità o accede ad una risorsa impegnata dall'altra entità. In tal caso le due entità devono essere sincronizzate in modo opportuno, cioè, ad esempio, se la computazione dell'entità richiedente si basa sui risultati del servizio richiesto, allora tale entità deve attendere questi risultati, comunque questa sincronizzazione è dettata dal problema.
Discorso più complesso e delicato riguarda la concorrenza tra oggetti attivi operanti in livelli differenti.
Il problema nasce dal fatto che le entità del livello inferiore sono ignare della presenza e soprattutto dell'operato delle meta-entità, come tale non sono in grado di determinare come e quando sincronizzarsi con queste meta-entità.
Questa loro "ignoranza", unita al fatto che le meta-entità possono in qualsiasi istante della computazione riflessiva modificare lo stato[5] dell'entità che monitorizzano (riflessione strutturale) può condurre ad elaborazioni erronee o peggio a risultati differenti da quelli attesi.
Si consideri ad esempio un oggetto attivo $\mathcal{A}$ caratterizzato da un attributo a di tipo intero e da un metodo meth_a.
Il metodo meth_a effettua delle computazioni che non modificano lo stato dell'oggetto $\mathcal{A}$, cioè che non modificano il valore del suo attributo a. Si supponga che in un certo punto del programma sia presente il seguente stub di istruzioni appartenenti all'oggetto $\mathcal{A}$[6]

1) $\mathcal{A}$.a := 1;
2) $\mathcal{A}$.send(meth_a, 0); oppure $\mathcal{A}$.send(this, $\mathcal{A}$, "CHANNEL", Channel, meth_a, 0);
3) $\mathcal{A}$.a++;

Si supponga, inoltre, che ogni meta-computazione richieda la computazione del metodo richiesto e successivamente raddoppi il contenuto dell'attributo a dell'oggetto

[5] Si suppone che tramite la riflessione strutturale si abbia accesso solo allo stato dell'oggetto (in realtà è un meccanismo molto più potente che può coinvolgere anche le variabili temporanee o i metodi), comunque, quale che sia la riflessione strutturale implementata, le problematiche inerenti e la risoluzione delle stesse sono riconducibili a quelle presentate in questa sezione.

[6] Sinora non si sono fatte ipotesi sul formalismo che si adotta per esprimere il parallelismo del sistema, gli esempi quindi vengono espressi tramite il formalismo usato nel precedente capitolo (linguaggio C++) indipendentemente dal tipo di computazione adottata, sequenziale o concorrente.

attivo, come si può vedere nel codice del metodo **HandleMessage** riportato di seguito:

```
   void *HandleMessage( ReflObj *ro, void *(*f)(), message &m )
   {
4)   void *reply = ro→apply( f, m );
5)   ro→var_env[1] ≪= 1;[7]
6)   return( reply );
   }
```

Esaminiamo in ambiente sequenziale il comportamento di questo stub di codice:

① Il valore dell'attributo a viene posto a 1.

② Si richiede la computazione riflessiva del metodo meth_a.

⋮

④ Si esegue il metodo meth_a.

⑤ Si raddoppia il valore dell'attributo dell'oggetto $\mathcal{A}$, a $= 2$.

⑥ La computazione ritorna al livello base.

③ Il valore dell'attributo dell'oggetto $\mathcal{A}$ viene incrementato, a $= 3$.

Quindi in ambiente sequenziale il valore contenuto nell'attributo a una volta completato lo stub di codice è 3. È logico assumere che questo sia anche il valore che si vuole ottenere da una sequenza di esecuzione concorrente corretta.
Si osserva che l'esecuzione del metodo meth_a non fornisce valori di ritorno e non produce effetti collaterali sullo stato dell'oggetto. Questo fatto unito alla trasparenza fornita dalla riflessione, può portare il programmatore a pensare che qualsiasi tipo di attivazione, anche quella non bloccante, porti a risultati corretti, ma questo come si vede di seguito è falso.

In ambiente concorrente l'oggetto $\mathcal{A}$ e la meta-entità che lo controlla sono degli oggetti attivi e come tali le loro esecuzioni si possono svolgere concorrentemente dando origine a parecchie sequenze di attività differenti da quella sequenziale. Consideriamo la seguente sequenza ①-②-④-③-⑤-⑥ ed esaminiamo a quale risultato porta tale sequenza esecutiva (questa è una sequenza possibile considerando l'attivazione del metodo meth_a non bloccante):

① Il valore dell'attributo a viene posto a uno.

[7] Questa istruzione effettua la riflessione strutturale che ci interessa, cioè il raddoppio del contenuto dell'attributo a, per maggiori ragguagli sull'implementazione della riflessione strutturale vedere la sezione [2.3].

② Si richiede la computazione riflessiva del metodo meth_a.

⋮

④ Si esegue il metodo meth_a.

③ Il valore dell'attributo dell'oggetto $\mathcal{A}$ viene incrementato, $\mathsf{a} = 2$.

⑤ Si raddoppia il valore dell'attributo di $\mathcal{A}$, $\mathsf{a} = 4$.

⑥ Si conclude la computazione della meta-entità.

Questa computazione modifica il valore dell'attributo a in un modo che è in contrasto con quanto atteso. È pertanto un esempio di "race-condition".
È evidente che le due entità debbano essere coordinate, ma ciò risulta piuttosto arduo, vuoi per le proprietà di trasparenza delle meta-computazioni, vuoi per il fatto che entità e meta-entità sono costruite in tempi differenti e quindi spesso senza la disponibilità del codice sorgente di entrambe. Non è possibile pensare di coordinare queste entità semplicemente chiedendo al programmatore di prevedere ed evitare le race-condition. La soluzione sta nel definire opportunamente il modo di attivazione delle meta-computazioni, scegliendo tra un comportamento asincrono (o non bloccante) o sincrono (o bloccante) rispetto alle computazioni delle entità del livello base.
Non sincronizzare le computazioni dei due livelli porta, come ha mostrato l'esempio precedente, a possibili race-condition se non si programma esplicitamente la mutua esclusione sulla struttura oggetto della riflessione.
L'approccio sincrono, invece, sospende l'attività dell'entità del livello base, consentendo alla meta-entità la possibilità di modificare lo stato dell'entità del livello base senza conflitti, ottenendo un comportamento analogo a quello in ambiente sequenziale e coincidente quindi con quello atteso.
Da quanto affermato risulta che, per ottenere una elaborazione corretta, in presenza di meta-entità, tutte le richieste di computazione dovrebbero essere di tipo sincrono, cioè l'entità la cui richiesta viene intercettata dalla meta-entità, sospende la propria esecuzione in attesa del compimento della computazione riflessiva in questione, nonostante la computazione non riflessiva non richieda un tale tipo di sincronizzazione.
Un vincolo così pesante riduce notevolmente il grado di concorrenza del sistema. Si può rilasciare tale vincolo sfruttando una delle seguenti tecniche:

- variabili future: La computazione delle due entità prosegue concorrentemente finché l'esecuzione dell'entità del livello inferiore non richiede di utilizzare un elemento dello stato dell'oggetto o il valore di ritorno del metodo che si sta calcolando concorrentemente. Allora la sua esecuzione si sospende in attesa del completamento della computazione della meta-entità per poi riprendere

l'esecuzione; in questo modo lo stato dell'oggetto assume il risultato corretto, cioè quello ottenuto dall'esecuzione sequenziale, evitando quindi la race-condition descritta prima.
Si riconsideri l'esempio precedente: l'attivazione del metodo send è non bloccante, istruzione ②, quindi si attiva la computazione riflessiva, istruzione ④, e l'oggetto $\mathcal{A}$ continua la propria computazione, istruzione ③, tentando di incrementare il suo attributo a. Poiché a è un attributo di $\mathcal{A}$ e come tale è una variabile futura, $\mathcal{A}$ sospende la propria esecuzione ed attende il completamento della meta-computazione per incrementare a; in questo modo il valore di a è determinato univocamente ed è pari a 3 come si attendeva.

- Sincronizzazione gestita dalle meta-entità : Si supponga, inizialmente, che sia l'oggetto riflessivo a stabilire il tipo di computazione necessaria, bloccante o non bloccante, per espletare la propria richiesta.
  Per poter essere in grado di prendere una simile decisione l'oggetto riflessivo deve conoscere, oltre al comportamento del servizio richiesto, il comportamento della meta-entità che lo controlla. Ciò come è già stato sottolineato è in contrasto con il concetto di riflessione e quindi con la struttura riflessiva dell'applicazione.
  È logico supporre che le conoscenze necessarie per prendere tale decisione siano equamente ripartite tra entità e meta-entità.
  L'entità conosce il comportamento del servizio richiesto, dove con comportamento, in questo caso, si intende la possibilità di valori di ritorno da parte del servizio. Mentre la meta-entità conosce il proprio comportamento e quindi se effettua o meno delle computazioni che alterano lo stato dell'entità.
  Si può, quindi, pensare a due livelli di conoscenza che partecipano alla decisione sul tipo di computazione necessaria. Siccome dotare di conoscenza completa l'entità è in contrasto con la proprietà di trasparenza delle riflessioni, non resta che dotare le meta-entità di tale conoscenza, lasciando, quindi, a loro l'onere della decisione sul tipo di attivazione.
  Affinché la meta-entità abbia piena conoscenza, deve conoscere il comportamento del servizio richiesto. Tale informazione le può essere fornita diversificando le attivazioni (sincrona o asincrona) dei servizi a seconda del loro comportamento. Cioè l'entità, che come detto è a conoscenza di tale informazione, non usa più, in modo generico, il metodo send, ma discrimina usando il metodo send se il servizio ha un comportamento tale da richiedere un'attivazione bloccante e asend altrimenti. In questo modo la meta-entità è in grado di ricavare la parte di conoscenza che le mancava.
  Entrambi i metodi, inizialmente, effettuano un'attivazione bloccante, passando il controllo alla meta-entità, è poi quest'ultima che, avendo completa conoscenza, decide se rilasciare le entità, trasformandola in una computazione non bloccante, o continuare con una computazione bloccante.

Si consideri l'esempio di pagina 139, in cui il servizio richiesto (meth_a) ha un comportamento tale da permettere un'attivazione asincrona. Si supponga, quindi, che l'istruzione ② sia effettuata tramite **asend**, allora eseguita l'istruzione ② l'oggetto $\mathcal{A}$ sospende la propria computazione passando il controllo alla meta-entità, questa nonostante l'attivazione asincrona è a conoscenza delle sue azioni (la riflessione strutturale) e quindi non rilascia l'entità finché non ha completato la propria computazione. In questo modo il risultato ottenuto è lo stesso di quello che si sarebbe ottenuto eseguendo l'applicazione in un ambiente sequenziale.

Complicazioni dovute all'uso delle variabili future

La chiamata asincrona usata con le variabili future presenta l'inconveniente di possibili sovrapposizioni di meta-computazioni; cioè se si considerano gli oggetti attivi completamente concorrenti, allora può accadere di avere la richiesta di due computazioni riflessive da parte della stessa entità per la stessa entità. Anche se apparentemente questo non coinvolge attributi dell'entità e quindi variabili future, la seconda computazione riflessiva potrebbe modificarle indirettamente tramite la riflessione strutturale e questo potrebbe portare ad una inconsistenza dei dati.
Chiariamo su un esempio. Consideriamo l'oggetto $\mathcal{A}$ dell'esempio precedente e supponiamo l'esistenza di due meta-entità che hanno rispettivamente il seguente comportamento riflessivo. A seconda del modello di riflessione adottato, potrebbero essere inserite, rispettivamente, in due meta-oggetti distinti o in due canali di tipo diverso. Tali entità sono riferite genericamente con me_1 e me_2.

```
void *HandleMessage( ReflObj *ro, void *(*f)(), message m )
  {
5)    void *reply = ro→apply( f, m );
6)    ro→var_env[1]→value <<= 1;
7)    return( reply );
  }

void *HandleMessage( ReflObj *ro, void *(*f)(), message m )
  {
5̄)    void *reply = ro→apply( f, m );
6̄)    ro→var_env[1]→value++;
7̄)    return( reply );
  }
```

Supponiamo inoltre che un'attività di un oggetto $\mathcal{A}$ sia rappresentata dal seguente frammento di codice

```
1)   A.a := 1;
2)   A.send( meth_a, 0 );           A.send( this, A, ''CHANNEL'', Channel, meth_a, 0 );
     A.set_meta( mo2 );[8]   oppure
3)   A.send( meth_a, 0 );           A.send( this, A, ''CHANNEL2'', Channel2, meth_a, 0 );
4)   futures ( A.a )++;
```

La richiesta di computazione riflessiva è asincrona rispetto all'esecuzione dell'oggetto attivo $\mathcal{A}$, e la richiesta non coinvolge attributi di $\mathcal{A}$ e quindi variabili future, perciò la computazione delle tre entità coinvolte può correre concorrentemente dando origine a diverse sequenze di esecuzione come ad esempio ①-②-③-⑤-⑤̄-⑥-⑥̄-⑦-⑦̄-④ e ①-②-③-⑤-⑤̄-⑥̄-⑥-⑦-⑦̄-④. Le due sequenze sono pressoché identiche eccezione fatta per l'inversione dell'ordine di esecuzione delle istruzione ⑥ e ⑥̄, vediamo il dettaglio:

- ① Il valore dell'attributo a viene posto a 1.
- ② Si richiede la computazione riflessiva del metodo meth_a alla meta-entità me_1, il cui comportamento riflessivo è descritto dalla prima HandleMessage presentata.

⋮

- ③ Si richiede la computazione riflessiva del metodo meth_a alla meta-entità me_2, il cui comportamento riflessivo descritto dalla seconda HandleMessage presentata.

⋮

- ⑤ me_1 richiede all'entità riflessiva la computazione del metodo meth_a.
- ⑤̄ me_2 richiede all'entità riflessiva la computazione del metodo meth_a.
- ⑥-⑥̄ Nella prima sequenza (⑥) me_1 raddoppia il contenuto dell'attributo di $\mathcal{A}$, quindi $\mathsf{a} = 2$. Nella seconda sequenza (⑥̄), me_2 incrementa il contenuto dell'attributo di $\mathcal{A}$, quindi $\mathsf{a} = 2$.
- ⑥̄-⑥ Nella prima sequenza (⑥̄) me_2 incrementa il contenuto dell'attributo di $\mathcal{A}$, quindi $\mathsf{a} = 3$. Nella seconda sequenza (⑥), me_1 raddoppia il contenuto dell'attributo di $\mathcal{A}$, quindi $\mathsf{a} = 4$.
- ⑦ e ⑦̄ Entrambe le meta-entità terminano la propria computazione.
- ④ La computazione dell'oggetto $\mathcal{A}$ riprende con l'incremento dell'attributo di $\mathcal{A}$ che, a seconda della sequenza diventa 4 o 5.

Quindi la race condition permane. Il problema può essere superato propagando la caratteristica di variabili future degli attributi anche alle meta-entità.

Ricapitolando, ogni attivazione di una meta-entità dovrebbe essere di tipo sincrono, cioè le attività delle entità coinvolte vengono momentaneamente sospese in attesa

[8] Nel caso in cui si usi il modello a meta-oggetto questa istruzione permette di cambiare il meta-oggetto che supervisiona l'oggetto; nel seguito della descrizione, questa particolare istruzione, non viene considerata.

dell'espletamento della meta-computazione. Questa limitazione può essere rilasciata con l'uso delle variabili future, però l'implementazione risulta macchinosa perché l'uso delle variabili future deve essere propagato alle meta-entità attivate successivamente. Inoltre l'incremento del grado di concorrenza delle applicazioni che ne consegue è abbastanza limitato. Un effettivo incremento del grado di concorrenza lo si ha esclusivamente nel caso in cui l'oggetto riflessivo effettua operazioni di sola lettura. Inoltre può presentarsi una situazione in cui la meta-entità non modifica in alcun modo lo stato dell'entità riflessiva che monitorizza, mentre l'attività di quest'ultima accede a degli attributi sospendendosi, anche quando non è effettivamente necessario.
Soluzione migliore è rappresentata dal lasciare la sincronizzazione alla meta-entità; con questo approccio diventa più pesante la progettazione delle meta-entità perché si devono distinguere i casi in cui le azioni possono proseguire concorrentemente senza conflitti e quelli in cui si potrebbero presentare dei conflitti, incrementando il grado di concorrenza delle applicazioni (in tutte quelle occasioni in cui la meta-entità non utilizza|modifica le informazioni di stato dell'oggetto riflessivo).

<u>Nota</u> Negli esempi si è fatto coincidere sia il sender che il receiver nello stesso oggetto. Tutto ciò che si è detto rimane valido anche nel caso in cui queste entità siano due oggetti distinti. Nel modello a meta-oggetto solo l'attività del receiver può essere in conflitto con quella del meta-oggetto, quindi la sospensione riguarda soltanto il receiver. Nel modello a materializzazione del canale, sia il sender che il receiver sono monitorati dal canale, quindi sia l'attività del sender che quella del receiver interferiscono con l'attività del canale, come tale entrambe le entità del livello base devono essere sospese. Il ripristino della computazione prima del termine della meta-computazione, invece, potrebbe coinvolgere anche solo una delle due entità.

<u>Nota</u> Quando la meta-entità deve attivare il metodo, la cui richiesta ha intercettato, può dover riattivare un oggetto sospeso. Questa operazione non crea problemi perchè l'oggetto è stato sospeso proprio in attesa di tale elaborazione. Dopo tale esecuzione viene nuovamente sospeso in attesa dell'effettiva riattivazione al termine della meta-computazione, come accade in ambiente sequenziale.

3.5 Modello a Meta-Oggetto e OpenC++

In questo capitolo si intende progettare l'implementazione dei modelli riflessivi esaminati (modello a meta-oggetto e modello a materializzazione del canale), sfruttando linguaggi derivati del C++ in ambiente concorrente.
Il modello a meta-oggetto è un modello di riflessione in uso da alcuni anni; si deve proprio a questa longevità l'esistenza di linguaggi riflessivi basati su di esso, come ad esempio CCEL [Duby-Meyers-Reiss 92], OpenC++ [Chiba 93, Chiba-Masuda 93] e

ABCL-R [Matsuoka-Watanabe-Yonezawa 91] ed alcuni di essi offrono anche la possibilità di sviluppare applicazioni in ambiente concorrente (ABCL-R e OpenC++).
Il linguaggio ABCL-R (Actor Based Concurrent Language-Reflective) sviluppato dal gruppo di Yonezawa, accomuna un'evoluzione ad oggetti del linguaggio Lisp, il modello degli attori ed il modello a meta-oggetto.
In ABCL-R ogni oggetto è attivo ed è modellato dal modello degli attori.
Un attore ha un *indirizzo di posta*, una *casella postale* che mantiene la coda dei messaggi da evadere ed un *comportamento* che può, ad ogni computazione, leggere dalla casella il prossimo messaggio ed in base a tale messaggio può creare un nuovo attore, mandare un messaggio ad un altro attore o cambiare il proprio comportamento in modo da elaborare il successivo messaggio.
L'operazione di cambiamento del comportamento è il concetto portante del modello degli attori; ogni passo computazionale è caratterizzato da questo cambiamento di comportamento, che può manifestarsi con un semplice cambiamento di stato, od anche con una modifica completa dei messaggi (operazioni) a cui può rispondere. In ABCL-R ogni meta-oggetto è un attore.

Il linguaggio OpenC++, sviluppato da Chiba fornisce capacità riflessive secondo il modello a meta-oggetto, basandosi sul linguaggio C++.
OpenC++ è un linguaggio riflessivo sequenziale, al quale l'autore ha fornito, tramite delle librerie, la possibilità di gestire l'interazione tra processi distribuiti su più macchine. La libreria si limita a fornire un framework (le comunità di oggetti) che permette di comunicare ed effettuare attivazioni di metodi remoti tramite i socket. Tale framework non prevede delle primitive da usare per generare delle computazioni concorrenti oltre a quelle implicite del C|C++ (ad esempio *fork-exec*). La concorrenza è, quindi, ottenuta usando tali primitive o eseguendo in background da una shell i diversi processi che compongono un'applicazione.
Siccome esistono già linguaggi di programmazione derivati dal C++, che supportano il modello a meta-oggetto e siccome un'eventuale progettazione ex-novo non differirebbe molto da quella che si effettua per il modello a materializzazione del canale (vedere sezione [3.6]), si è preferito, in questo caso, esaminare l'approccio usato da altri, nel caso specifico si descrive il linguaggio OpenC++.

3.5.1 Il Linguaggio OpenC++

OpenC++ è una variante del linguaggio C++, in cui le attivazioni dei metodi sono di tipo aperto[9]. Il programmatore può usare questa caratteristica per implementare nuove funzionalità sul linguaggio stesso. Il linguaggio OpenC++ fornisce un protocollo a meta-oggetto (MOP) molto semplice per rendere estendibile la chiamata di un metodo.

[9] Metodo aperto è una terminologia adottata da Chiba e sottolinea il fatto che i metodi possano essere estesi dotandoli di comportamenti aggiuntivi.

Il linguaggio OpenC++ adotta il modello a meta-oggetto, con la limitazione che un oggetto riflessivo è monitorato per tutta la durata del suo ciclo di vita sempre dallo stesso meta-oggetto. Inoltre due o più oggetti non possono condividere lo stesso meta-oggetto.
La riflessione è introdotta in OpenC++ da un preprocessore, che agisce essenzialmente sulle linee che iniziano con un commento speciale //MOP. Esso identifica classi, metodi, etc... che si desidera rendere riflessivi, come vedremo in seguito.
Il linguaggio fornisce la possibilità di avere, a partire da una stessa classe, sia istanze riflessive, sia non riflessive. Questo è realizzato dal preprocessore, che a partire da una dichiarazione di classe (per esempio X) crea una classe riflessiva per sottotipazione, a cui da un nome speciale (nell'esempio **Refl_X**). Nella classe discendente vengono inoltre introdotti i metodi necessari alla riflessione. La dichiarazione di una classe a cui appartengono oggetti riflessivi ha quindi la forma:

```
//MOP reflect class X: M;
```

Questa dichiarazione introduce la classe riflessiva **Refl_X**, le cui istanze sono oggetti riflessivi controllati da un meta-oggetto, istanza della classe M. La classe M può essere una qualsiasi classe C++, con l'unico obbligo di dover ereditare dalla classe MetaObj. Il meta-oggetto può essere a sua volta un oggetto riflessivo, controllato da un meta-meta-oggetto. In questo modo il linguaggio implementa la torre riflessiva.
I metodi di un oggetto riflessivo sono divisi in due gruppi, a seconda se l'attivazione del metodo è controllata o meno dal meta-oggetto. I metodi la cui attivazione è controllata dal meta-oggetto sono detti metodi riflessivi. Gli altri metodi vengono attivati con la semantica usuale della chiamata dei metodi del C++. Di seguito è mostrato un esempio di come vengono specificati i metodi riflessivi.

```
class X {
  public:
    X();
    int func1
   //MOP reflect:
    int func2( int );
  private:
    int p;
};
```

Il metodo func_1 non è riflessivo, invece il metodo func_2, che è introdotto dalla direttiva //MOP **reflect:**, è un metodo riflessivo.
Peculiarità di OpenC++ è l'introduzione del concetto di *categoria* . Le categorie sono delle speciali etichette definite dal programmatore ed associate ai metodi riflessivi. Le categorie permettono al meta-oggetto di riconoscere il ruolo del metodo, cioè il meta-oggetto altera l'esecuzione del metodo in accordo alla categoria a cui appartiene.
Il seguente è un esempio di utilizzo delle categorie in cui il metodo **put** è un metodo riflessivo ed appartiene alla categoria **write**.

```
//MOP reflect( write ):
  void put( char c );
```

Nota Si osservi che, essendo le direttive del preprocessore OpenC++ dei commenti nel linguaggio C++, un programma scritto in OpenC++ è compilabile da un compilatore C++, e genera un programma equivalente a quello generato dal "compilatore" OpenC++, senza alcun comportamento riflessivo.

3.5.1.1 Meta Object Protocol

L'esecuzione di un metodo riflessivo è controllata dal suo meta-oggetto. Un meta-oggetto è definito in C++, in modo che la sua classe erediti dalla classe MetaObj, che principalmente definisce i seguenti due metodi:

- **Meta_MethodCall**: questo metodo implementa una chiamata al livello base. È usato quando viene chiamato un metodo riflessivo. I parametri sono: un riferimento al metodo riflessivo, un riferimento alla categoria a cui appartiene e due strutture, contenenti rispettivamente i parametri attuali ed i valori di ritorno.

- **Meta_HandleMethodCall**: questo metodo è usato per eseguire effettivamente il metodo riflessivo. I parametri di questo metodo sono: un riferimento al metodo riflessivo da eseguire, una struttura contenente i suoi parametri attuali ed una struttura per i valori di ritorno.

Per modificare l'implementazione di una chiamata di metodo, il programmatore può definire una sotto-classe di MetaObj, in cui il metodo Meta_MethodCall è ridefinito così che il comportamento del meta-oggetto sia quello desiderato.
Si supponga che f sia un metodo riflessivo. Il preprocessore del linguaggio OpenC++, che esegue le direttive del linguaggio traducendo un programma in OpenC++ nell'equivalente C++, sostituisce al codice dell'attivazione di questo metodo (f) il codice per l'attivazione del metodo Meta_MethodCall. Il primo parametro è un identificatore intero che si riferisce al metodo f, il secondo è un identificatore rappresentante la categoria del metodo f. I parametri attuali del metodo f sono passati come terzo argomento, impacchettati in una struttura simile ad uno stack. Su questa struttura, i parametri attuali vengono impilati e spilati tramite le operazioni push e pop. Le operazioni pop e push sono predefinite per tutti i tipi scalari e per i puntatori, mentre per i tipi strutturati devono essere definite dal programmatore. La codifica degli argomenti in una struttura apposita permette il passaggio degli stessi ad oggetti residenti su macchine diverse.
Il metodo Meta_MethodCall effettua delle computazioni, che normalmente coinvolgono il metodo Meta_HandleMethodCall, per calcolare il valore di ritorno, che a sua

volta viene impacchettato in una struttura analoga a quella usata per i parametri attuali; il valore di ritorno è poi restituito all'entità che ha invocato il metodo.
L'uso del metodo Meta_HandleMethodCall permette l'esecuzione di un qualsiasi metodo riflessivo in ogni momento.

In modo analogo alla gestione dei metodi riflessivi, il linguaggio OpenC++ offre la possibilità di gestire gli accessi riflessivi alle variabili, tramite i metodi Meta_Assign, Meta_HandleAssign per gestire l'assegnazione e Meta_Read, Meta_HandleRead per gestire la lettura del contenuto di una variabile. Le due coppie di metodi hanno comportamento ed utilizzo completamente analogo ai metodi Meta_MethodCall e Meta_HandleMethodCall.
Inoltre sfruttando i metodi Meta_Create e Meta_HandleCreate viene gestito il caso di un costruttore riflessivo; in realtà non è il meta-oggetto a supervisionare la creazione dell'oggetto riflessivo, perché non esiste ancora, ma bensì la sua classe (questi metodi riflessivi sono definiti come *static*).

3.5.2 OpenC++ e Concorrenza

Il protocollo a meta-oggetto (MOP) di OpenC++ non supporta la concorrenza direttamente; viene però fornito un framework, sul quale vengono implementate le funzionalità per il calcolo distribuito. Questo framework è chiamato Object Communities (Comunità di oggetti) e consiste di una libreria di classi di meta-oggetti che aggiungono un nuovo strato al MOP dell'OpenC++.

3.5.2.1 Panoramica sulle Comunità di Oggetti

La funzionalità fondamentale delle comunità di oggetti è quella di gestire un gruppo di oggetti distribuito su diverse macchine (vedere figura [3.6]). Si suppone che ogni oggetto appartenga ad un singolo processo, che ha il proprio spazio indirizzi separato da quello di tutti gli altri. I processi cooperano all'interno della stessa applicazione. Ogni oggetto della comunità di oggetti agisce in modo dipendente dal comportamento degli altri oggetti della comunità. Le chiamate dei metodi degli oggetti sono eseguite in cooperazione dai meta-oggetti così che gli oggetti forniscano una certa funzionalità per il calcolo distribuito.
Le comunità di oggetti sono assimilabili ai canali di comunicazione: una volta creati esistono indipendentemente dal fatto che qualche oggetto ne sia membro, gli oggetti possono entrare o lasciare una comunità dinamicamente e la comunità ha il compito principale di mettere in contatto oggetti distribuiti su più macchine. Si osserva che i concetti relativi alla distribuzione sono contenuti solo nella definizione dei meta-oggetti, quindi la funzionalità fornita è implementata nel meta-livello. Le comunità di oggetti forniscono una separazione netta tra la computazione distribuita e la com-

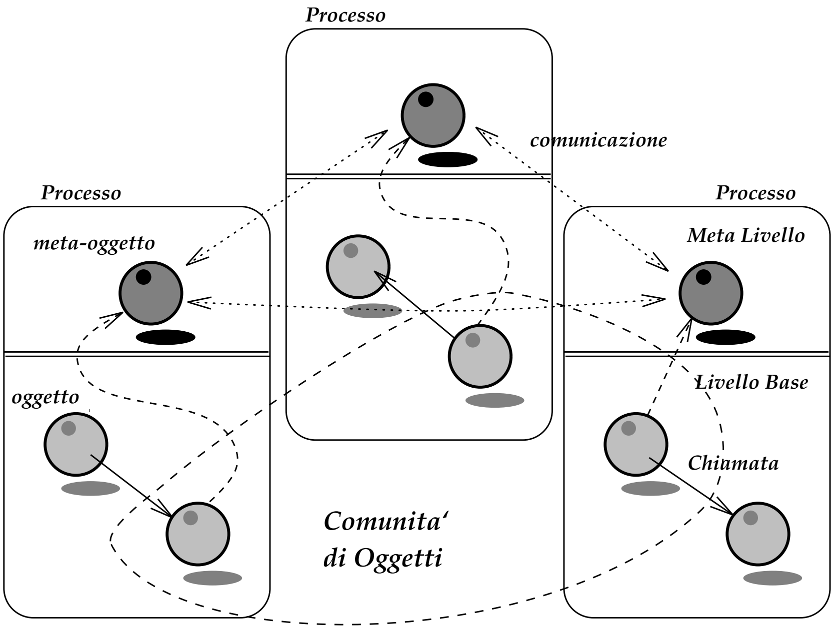


Figura 3.6: **Comunità di Oggetti**

putazione eseguita dall'applicazione.
Nelle comunità di oggetti, un meta-oggetto ha le seguenti capacità aggiuntive:

- Controllo della concorrenza: un meta-oggetto controlla la concorrenza interna dell'oggetto. Può ignorare o ritardare l'esecuzione di chiamate ai metodi dell'oggetto finché alcune condizione non sono soddisfatte. Un meta-oggetto può eseguire più metodi del suo oggetto concorrentemente. Un meta-oggetto può eseguire un metodo del suo oggetto su richiesta da parte di altri oggetti.

- Comunicazione: un meta-oggetto ha due mezzi di comunicazione con gli altri membri della comunità: *remote method call* e *broadcasting* dei messaggi. Un meta-oggetto può chiamare un metodo di un altro meta-oggetto. Questo è fatto in modo simile a quello della chiamata di un metodo locale. Il meta-oggetto chiamante è bloccato finché non riceve una risposta dall'oggetto chiamato (*remote procedure call*). Un meta-oggetto può anche mandare un messaggio a tutti i meta-oggetti della sua stessa comunità (*broadcasting*).

Per estendere gli oggetti con l'abilità di comunicare con altri oggetti e di controllare la concorrenza, la comunità di oggetti fornisce la classe OcCoreMetaObj, che è una sotto-classe della classe MetaObj. Le altre classi di meta-oggetti che devono implementare funzionalità basate sulle comunità di oggetti devono ereditare da questa sotto-classe.

La classe OcCoreMetaObj definisce alcuni metodi per manipolare una comunità di oggetti e le sue funzionalità, come Meta_CreateOc, Meta_DestroyOc, Meta_Join, Meta_Leave, Meta_EventNotify, Meta_Query, Meta_WaitForEvent...
Come esempio si consideri la costruzione di una classe riflessiva che consenta di condividere dati tra oggetti distribuiti. I dati condivisi sono replicati e mantenuti da oggetti appartenenti alla stessa comunità. Compito dei meta-oggetti è quello di mantenere la consistenza dei dati condivisi. Si supponga, inoltre, che il dato condiviso sia un intero rappresentato dall'attributo p della classe SharedData. Per aggiornare il valore di p la classe SharedData fornisce il metodo update.

```
class SharedData {
          ...
  public:
   //MOP reflect:
    void update( int np ) { p= np; };
  private:
    int p;
};

//MOP reflect class SharedData: OcMetaObj;
```

Il metodo update è riflessivo. Se viene creato un oggetto istanza della classe Refl_SharedData, il corrispondente meta-oggetto fa entrare l'oggetto nella comunità. Da questo momento p mantiene lo stesso valore mantenuto dagli altri oggetti della comunità. Se viene invocato il metodo riflessivo update allora questa richiesta è propagata ad ogni membro della comunità, in questo modo è mantenuta la consistenza tra tutti gli oggetti. I metodi del meta-oggetto sono ridefiniti come segue:

```
void OcMetaObj::Meta_MethodCall( Id method_id, Id category, ArgPac &args, ArgPac &reply ) {
 // Notifica agli altri membri di una chiamata al metodo
  Id event = Meta_EventNotify( method_id, args );
 // Attende finché non è serializzata la richiesta
  Meta_WaitforEvent( event, args );
 // Esecuzione effettiva del metodo
  Meta_HAndleMethodCall( method_id, args, reply );
}

void OcMetaObj::Meta_EventCallbackBody( Id method_id, ArgPac &args, ArgPac &reply ) {
 // Se un altro meta-oggetto  riporta un'invocazione di metodo
 // il meta-oggetto  esegue il metodo invocato
  Meta_HandleMethodCall( method_id, args, reply );
}
```

3.5.3 ... Confrontandosi con OpenC++

Si è già schematizzato nel capitolo [2] (senza preoccuparsi di aspetti concorrenti) una strategia di implementazione per il modello a meta-oggetto. In questa sezione si confronta con OpenC++ tale schema di implementazione. Le idee utilizzate sono state poi riprese per implementare il modello a materializzazione del canale come si può vedere nella sezione [3.6].

Innanzitutto OpenC++ è fondamentalmente un preprocessore per il C++ che fornisce al programmatore due livelli di astrazione: il livello base ed il meta-livello.
È compito del preprocessore creare la classe Refl_X a partire dalla classe X, esplicitando la relazione tra entità e meta-entità e traducendo il codice OpenC++ in puro codice C++. Questa nuova classe, come detto, eredita dalla classe originaria X (che ricordo essere in codice OpenC++, anche se differisce dal codice C++ solo per i commenti) e da una seconda classe (**MetaMsgReceiptor**) che gli fornisce i metodi necessari per le operazioni di pack|unpack dei parametri.
Di seguito si riporta la classe Person in OpenC++, con la relativa classe Refl_Person (questa versione è priva, rispetto alla versione generata dal preprocessore, di diversi metodi inutili al fine del paragone che si intende effettuare).

Programma in OpenC++

```
% --- Person.oc ---

class Person {
public :
  Person( char *n, int a ) { name = n; age = a; };
  int Age() { return age; };
  //MOP reflect :
  int IncAge();
private :
  char *name;
  int age;
};

//MOP reflect class Person : VerboseMetaObj;

int Person::IncAge() { return ++age; }
```

Note a Margine

Il costruttore ed il distruttore della classe Refl_Person si occupano oltre che di creare e distruggere l'oggetto riflessivo anche di creare e distruggere il suo meta-oggetto.

I metodi del meta-oggetto Meta_Startup e Meta_Cleanup eseguono le operazioni di inizializzazione e di clean-up definite dal programmatore sul meta-oggetto.

Il metodo IncAge incapsula l'attivazione riflessiva del metodo IncAge della classe Person (Meta_MethodCall). Non ha categoria (category_none) perché non ne è stata definita alcuna e è identificato dal numero 3 perché questa è la sua posizione nell'array method_name.

Il metodo Meta_MethodCall attiva il metodo del meta-oggetto Meta_HandleMethodCall che attiva nel livello base il metodo Base_HandleMethodCall che attiva effettivamente il metodo IncAge della classe Person.

Risultato del preprocessore

```
% --- Person.cc ---

class Refl_Person : public Person, private MetaMsgReceiptor {
public :
  Refl_Person( char *, int );
  ~Refl_Person();
  int IncAge();
private :
  void Base_HandleMethodCall( Id, ArgPac &, ArgPac & );
  VerboseMetaObj *metaobj;
  static const char *method_names[];
};

Refl_Person::Refl_Person( char *p1, int p2 ) : Person(p1, p2) {
  metaobj = new VerboseMetaObj();
  metaobj->Meta_StartUp();
}

Refl_Person::~Refl_Person() {
  metaobj->Meta_CleanUp();
  delete metaobj;
}

int Refl_Person::IncAge() {
  ArgPac args, reply;
  metaobj->Meta_MethodCall( 3, Category_none, args, reply );
  return( reply.PopInt() );
}

void Base_HandleMethodCall( Id method, ArgPac &args, ArgPac &reply ) {
  switch( method ) {
    case 1 : Meta_refl_Init( args ); break;
    case 2 : reply.BeginSettingArgs();
             Meta_refl_Copy( reply );
             break;
    case 3 : reply.BeginSettingArgs();
             reply.PushInt( Person::IncAge() );
  }
}

const char *Refl_Person::method_names[] = {
  "Person", "Meta_refl_Init", "Meta_refl_Copy", "IncAge", nil };
```

Come si può osservare, l'espansione effettuata dal preprocessore impone delle limitazioni, come:

- l'impossibilità di scegliere un meta-oggetto da associare ad un oggetto riflessivo, che deriva dalla possibilità di esprimere preferenze unicamente sulla classe del meta-oggetto e non sui meta-oggetti.

- l'impossibilità di cambiare la classe del meta-oggetto variando così a run-time il comportamento riflessivo dell'oggetto riflessivo.

- l'impossibilità di cambiare a run-time il meta-oggetto che controlla un oggetto riflessivo.

Il meccanismo presentato nel capitolo [2], invece, permette di scegliere e successivamente cambiare il comportamento riflessivo tramite il metodo set_meta, permettendo di sostituire il meta-oggetto anche con meta-oggetti istanze di classi diverse, purché tutte discendenti dalla classe MetaObj.
Ovviamente in OpenC++ ogni meta-oggetto è legato ad un solo oggetto riflessivo e viceversa, mentre con il meccanismo send-HandleMessage-apply un meta-oggetto può essere associato a più oggetti riflessivi, potendo ad esempio svolgere il ruolo di coordinatore di un gruppo di oggetti. Ogni oggetto che usi il meccanismo send-HandleMessage-apply può variare dinamicamente il meta-oggetto associato, permettendo una maggiore flessibiltà nel comportamento riflessivo.

Anche OpenC++ fornisce la possibilità di costruire una gerarchia di classi che definiscono i possibili meta-oggetti. La presenza delle categorie assegnate ai metodi da dover attivare riflessivamente è solo un paravento utilizzato per mascherare il problema dell'oggetto factotum (visto nella sezione [1.8]); in realtà il metodo Meta_MethodCall ha comunque la funzione di metodo factotum, come del resto avviene nella definizione del metodo HandleMessage (questo è, infatti, il principale difetto del modello a meta-oggetto che non è possibile eliminare qualsiasi strategia implementativa si scelga).

Il linguaggio di Chiba presenta limitate possibilità di effettuare riflessioni strutturali, in particolare permette solo di sapere il nome del metodo attivato o della variabile acceduta. Questo è dovuto essenzialmente al fatto (come si può osservare nell'esempio riportato sopra) che la classe generata dal preprocessore ridefinisce i metodi riflessivi in modo che attivino il metodo del meta-oggetto Meta_MethodCall. Quest'ultimo metodo a sua volta attiva il metodo dell'oggetto riflessivo Base_HandleMethodCall, il quale, in base all'identificatore del metodo, attiva il metodo della classe di partenza. Una modifica dinamica ai metodi implicherebbe una modifica al metodo Base_HandleMethodCall che però è creato in fase di pre-processing, quindi non modificabile. Invece il meccanismo presentato nel capitolo [2] si basa sullo scambio dinamico di puntatori ai metodi, per cui una semplice modifica di questi puntatori sortisce l'effetto desiderato.

In OpenC++, analogamente a quanto scritto nella sezione [2.4], è possibile monitorare la creazione di un oggetto riflessivo. Inoltre è possibile monitorare anche gli accessi alle variabili, caratteristica non considerata finora, ma implementabile in modo analogo al controllo sui metodi. Questa caratteristica non è stata considerata nel capitolo [2] per non appesantire la presentazione, infatti, non essendo stata ipotizzata la presenza di un preprocessore, si sarebbe dovuto abbandonare l'uso dell'operatore di assegnazione, sostituendolo con una notazione funzionale. Inoltre non avrebbe aumentato la potenza espressiva del meccanismo.

La riflessione, si manifesta con uno switch del flusso computazionale dal livello base al meta-livello e viceversa. Nel meccanismo da noi studiato queste azioni di shift-up e shift-down del flusso computazionale sono effettuate tramite tre metodi:

- **send**: che si occupa di effettuare l'operazione di shift-up.
- **HandleMessage**: che effettua la computazione al meta-livello.
- **apply**: che effettua l'operazione di shift-down.

OpenC++, ovviamente, si basa sullo stesso principio di shift, solamente che i metodi che eseguono tale operazione, apparentemente, sono due:

- **Meta_MethodCall**: che praticamente svolge il ruolo svolto dal metodo send e dal metodo HandleMessage.
- **Meta_HandleMethodCall**: che svolge il ruolo svolto dal metodo apply.

Si è detto apparentemente, innanzitutto, perché è più sensato parlare di famiglie di metodi le cui componenti hanno ruolo analogo. Ad esempio alla prima famiglia appartengono i metodi Meta_MethodCall, Meta_Assign, Meta_Read, Meta_Create, ... che hanno tutti lo stesso ruolo (shift-up ed esecuzione riflessiva); l'unica differenza è il contesto in cui vengono attivati (attivazione di un metodo, assegnazione e lettura del contenuto di un attributo, creazione di un oggetto riflessivo, ...). Inoltre questi metodi sono ridefinibili dall'utente in modo da definire nuovi comportamenti riflessivi (come accade nel caso di HandleMessage).
Alla seconda famiglia appartengono metodi come Meta_HandleMethodCall, Meta_HandleAssign, Meta_HandleRead, Meta_HandleCreate, ... che effettuano lo shift-down.
La situazione è ulteriormente complicata dal fatto che ogni metodo è in realtà composto da due metodi, uno appartenente alla meta-entità ed uno appartenente all'entità del livello base. Ad esempio Meta_HandleMethodCall appartiene alla meta-entità ed attiva il metodo Base_HandleMethodCall che appartiene all'entità del livello base.
Quindi i due meccanismi, dinamicamente, risultano avere comportamento analogo. OpenC++ gestisce una casistica più varia, rispetto al meccanismo send-HandleMessage-apply, che permette di effettuare computazioni riflessive (come riflessione sugli accessi alle varibili). Comunque il meccanismo send-HandleMessage-apply, come detto, può essere adeguato a gestire tali casi.

L'ultima osservazione riguarda il ruolo giocato dai due meccanismi presentati nei confronti della concorrenza.

Il presente lavoro intende integrare un linguaggio di programmazione concorrente ad oggetti con il meccanismo presentato per gestire la riflessione, mentre OpenC++ vede la concorrenza come una proprietà implementabile tramite la riflessione. Quindi i due approcci alla concorrenza sono completamente diversi. Si sottolinea che comunque l'approccio alla concorrenza adottato da Chiba può essere implementato anche con il meccanismo presentato nel capitolo [2], ma non è quanto ci si è prefissi.

Entrambe le soluzioni hanno dei pregi e dei difetti; nel seguito quando si esaminano le applicazioni alla fault tolerance della riflessione nel caso del modello a meta-oggetto si usa il linguaggio OpenC++.

3.6 Modello a Materializzazione del Canale e PVM

In questa sezione viene presentata la metodologia necessaria per implementare il modello a materializzazione del canale in ambiente concorrente. Si è scelto di utilizzare PVM [PVM3 94, Manchek 94] in quanto è uno degli strumenti più diffusi ed utilizzabile su un gran numero di piattaforme hardware.
Va comunque sottolineato che i risultati ottenuti non dipendono dallo strumento usato per gestire la concorrenza, infatti si tratta di un meccanismo generale, e come tale facilmente adeguabile ad ogni altro supporto implementativo della concorrenza basato sul linguaggio C++.

3.6.1 Introduzione a PVM

PVM Parallel Virtual Machine (Macchina Parallela Virtuale) è un sistema software che permette di virtualizzare un'architettura parallela astraendo dalla sottostante architettura. PVM permette, ad esempio, ad una rete di workstation UNIX eterogenea di essere usata come un unico computer parallelo.
Procedendo in questo modo una vasta classe di problemi computazionali può essere risolta sfruttando il potere aggregato di molti computer.
Le caratteristiche di maggiore interesse del PVM sono la *macchina virtuale*, i *tid* ed il modello di scambio dei messaggi tra i processi. Altra caratteristica interessante di PVM, per il cui approfondimento si rimanda a [PVM3 94], è il supporto ai gruppi di processi.

3.6.1.1 La Macchina Virtuale di PVM

PVM si basa sulla presenza di un demone, chiamato *pvmd*, su ogni host reale. L'insieme dei demoni viene detto macchina virtuale.
Il demone di PVM serve per scambiare i messaggi e per controllare lo stato dei task interni alla macchina virtuale. Esso fornisce un punto di contatto su ogni host, sia per intra-comunicazioni (cioè comunicazioni fra processi residenti sullo stesso host),

che per inter-comunicazioni (cioè comunicazioni tra processi residenti su host diversi), controllo sui processi e riconoscimento del fallimento sia degli host che dei task. I demoni verificano periodicamente la raggiungibilità degli altri demoni. Se questi risultano irraggiungibili, vengono marcati come "morti".
Il *pvmd* della macchina che fa partire il calcolo è detto *master*, mentre gli altri sono detti *slave*. Solo il master può aggiungere o togliere dei *pvmd* alla macchina virtuale, quindi qualsiasi richiesta di riconfigurare la macchina virtuale fatta su un host, il cui demone è un *pvmd* slave, viene inviata al master. Se il master perde contatto con uno slave, lo marca come morto e lo elimina dalla macchina virtuale. Se è lo slave a perdere contatto con il master questo termina ogni sua operazione ed esce. Questo approccio garantisce che la macchina virtuale non possa essere partizionata e continuare a lavorare come due macchine parallele distinte, comunque in questa versione di PVM il master non deve fallire (quindi l'host su cui risiede il master deve essere considerato sicuro).
Le strutture dati principali sono le tavole degli host e dei task, che descrivono la configurazione della macchina virtuale. Associate a queste tavole ci sono le code di messaggi inevasi dei vari task.

3.6.1.2 Identificatori dei Task

PVM usa un intero di 32 bit, detto identificatore dei task (TID, Task IDentifier), per indirizzare i demoni, i task ed i gruppi di task all'interno della macchina virtuale. Un tid identifica un oggetto unico entro l'intera macchina virtuale.
Data l'importanza fondamentale e l'uso pesante dei tid, questi sono progettati in modo da poter essere agevolmente usati dal maggior numero di macchine senza ulteriori codifiche; ciò è ottenibile sfruttando un tipo di dato, per contenere il tid, di una dimensione tale che per qualunque macchina esiste un tipo di dato con dimensione analoga. I tid sono suddivisi in quattro campi (S, G, H e L). I campi S, G e H hanno significato globale, cioè qualunque demone della macchina virtuale li interpreta nella stessa maniera.

- Il campo H contiene il numero dell'host relativo alla macchina virtuale; il mapping tra il numero e l'host è noto ad ogni demone pvmd.

- Il campo S in combinazione con il campo H è utilizzato per individuare i messaggi inviati al demone dell'host identificato dal campo H . In tal caso il campo L ha valore zero.

- Il campo G è usato per formare indirizzi multicast (GID), che si riferiscono ai gruppi di task.

Ogni demone PVM assegna significato locale al campo L, riservando il valore zero per il demone stesso, e rappresenta il numero di un processo PVM residente sulla macchina determinata dal campo H. La caratteristica di località del campo L permette al proprio *pvmd* di assegnare un tid al task senza dovere comunicare con il master, eliminando così un possibile collo di bottiglia nelle prestazioni del sistema. Questa è la tecnica utilizzata da PVM per localizzare i processi all'interno della macchina virtuale, tale meccanismo è assimilabile all'approccio di localizzazione degli oggetti tramite la codifica della loro posizione nel sistema all'interno dell'identificatore d'oggetto presentato nella sezione [3.3.5]. Concettualmente il problema di localizzare un processo è analogo a localizzare un oggetto attivo e come tale anche le soluzioni sono simili.

3.6.1.3 Il modello dei messaggi adottato da PVM

I demoni di PVM ed i task possono costruire ed inviare messaggi di lunghezza arbitraria contenente dati tipati. La macchina virtuale di PVM può contenere host eterogenei, per cui può essere necessario, in caso di incompattibilità dei formati, convertire i dati in uno standard riconoscibile da ogni macchina, lo standard adottato da PVM è XDR [Sun 87]. I messaggi sono etichettati al momento dell'invio con un codice intero (etichetta) definito dall'utente e possono essere selezionati per la ricezione tramite il tid del mittente e|o l'etichetta.
L'azione di invio di un messaggio è di tipo asincrono o non bloccante, cioè il mittente di un messaggio non attende un segnale di ricezione (acknowledge) da parte del destinatario, ma prosegue la propria esecuzione non appena il messaggio è gestito dalla rete ed il buffer del messaggio può essere riutilizzato in modo sicuro. I messaggi sono bufferizzati dalla rete nei punti di ricezione finché il destinatario non è in grado di processarli effettivamente.
PVM offre la possibilità di gestire la ricezione dei messaggi sia in modo sincrono o bloccante che asincrono o non bloccante. Nel caso bloccante (ad esempio usando la primitiva *pvm_recv*) il destinatario si pone in attesa di un messaggio finché non lo riceve effettivamente. Nel caso non bloccante (ad esempio usando la primitiva *pvm_nrecv*) il destinatario verifica se ha ricevuto un messaggio senza fermarsi ad attenderlo.
PVM è progettato in modo da garantire la consegna dei messaggi secondo l'ordine di invio. Inoltre fornisce una versione non bloccante ed una bloccante delle primitive di ricezione, permettendo ai processi di effettuare il polling dei messaggi, usando il primo tipo di primitive, o di attendere il messaggio senza consumare tempo di CPU, usando il secondo tipo di primitive.
A queste primitive di comunicazione si aggiungono quelle necessarie per comunicare con i gruppi di processi. I membri di un gruppo possono essere raggiunti da un processo esterno al gruppo (comunicazione inter-gruppo) singolarmente, tramite le primitive già descritte, o tutti insieme, tramite una comunicazione in broadcast (usando la primitiva *pvm_bcast*). Le comunicazioni intra-gruppo vengono effettuate con le stesse primitive che permettono la comunicazione tra qualsiasi processo.

3.6.2 Costruzione degli Oggetti Attivi

PVM è progettato per interfacciarsi con i linguaggi Fortran e C. Essendo il C++ un'estensione ad oggetti del C, anche il C++ può essere interfacciato con PVM [10].
Non essendo il C un linguaggio ad oggetti, il supporto di PVM per la concorrenza è limitato alla gestione dei processi ed alle loro comunicazioni e non alla gestione degli oggetti attivi; in questo ambiente l'unità di concorrenza è rappresentata dal processo. Quindi in attesa di una libreria PVM sviluppata appositamente per la gestione degli oggetti attivi, occorre, per ottenere un'applicazione concorrente con oggetti cooperanti, implementare, innanzitutto, gli oggetti attivi.
Il primo passo da intraprendere nella costruzione degli oggetti attivi è quello di scegliere su quale modello degli oggetti basarsi (*passivo* o *attivo* per i dettagli vedere la sezione [3.3.1]).
Si è scelto di associare un processo ad ogni oggetto come accade nel *modello attivo degli oggetti*. Il processo ha il compito di gestire le richieste di computazione che giungono all'oggetto attivo.
Siccome PVM (nella versione 3.3 attualmente in distribuzione) non supporta i thread, un oggetto attivo può gestire solamente una richiesta per volta. PVM, come già detto, nel caso pervengano più messaggi per un certo processo, li inserisce in una coda di messaggi inevasi, in attesa che vengano evasi. Tale comportamento rende inutile la presenza di più processi gestori per ogni oggetto, in quanto comunque si limiterebbero ad effettuare una bufferizzazione in attesa della disponibilità dell'oggetto, operazione svolta automaticamente da PVM.
Il secondo passo da intraprendere è quello di determinare cosa sono le richieste e come vengono gestite da parte dei processi gestori. Essendo PVM un ambiente basato sullo scambio dei messaggi, le richieste hanno ovviamente forma di messaggi. I servizi|metodi richiesti ad un oggetto sono determinabili tramite le etichette associate ai messaggi che li richiedono. Si definiscono, quindi, per ogni oggetto attivo un insieme di etichette di messaggi, uno per ogni metodo dell'oggetto che il processo gestore deve essere in grado di gestire. Il corpo del messaggio contiene le informazioni utili all'attivazione del metodo, come i parametri attuali.
Il processo gestore ricevuto un messaggio sa dall'etichetta quale metodo attivare, mentre dal corpo del messaggio ottiene i parametri attuali con i quali attivare il metodo. Eseguito il metodo, l'oggetto si occupa di restituire un messaggio etichettato come "risposta" al mittente, contenente il valore di ritorno. In questo modo si ottiene una specie di remote procedure call [Birrel-Nelson 84].
Vediamo di chiarire su un esempio; si supponga di avere un oggetto $\mathcal{A}$ che offre il

[10] Esiste un tentativo di interfacciare PVM con il C++ chiamato PVM++, che è costituito da una libreria di classi che si limitano ad incapsulare le funzioni di PVM senza offrire la possibilità di gestire oggetti attivi e l'invocazione remota dei metodi. L'uso di PVM++ in luogo di PVM non offre alcun vantaggio per il nostro obiettivo, per cui si è preferito utilizzare direttamente PVM che è sicuramente più diffuso e soprattutto più stabile.

servizio x, il quale richiede un intero. Un oggetto cliente $\mathcal{B}$ per richiedere ad $\mathcal{A}$ il servizio x deve inviare ad $\mathcal{A}$ un messaggio, contenente l'intero con cui attivare il metodo x, etichettato da "X". $\mathcal{A}$, elaborato il servizio, ritorna un messaggio a $\mathcal{B}$ contenente il risultato del servizio.
Si osservi che nel caso di oggetti attivi, che sono anche oggetti riflessivi, questi devono essere in grado di gestire messaggi che richiedono l'attivazione dei metodi send e apply. I metodi send e apply di un oggetto riflessivo sono i metodi più usati e comunque vengono gestiti dal processo gestore in modo analogo agli altri. Nel caso di un oggetto canale, che è anch'esso un oggetto attivo, il processo gestore riceve richieste per la gestione del metodo HandleMessage.
Ricapitolando, le classi vengono costruite secondo gli usuali principi del paradigma, ad ogni istanza delle varie classi viene associato un processo che crea e gestisce l'oggetto, e la fase di gestione è rappresentata da un ciclo che verifica l'arrivo di un messaggio, lo interpreta, esegue la richiesta e quindi restituisce un messaggio contenente i risultati al mittente. Il processo gestore segue lo schema riportato di seguito.

```
main( int argc, char **argv ) {
  int bufid, msg_tag, sender_tid, bytes;
  <type_object> obj ( argv[0], ..., argv[argc-1]); // creazione dell'oggetto
  while( TRUE ) {
    bufid = pvm_recv( any_proc, any_tag ); // any_proc = any_tag = -1 (costanti)
    pvm_bufinfo( bufid, &bytes, &msg_tag, &sender_tid );
    switch( msg_tag ) {
      case <etichetta1 >:
        // spacchetta tid del mittente
        // spacchetta parametri
        // eseguo richiesta con i parametri
        // impacchetta parametri
        // impacchetta valore di ritorno
        pvm_send( sender_tid, REPLY );
      case <etichetta2 >:
          ⋮
      default:
          ⋮
    }
  }
}
```

Questo processo gestore soddisfa le richieste con l'ordine con cui arrivano perché non attende uno specifico messaggio (la primitiva PVM *pvm_rec* è di tipo bloccante ed attende un messaggio qualsiasi (*any_tag*), proveniente da qualsiasi oggetto attivo (*any_proc*)).
Oggetti istanza della stessa classe hanno processi gestori con comportamento identico; l'unica differenza riguarda i parametri di attivazione che caratterizzano l'oggetto che gestiscono.

3.6.3 Approccio Concorrente e Modello a Materializzazione del Canale

In ambiente concorrente un'applicazione object-oriented è il risultato della cooperazione tra oggetti attivi. Un canale, in ambiente concorrente, è instaurato tra due oggetti attivi, analogamente il canale stesso è un oggetto attivo.
Perciò la struttura delle applicazioni impone delle modifiche ai parametri dei metodi usati per la riflessione (send, HandleMessage e apply). Questi metodi richiedono la conoscenza delle entità che coinvolgono. Tale conoscenza, in ambiente sequenziale, è fornita loro tramite il passaggio del riferimento a tali entità (indirizzo). L'indirizzo di un oggetto in ambiente distribuito perde di significato. L'indirizzo di una entità è significativo solo nel contesto della macchina su cui risiede, nel momento in cui si opera su più macchine tale informazione perde significato.
Per reperire l'entità coinvolta si deve ricorrere ad una delle strategie presentate nel paragrafo [3.3.5].
PVM usa degli interi detti identificatori dei task (tid) per localizzare un processo all'interno della macchina virtuale (strategia assimilabile al primo approccio presentato nella sezione [3.3.5]). Essendo associato un unico oggetto ad ogni processo gestore, il tid del processo gestore può essere usato anche per localizzare l'oggetto attivo corrispondente e quindi l'oggetto stesso.
Quindi le informazioni utili ai suddetti metodi sono i tid degli oggetti che coinvolgono. Analogamente anche il dizionario dei canali non mantiene più i riferimenti agli oggetti (sender, receiver e canale) bensì i tid degli oggetti attivi corrispondenti.

3.6.4 Remote Procedure Call e Funzioni di Pack|Unpack

Ogni richiesta ad un oggetto attivo è effettuata tramite un messaggio. Le informazioni che compongono il messaggio, per ragioni di efficienza e per superare più agevolmente le differenze tra le varie architetture collegate al sistema, vengono codificate ed impacchettate[11] in un buffer. PVM fornisce parecchie funzioni dedicate alla gestione dei messaggi, limitandosi però a gestire informazioni di tipo scalare (interi, reali, caratteri, stringhe, ...) od array di scalari, lasciando all'utente la progettazione di funzioni appositamente pensate per l'impacchettamento e lo spacchettamento di dati strutturati (oggetti, strutture e funzioni). Rifacendosi a quanto detto nel capitolo [2], i metodi che implementano la riflessione si scambiano, tra l'altro, il metodo da attivare ed un oggetto di tipo message (da non confondere con i messaggi di PVM), contenente gli argomenti attuali con cui attivare il metodo. In ambiente concorrente le informazioni da gestire sono sempre le stesse; quindi, per poter usare adeguatamente lo scambio di messaggi proprio di PVM, si devono costruire

[11] Nel seguito, per non appesantire la trattazione, si tende a confondere in un'unica fase, la codifica e l'impacchettamento, usando i due termini in modo interscambiabile.

le funzioni necessarie per la codifica|decodifica di queste informazioni all'interno dei messaggi di PVM.

3.6.4.1 Pack e Unpack dei Metodi

```
% --- primitive di pack e unpack ---

  int pvm_pkmeth( void *(*enc)( reflobj *, int, void **) ) {
      // impacchetta nel buffer di invio la codifica
      // dell'incapsulatore; ritorna la codifica o -1 in caso di insuccesso
    for( int i=0; i<NENTRY; i++ )
      if ( *(enc_table+i) == enc ) {
        pvm_pkint( &i, 1, 1 );
        return i;
      }
    return -1;
  }

  int pvm_upkmeth( void *(*enc)( reflobj *, int, void **) ) {
      // spacchetta dal buffer di arrivo la codifica
      // dell'incapsulatore; ritorna la codifica o -1 in caso di insuccesso
    int i;
    pvm_upkint( &i, 1, 1 );
    if ( i >= NENTRY ) return -1;
    enc = *(enc_table+i);
    return i;
  }

% --- tabella di codifica degli incapsulatori ---

  const int NENTRY = 2;
  void *(*enc_table[NENTRY])( reflobj *, int, void ** ) = { enc_incage, enc_printperson };
```

Codice PVM e C++ 3.1: Funzioni di Pack e Unpack per i metodi

In ambiente sequenziale, il problema del passaggio dei metodi come parametri è stato risolto usando gli incapsulatori (vedere la sezione [2.2.1]) e passando l'indirizzo di questi ultimi ai metodi.
In ambiente concorrente, si deve costruire una funzione di codifica per i metodi tale che la codifica effettuata da un processo sia decodificabile da un altro processo in modo da ricostruire l'attivazione dello stesso metodo codificato.
Si continuano ad utilizzare gli incapsulatori per le ragioni di generalità già spiegate in precedenza. Si consideri come funzione di codifica, quella funzione che codifica l'indirizzo come una sequenza di byte interpretabili dal processo ricevente. Questa è una soluzione semplice e diretta, però il problema risiede proprio nella natura concorrente del sistema in cui la stessa funzione, ha in processi diversi indirizzo diverso. Questo è un problema assimilabile ai tipici problemi di localizzazione di oggetti attivi in un sistema ad oggetti concorrente (vedere sezione [3.3.5]) o di un file in un file system distribuito (vedere [Needham 89]). Come tale è possibile reperirne in

letteratura delle soluzioni adattabili al nostro caso.
Per gli scopi di questa sezione non è necessario utilizzare meccanismi particolarmente efficienti, ma al contempo complessi, soprattutto quando è possibile, con alcune restrizioni, ritagliarsi una soluzione su misura e molto semplice da realizzare.
Supposto che le applicazioni che si vogliono realizzare non abbiano la possibilità, una volta in esecuzione, di aggiungere oggetti attivi che offrono servizi (metodi) non previsti, i metodi sono un numero fissato e determinabile al momento della compilazione[12].
È possibile, quindi, costruire una tabella di codifica|decodifica dei metodi, basandosi sui servizi (incapsulatori). Le routine di pack e unpack si basano su tale tabella. La tabella, che è contenuta in un apposito *header*-file, contiene per ogni incapsulatore un codice che è il risultato della funzione di codifica, mentre il corrispondente incapsulatore è il risultato della funzione di decodifica. Questa soluzione risolve il problema precedente perché ogni processo ha la propria tabella, inclusa durante la compilazione, avente quindi gli indirizzi degli incapsulatori locali al processo e la codifica numerica dell'incapsulatore comune ad ogni processo. Ogni item della tabella, che può essere strutturata come un array, ha il seguente tipo:

```
void *(**enc_table)( reflobj *, int, void ** )
```

va rimarcato che una struttura di questo tipo è possibile perché gli incapsulatori hanno tutti lo stesso tipo unificato a:

```
void *(*enc)( reflobj *, int, void ** )
```

La codifica numerica dell'incapsulatore è rappresentata dall'indice dell'entry della tabella.
Data la semplicità dei contenuti della tabella, è possibile lasciarne la compilazione al programmatore oppure si può costruire una semplice utility che cerca gli incapsulatori all'interno dei file che compongono l'applicazione e costruisce, quindi, la tabella e relativo header file.
Nel riquadro [3.1] è riportato il codice delle funzioni di pack e unpack per i metodi, più un esempio di inizializzazione della tabella relativamente alla classe `person` presentata nel riquadro [2.5].

3.6.4.2 Pack e Unpack di oggetti istanza della classe message

Sorge spontaneo interrogarsi sull'effettiva utilità della costruzione di un oggetto della classe `Message` per contenere i parametri del metodo da attivare. In ambito sequenziale tale messaggio viene costruito una sola volta, al momento della richiesta di una

[12] In realtà risulta fisso il numero di servizi diversi offerti e non il numero dei metodi distinti (oggetti attivi istanza della stessa classe hanno metodi che non sono confrontabili, anche se offrono lo stesso servizio), i servizi sono associati agli incapsulatori.

```
int pvm_pkvoidp( void **args, int n_args, int stride ) {
    // impacchetta nel buffer di invio la codifica di un array
    // di puntatori a void; ritorna il numero di elementi impacchettati
  int size;
  for( int i=0; i< n_args; i++ ) {
    size = sizeof( *(*(args+i)) );
    pvm_pkint( &size, 1, stride );
    pvm_pkbyte( *(*(args+i)), size, stride );
  }
  return i+1;
}

int pvm_upkvoidp( void **args, int n_args, int n_args, int stride ) {
    // spacchetta dal buffer di arrivo la codifica di un array
    // di puntatori a void; ritorna il numero di elementi spacchettati
  int size; char *aux;
  for( int i=0; i< n_args; i++ ) {
    pvm_upkint( &size, 1, stride );
    aux = new char[size];
    pvm_upkbyte( aux, size, stride );
    *args[i] = (void *)aux;
  }
  return i;
}
```

Codice PVM e C++ 3.2: Funzioni di Pack e Unpack per il tipo void *

computazione riflessiva, e viene, quindi, passato alle varie routine che lo necessitano. In ambiente concorrente questo oggetto viene costruito ogniqualvolta viene inserito all'interno di un messaggio scambiato nel sistema (questa costruzione è implicita nella routine di decodifica) e quindi incrementa l'overhead dovuto alle comunicazioni. Per ridurre tale overhead non si usa più la classe Message, usando in luogo delle istanze di Message, direttamente le varie componenti e modificando di conseguenza anche i metodi send, HandleMessage e apply. Ovviamente risulta inutile costruire delle primitive per la codifica|decodifica di questo tipo di dato.

Anche se non si usa la classe Message, è, comunque, necessario impacchettare i valori che sarebbero stati contenuti dalle istanze di questa classe (numero degli argomenti ed argomenti dei metodi riflessivi); particolare attenzione va posta per gli argomenti del metodo, questi per poter unificare il tipo degli incapsulatori sono contenuti in un array di puntatori a void. Il problema è lo stesso che si è incontrato con i metodi nella sezione precedente. La primitiva che effettua l'azione di pack deve, quindi, codificare il contenuto di tali locazioni di memoria e non le locazioni stesse in qualcosa di non tipato come ad esempio una sequenza di byte. Le routine di codifica e decodifica di elementi del tipo byte sono previste da PVM. Rimane ancora il problema che non tutti i valori contenuti nelle locazioni di memoria puntate da elementi di tipo void sono rappresentabili con lo stesso numero di byte (determinabili con la macro *sizeof*) quindi per una corretta decodifica si deve, per ogni elemento, impacchettare anche la sua dimensione[13]. Nel riquadro [3.2] è riportato il codice delle primitive di pack|unpack per il tipo *void **.

3.6.5 Dizionario dei Canali Distribuito

Struttura fondamentale del modello a materializzazione del canale è il dizionario dei canali, questo come si è detto nelle sezioni [1.8] e [2.6], mantiene i riferimenti ai canali già attivati in modo da facilitare il loro riuso all'interno del sistema.
In ambiente sequenziale questa entità è rappresentata da una struttura condivisa da tutti gli oggetti riflessivi e viene interrogata, al momento di una richiesta riflessiva, alla ricerca di un canale adatto a soddisfare il comportamento riflessivo richiesto.
In ambiente concorrente, una modifica, dovuta al problema della localizzazione degli oggetti attivi, si è già resa necessaria e concerne il mantenimento dei riferimenti che non sono più dei puntatori agli oggetti, ma i tid della corrispondente entità[14]. Di seguito si presenta l'evolvere di questa struttura per adattarsi all'ambiente concorrente.

3.6.5.1 Dizionario dei Canali come Oggetto Attivo

In ambiente sequenziale, il dizionario dei canali è una struttura dati comune ad ogni oggetto riflessivo. Trasporre tale implementazione in ambiente concorrente, dove ogni oggetto riflessivo diventa un oggetto attivo (che come visto è sostanzialmente un processo), significa restringere i tipi di archittetture hardware in cui poter implementare un sistema riflessivo. Infatti in ambiente concorrente, per avere un'efficiente condivisione di una struttura dati (il dizionario) a diversi processi (gli oggetti riflessivi), occorrerebbe avere a disposizione delle aree di memoria condivise (limitando l'implementazione solamente ai sistemi concorrenti con memoria condivisa).
Rendendo il dizionario un oggetto attivo, la sua funzionalità viene leggermente modificata; infatti non si limita solo a gestire il ritrovamento del canale, ma si occupa anche di crearlo e di metterlo in contatto con il sender ed il receiver, compito precedentemente lasciato al costruttore del canale, quindi <u>le richieste di attività riflessiva passano tutte attraverso il dizionario dei canali</u>.

3.6.5.2 Decentralizzazione del Dizionario dei Canali

Basandosi sull'osservazione effettuata nella precedente sezione, risulta che questa strategia è affetta da un grave difetto rappresentato dal fatto che il dizionario dei

[13] In queste routine, per non appesantirne la trattazione, non è stato considerato il fatto che processori diversi, possono adottare ordinamenti dei byte diversi (little, middle o big endian), è comunque semplice risolvere tale problema sfruttando, ad esempio, i concetti e le routine illustrate in [Rogers 95].

[14] Anche se non è stato detto esplicitamente questa modifica coinvolge anche i canali, i quali non mantengono più il riferimento ai sender ed ai receiver, ma bensì i loro tid.

canali viene interrogato da tutti gli oggetti riflessivi, ovunque si trovino all'interno della macchina virtuale. Questa situazione, oltre a presentare un notevole carico di lavoro per il dizionario dei canali e per la macchina su cui risiede, rappresenta anche un collo di bottiglia nelle prestazioni del sistema (troppe comunicazioni dirette allo stesso processore) e presenta anche dei problemi logistici, come con quale criterio scegliere il processore su cui far risiedere il dizionario. Non ultimo, in caso di fault tolerant può divenire un tallone di Achille del sistema.
Dato che le funzionalità del dizionario sono essenziali non è possibile eliminarlo, ma si può partizionarlo, distribuendo il carico tra diversi oggetti.
Continuando a mantenere per il dizionario dei canali la struttura presentata nella sezione [2.6], si può osservare che le richieste riflessive vengono sempre effettuate dai sender, quindi è possibile partizionare il dizionario dei canali in tanti sotto-dizionari, uno per ogni host della macchina virtuale, contenente i riferimenti ai canali instaurati con sender siti sull'host su cui risiede il dizionario. Questa strategia limita le richieste di azioni riflessive ad un dizionario da parte di oggetti attivi residenti sullo stesso host; tali richieste, inoltre, si esauriscono nella ricerca all'interno di una sola struttura, più piccola di quella originale[15].
Vediamo ora come vengono gestiti i servizi che ogni dizionario offre. Il processo gestore deve essere in grado di gestire le richieste di operazioni riflessive da parte degli oggetti riflessivi (ricezione del messaggio REFLECTIVE_REQUEST).
Le richieste di computazioni riflessive provvengono esclusivamente da sender residenti sullo stesso processore su cui risiede il sotto-dizionario interpellato. L'azione può essere suddivisa in sei fasi principali:

① Nella prima fase il processo gestore del dizionario spacchetta il messaggio ricevuto, contenente i tid del sender e del receiver, il genere del canale richiesto ed il metodo da attivare riflessivamente con i suoi parametri attuali.

② Nella seconda fase, in accordo con quanto affermato nella sezione [3.4.2.1], si richiede il locking del receiver (se questo è differente dal sender).

③ Nella terza fase si effettua l'interrogazione del dizionario alla ricerca di un canale che soddisfi i requisiti imposti, in caso che non esista tale canale esso viene creato (è un oggetto attivo) e lo si inserisce nel dizionario.

④ Nella quarta fase il dizionario richiede al canale recuperato|creato la gestione riflessiva del metodo richiesto dal sender (messaggio HANDLE_REQUEST).

⑤ Nella quinta fase il dizionario attende la risposta del canale alla sua richiesta.

⑥ Nella sesta fase il dizionario ricevuta la risposta del canale, la gira al sender ed al receiver, informandoli anche del tid del canale.

[15] In base a quest'ultima osservazione, come già detto nella sezione [2.6], si può adottare per il dizionario una struttura più semplice pur mantenendo buone prestazioni.

La seconda fase, come si può osservare ha il compito di sincronizzare le richieste riflessive dirette ad uno stesso oggetto attivo. Infatti le richieste riflessive non vengono soddisfatte finché l'oggetto attivo, che ricopre il ruolo di receiver, non ha terminato la computazione corrente. Quindi le richieste riflessive dirette allo stesso oggetto sono serializzate in base al loro ordine di arrivo.

Il processo gestore del dizionario dei canali può anche ricevere richieste di eliminazione esplicita od implicita di un canale, quando un sender od un receiver abortisce. La cancellazione esplicita può venire richiesta da qualsiasi entità e deve essere rivolta al dizionario residente sul processore su cui risiede il sender che è collegato dal canale che si vuole eliminare. Per il resto viene effettuata come in ambiente sequenziale.
Più complessa la cancellazione implicita; la necessità di tale operazione sorge al momento in cui un oggetto riflessivo cessa la propria attività. A differenza del caso esplicito in cui si sa quale canale si vuole cancellare ed in quale dizionario è inserito, i canali da eliminare sono diversi e non sono tutti situati nello stesso dizionario in quanto un'entità può svolgere sia il ruolo di sender che quello di receiver in una comunicazione. Si procede, quindi, nel seguente modo:

- Nel momento in cui un'entità cessa la propria attività viene comunicato al dizionario residente sullo stesso processore.

- Ricevuta la segnalazione, il dizionario provvede a cancellare tutti i canali di cui l'entità è sender (per come è strutturato il sistema, sono inseriti tutti nello stesso sotto-dizionario).

- Successivamente propaga la notizia agli altri dizionari (compreso se stesso), i quali provvedono a cancellare tutti i canali di cui l'entità è receiver.

Come si può notare, questa operazione richiede la possibilità che un dizionario comunichi con gli altri dizionari. Questo comporta la conoscenza da parte di ogni dizionario del tid di tutti gli altri dizionari, questo è difficile perché in ogni istante può venire creato un nuovo dizionario in seguito all'inserimento di un nuovo host alla macchina virtuale. Un buon metodo per ovviare a questo problema è quello di creare un gruppo SOO (DICTIONARY_GROUP) composto da tutti i dizionari e quindi effettuare una comunicazione intra-gruppo (broadcast agli altri elementi del gruppo) per comunicare la dipartita di un'entità riflessiva.
Chiariamo quanto affermato con un esempio; riprendiamo la situazione considerata nell'esempio di figura [2.2.a], le cui componenti, per comodità, sono, nuovamente, riportate nella tabella seguente:

sender	receiver	tipo	canale
1	4	T_d	d
2	1	T_b	b
2	5	T_a	a
3	1	T_c	c
3	4	T_e	e
4	5	T_f	f
5	1	T_g	g

Consideriamo l'equivalente situazione in ambiente concorrente, in cui si hanno a disposizione tre processori (α, β, γ) e gli oggetti attivi (1, 2, 3, 4 e 5) sono distribuiti su tali processori come mostrato nella seguente tabella:

host	oggetto attivo
α	1
	2
β	3
	4
γ	5

Supponiamo che l'oggetto attivo 1 termini la propria attività e lo segnali[16] al dizionario sull'host α (**dict_alpha** figura [3.7.a]). Il dizionario, ricevuto il messaggio, elimina tutti i canali interagenti con l'oggetto attivo 1 in qualità di sender (figura [3.7.b]) e quindi comunica agli altri dizionari (**dict_beta** e **dict_gamma**) ed a se stesso la dipartita di tale oggetto attivo (figura [3.7.c]). Gli altri dizionari eliminano dalla propria base di informazioni i canali interagenti con l'oggetto 1 in qualità di receiver (figura [3.7.d]).

<u>Nota</u> Anche se non è stato detto, è ovvio che non è sufficiente cancellare il riferimento dal dizionario per eliminare il canale, come invece avviene nel caso sequenziale (si ricorda che un oggetto di cui si perde il riferimento viene "cestinato" dal garbage-collector del sistema). Si deve invece segnalare al canale di terminare le proprie computazioni e quindi uscire per eliminare effettivamente il processo; tale azione può essere realizzata tramite una funzione **kill_process** che viene passata alle routine di cancellazione dei canali dei dizionari che provvedono ad attivarla. Di seguito è riportata una possibile implementazione della funzione **kill_process**.

```
void kill_process( tid ch )
{
  pvm_initsend( PvmDataRaw );
```

[16] La segnalazione può essere effettuata, in maniera poco trasparente, tramite l'invio di un messaggio al dizionario, od in maniera più trasparente usando la primitiva PVM *pvm_notify*, che obbliga il demone di PVM a segnalare automaticamente ad un task prescelto (nel nostro caso il dizionario) la dipartita di un processo.

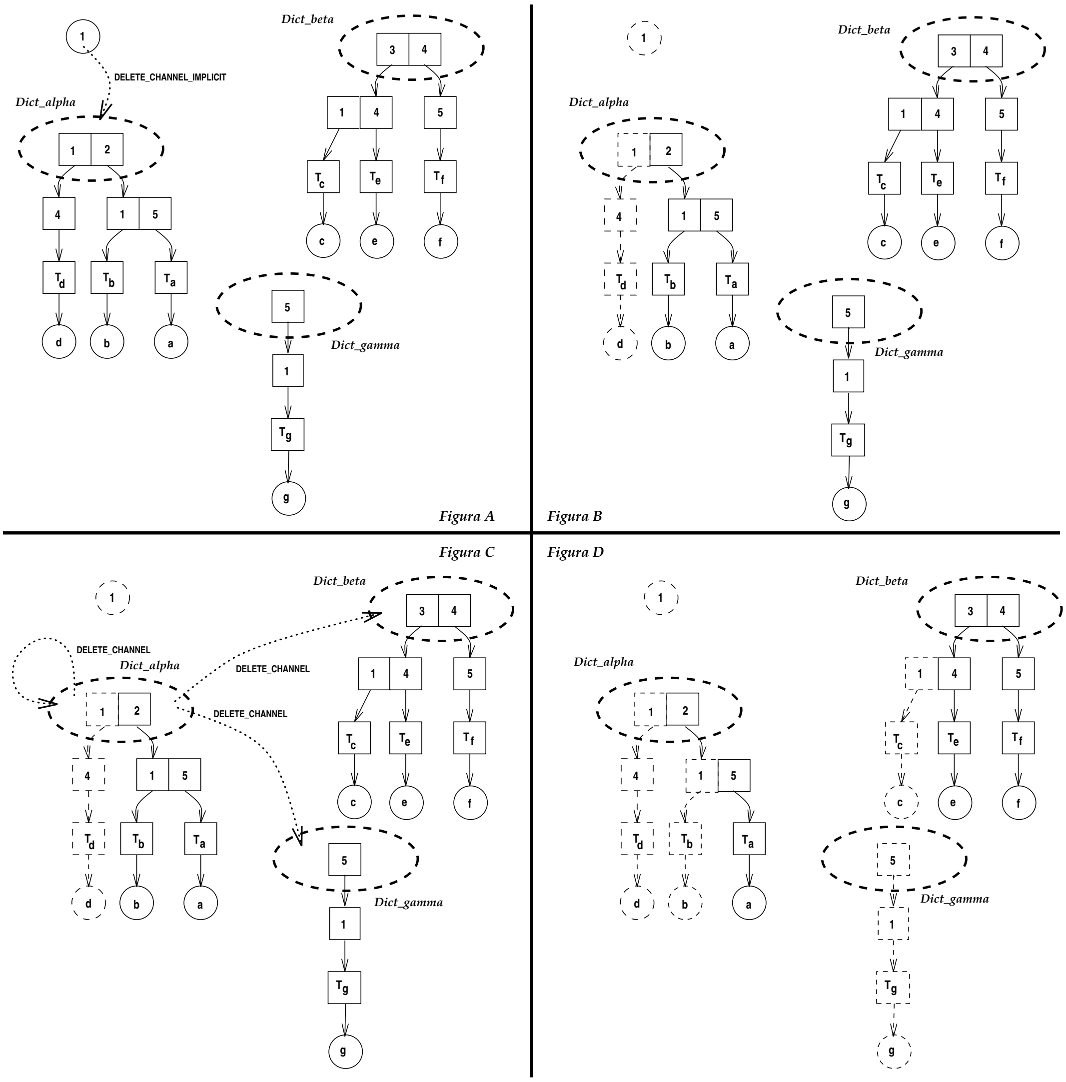


Figura 3.7: **Azione di cancellazione implicita**

```
    pvm_pkstr( ''DUMMY'' );
    pvm_send( ch, KILL_CHANNEL );
}
```

L'applicazione è composta da un sistema in evoluzione in cui gli oggetti attivi possono venire creati e distrutti continuamente.
La distruzione degli oggetti unita alla distribuzione del dizionario dei canali può portare a delle inconsistenze tra i dizionari ed il sistema che rappresentano, cioè un dizionario, causa ritardi di gestione, può mantenere riferimenti a canali che sono

stati cancellati od instaurati tra entità riflessive che non esistono più. Ad esempio si consideri un oggetto $\mathcal{A}$ che viene distrutto, questo segnala al proprio dizionario la decisione, il quale provvede ad eliminare tutti i canali di cui l'oggetto è sender ed a diramare la notizia agli altri dizionari, se uno di questi sta evadendo altre richieste posporrà l'aggiornamento delle proprie informazioni mantenendo una vista anacronistica del sistema.
In tale situazione può accadere che vengano richieste computazioni riflessive ad un receiver che non esiste più e che invece il dizionario, che gestisce la richiesta, abbia ancora canali instaurati con esso, portando ad errori di inconsistenza del sistema.
Un simile rischio può essere evitato effettuando una fase di locking accurata, come quella rappresentata dal seguente stub di codice:

```
struct timeval tmount = { 5, 5 };

while ( TRUE ) {
  if ( pvm_pstat( receiver_tid ) != PvmOk ) {
    reply = -1;
    pvm_initsend( PvmDataRaw );
    pvm_pkint( &reply, 1, 1 );
    pvm_send( sender_tid, REPLY );
    break;
  }
  if ( pvm_trecv( receiver_tid, ACKNOWLEDGE, tmout ) > 0 ) break;
}
if ( reply == -1 ) break;
```

Questa porzione di codice controlla, dopo aver mandato la richiesta di locking, se il receiver è attivo ed in caso affermativo attende con timeout inizializzato ad un valore prestabilito (ad esempio cinque secondi) il ritorno di un messaggio di acknowledge; dopo questo periodo di tempo se non ha ricevuto l'acknowledge ripete il ciclo finché non si verifica una delle due situazioni seguenti: il receiver non esiste più, allora il dizionario sblocca il sender con un messaggio di fallimento, oppure riceve l'acknowledge allora il receiver viene bloccato in attesa di risposta dal dizionario.
Facciamo alcune considerazioni di efficienza: l'operazione più costosa è la comunicazione tra oggetti (soprattutto quando risiedono su processori diversi), in base a questa affermazione risulta controproducente interpellare il dizionario per ogni computazione riflessiva richiesta, in particolare quando, per soddisfare la richiesta, viene riutilizzato un canale che è appena stato utilizzato. La cosa è altamente probabile in base al principio di località che quasi tutte le applicazioni di fatto seguono. Questa ulteriore ed, in questo caso, inutile comunicazione può essere evitata spostando la fase di gestione del locking del receiver nel metodo send ed effettuando, sempre in questo metodo, dei controlli. Se il canale che è stato usato nella precedente computazione può essere usato anche nella successiva, allora il dizionario non viene interpellato, ma si interpella direttamente il canale (concetto simile a quello di *hint* introdotto nel file system distribuito Andrew [Satyanarayanan 89]).
Nel riquadro [3.3] è mostrato il codice completo relativo al processo gestore di un

generico dizionario dei canali, con l'evidenziazione di cinque delle sei fasi descritte in precedenza (la fase ②, fase di locking, è stata spostata nel metodo **send**).

```
void main( int argc, char **argv )
{
  int group_member_number = pvm_joingroup( ''DICTIONARY_GROUP'' );
  dictionary OpenChannel;
  int reply = 0, dummy_tid, msgtag, bufid, bytes, nargs;
  int my_tid = pvm_mytid(), sender_tid, receiver_tid, channel_tid;
  char *channel_type, *group_name;
  void *(*f)( reflobj, int, void ** ), **args;
  while( TRUE ) {
    bufid = pvm_recv( any_proc, any_tag );
    pvm_bufinfo( bufid, &bytes, &msgtag, &sender_tid );
    switch( msgtag ){
      case REFLECTIVE_REQUEST:
            pvm_upkint( &receiver_tid, 1, 1 ); pvm_upkstr( channel_type );
            pvm_upkmeth( f ); pvm_upkint( &nargs, 1, 1 );                       1
            pvm_upkvoidp( args, nargs, 1 );
            channel_tid = OpenChannel.get( sender_tid, receiver_tid, channel_type );
            if ( channel_tid == NULL ) {
              char *channel_args[3];
              sprintf( channel_args[0], "%d", sender_tid );
              sprintf( channel_args[1], "%d", receiver_tid );                   3
              strcpy( channel_args[2], channel_type );
              pvm_spawn( channel_type, channel_args, PvmTaskDefault, '''', 1, &channel_tid );
              OpenChannel.insert( sender_tid, receiver_tid, channel_type, channel_tid );
            }
            bufid = pvm_initsend( PvmDataRaw );
            pvm_pkmeth( f );                                                    4
            pvm_pkint( &nargs, 1, 1 ); pvm_pkvoidp( args, nargs, 1 );
            pvm_send( channel_tid, HANDLE_REQUEST );
            int rbufid = pvm_recv( channel_tid, REPLY );                        5
            pvm_setsbuf( rbufid );
            pvm_pkint( &channel_tid, 1, 1 );                                    6
            pvm_send( sender_tid, REPLY );
            break;
      case DELETE_CHANNEL_IMPLICIT:
            pvm_setsbuf( bufid );
            pvm_bcast( ''DICTIONARY_GROUP'', DELETE_CHANNEL );
            OpenChannel.kill_sender( sender_tid, kill_process );
            break;
      case DELETE_CHANNEL:
            pvm_initsend( PvmDataRaw );
            pvm_upkint( &receiver_tid, 1, 1 );
            OpenChannel.kill_receiver( receiver_tid, kill_process );
            break;
      case DELETE_CHANNEL_EXPLICIT:
            pvm_upkint( &sender_tid, 1, 1 ); pvm_upkint( &receiver_tid, 1, 1 );
            pvm_upkstr( channel_type ); pvm_upkint( &channel_tid, 1, 1 );
            OpenChannel.cancella( sender_tid, receiver_tid, channel_type );
            kill_process( channel_tid );
      }
    }
}
```

Codice C++ e PVM 3.3: Processo Gestore di un Dizionario dei Canali

3.6.6 Oggetti Riflessivi

Come si è spiegato nella sezione [2.2.1] gli oggetti riflessivi sono caratterizzati da due primitive: send e apply. Il significato ed, in linea di massima, la semantica dei due metodi in ambiente concorrente rimangono pressoché inalterati rispetto a quelli che hanno in ambiente sequenziale. Il codice del metodo apply rimane inalterato. Il metodo send, invece, viene modificato per poter gestire la metamorfosi da oggetti ad oggetti attivi delle varie entità coinvolte nella transazione.
Il metodo send coinvolge oltre le entità solite (canale e receiver) anche la nuova entità

```
void *send( int receiver_tid, char *tt, void *(*f)( reflobj, int, void ** ), int n, void **args )
{
  int reply, send_info = 0, bufid = pvm_initsend( PvmDataRaw );
  int channel_replier_tid, my_tid = pvm_mytid();
  void *response;

  if ( !(my_tid == receiver_tid ) )  // fase di locking
    if ( block_request( receiver_tid ) < 0 ) return error_code;                    1
  known_handle( my_tid, receiver_tid, tt, f, n, args, send_info, &channel_replier_tid, &reply );   2
  if ( reply == -1 ) return error_code; // si è verificato un errore
  switch ( my_tid == receiver ) {
    case TRUE:          // sender == receiver
      pvm_recv( channel_replier_tid, APPLY_REQUEST );
      pvm_upkmeth( f );
      pvm_upkint( &n, 1, 1 );
      pvm_upkvoidp( args, n, 1 );
      response = apply( f, n, args );
      bufid = pvm_initsend( PvmDataRaw );                                          3
      pvm_pkvoidp( response, 1, 1 );
      pvm_send( channel_replier_tid, APPLY_RESPONSE );
    default:          // sender ≠ receiver
      pvm_recv( channel_replier_tid APPLY_RESPONSE );
      pvm_upkvoidp( response, 1, 1 );
      if ( reply ) pvm_recv( channel_replier_tid, ACK_REQUEST );
      return( response );
  }
}
```

Codice C++ e PVM 3.4: codice del metodo send

dizionario; come tale deve essere in grado di reperire il tid del dizionario residente sul suo stesso host. Il reperimento è possibile con l'aggiunta di un attributo alla struttura dell'oggetto riflessivo contenente tale informazione, tale attributo viene inizializzato nel momento della creazione dell'oggetto attivo. I parametri di tale metodo restano gli stessi ad eccezione dell'esplicitazione del contenuto del messaggio che, come detto nel paragrafo [3.6.4.2], non è più usato. Qui finiscono le affinità dei due metodi infatti in ambiente concorrente il metodo send diventa una vera e propria comunicazione. Inoltre, il metodo send è praticamente decomposto in due metodi send e asend. Il codice dei metodi send e asend è riportato rispettivamente nei riquadri [3.4] e [3.5]. Il loro comportamento può essere schematizzato in tre fasi:

```
void *asend( int receiver_tid, char *tt, void *(*f)( reflobj, int, void ** ), int n, void **args )
{
  int reply, asend_info = 1, bufid = pvm_initsend( PvmDataRaw );
  int channel_replier_tid, my_tid = pvm_mytid();
  void *response;

  if ( !( my_tid == receiver ) )  // fase di locking
    if ( block_request( receiver ) < 0 ) return error_code;                                    1
  known_handle( my_tid, receiver_tid, tt, f, n, args, asend_info, &channel_replier_tid, &reply );  2
  if ( reply == -1 ) return error_code; // si è verificato un errore
  switch ( my_tid == receiver ) {
    case TRUE:            // sender == receiver
      pvm_recv( channel_replier_tid, APPLY_REQUEST );
      pvm_upkmeth( f );
      pvm_upkint( &n, 1, 1 );                                                                  3
      pvm_upkvoidp( args, n, 1 );
      response = apply( f, n, args );
    default:            // sender ≠ receiver
      if ( reply )  // il canale richiede la sospensione del sender
        pvm_recv( channel_replier_tid, ACK_REQUEST );
  }
}
```

Codice C++ e PVM 3.5: codice del metodo asend

① Nella prima fase il metodo, verificato che sender e receiver sono entità distinte, procede a bloccare il receiver. Questa operazione è realizzata invocando il metodo block_request, privato alla classe ReflObj.

```
int block_request( int receiver_tid ) {
  const struct timeval time_out = { 5, 5 };
  pvm_initsend( PvmDataRaw );
  pvm_send( receiver_tid, BLOCK_REQUEST );
  while( TRUE ) {
    if ( pvm_pstat( receiver_tid ) != PvmOk ) return -1;
    if ( pvm_trecv( receiver_tid, ACKNOWLEDGE, time_out ) > 0 ) return 0;
  }
}
```

② Nella seconda fase, il metodo, determina se si può riutilizzare il canale usato precedentemente. In caso negativo, inoltra la richiesta per il canale al dizionario e si pone in attesa di ricevere dal dizionario informazioni sul canale da usare e sul tipo di computazione da intraprendere (cioè se il canale modifica o meno il suo stato [3.4.2.1]). In caso affermativo, noto il tid del canale ed il tipo di computazione necessaria, salta la comunicazione con il dizionario ed interpella direttamente il canale (queste informazioni vengono anche inoltrate al receiver). Il tutto è realizzato attivando il metodo known_handle, anch'esso privato della classe ReflObj.

```
void known_handle( int sender_tid, int receiver_tid, char *kind, void *(*f)( reflobj, int, void ** ),
```

```
                                        int n, void **args, int info, int *channel_tid, int *reply ) {
  static int static_channel_tid = NULL, static_receiver_tid, static_reply;
  static char *static_kind;
  int rbufid, not_reuse = ( static_channel_tid == NULL ) ||
                ( !strcmp( static_kind, tt ) ) || !(static_receiver_tid == receiver_tid );
  pvm_initsend( PvmDataRaw );
  if ( not_reuse ) {
    static_receiver_tid = receiver_tid; strcpy( static_kind, kind );
    pvm_pkint( &receiver_tid, 1, 1 );
    pvm_pkstr( kind );
  }
  pvm_pkmeth( f ); pvm_pkint( &n, 1, 1 );
  pvm_pkvoidp( args, n, 1 ); pvm_pkint( &info, 1, 1 );
  if ( not_reuse ) {
    pvm_send( dictionary_tid, REFLECTIVE_REQUEST );
    rbufid = pvm_recv( dictionary_tid, REPLY );
    pvm_upkint( &static_reply, 1, 1 );
    pvm_upkint( &static_channel_tid, 1, 1 );
  } else pvm_send( static_channel_tid, HANDLE_MESSAGE );
  if ( sender_tid == receiver_tid ) {
    if ( not_reuse ) pvm_sets( rbufid );
    else {
      pvm_initsend( PvmDataRaw );
      pvm_pkint( &static_reply, 1, 1 );
      pvm_pkint( &static_channel_tid, 1, 1 );
    }
    pvm_send( receiver_tid, REPLY );
  }
  *channel_tid = static_channel_tid;
  *reply = static_reply;
}
```

È quest'ultimo metodo a mantenere le informazioni relative all'ultimo canale utilizzato. Esso viene attivato sia da send che da asend, così le informazioni relative al canale riguardano sia i canali usati da send che quelli usati da asend. Questa strategia si basa sull'impiego del principio di località delle informazioni che permette di ridurre i tempi "morti" nei casi più frequenti, come si può osservare dalla figura [3.9]

③ La terza fase è quella più delicata ed è quella che distingue i metodi send e asend. Grazie alla risposta del canale attesa al punto ②, si è in grado di determinare se quest'ultimo necessita di una computazione sincrona o asincrona ed in base al valore del tid del receiver, il metodo, stabilisce quale comportamento tenere:

- Se la richiesta è stata inoltrata tramite send, allora in ogni caso (cioè computazione sincrona o asincrona):
  - Se il sender coincide con il receiver si attende una richiesta di apply (APPLY_REQUEST) e successivamente il valore di ritorno (APPLY_RESPONSE).

    - Se il sender e il receiver sono distinti, allora si attende esclusivamente il valore di ritorno (APPLY_RESPONSE).

  - Se la richiesta è stata inoltrata tramite il metodo asend allora si distinguono due casi

    - il canale usato modifica lo stato dell'oggetto allora
      - ⋆ Se il sender coincide con il receiver si pone in attesa di una richiesta di apply (APPLY_REQUEST) e successivamente di un segnale di acknowledge (ACK_REQUEST).
      - ⋆ Se il sender non coincide con il receiver, allora si pone in attesa esclusivamente del segnale di acknowledge (ACK_REQUEST).
    - il canale usato non modifica lo stato dell'oggetto allora
      - ⋆ Se il sender coincide con il receiver si pone in attesa di una richiesta di apply (APPLY_REQUEST) e successivamente termina.
      - ⋆ Se il sender è diverso dal receiver allora, semplicemente termina la propria computazione.

Il processo gestore di un oggetto riflessivo deve poter gestire i metodi dell'oggetto più i metodi send e apply ed ovviamente la richiesta di locking. Nel riquadro [3.6] è mostrato il codice di un processo gestore per un generico oggetto riflessivo, in cui si omette il codice relativo alla gestione dei metodi propri dell'oggetto riflessivo.

3.6.7 Il Canale ed il Metodo HandleMessage

Il canale è anch'esso un oggetto attivo, il cui processo gestore si limita a gestire la richiesta per il metodo HandleMessage e per la cancellazione del canale come è mostrato nel riquadro [3.7]. Da osservare che l'informazione sul tipo di computazione effettuata (sincrona o asincrona) è codificata nel canale e precisamente nell'attributo reply.
Per quanto riguarda l'unico metodo del canale (HandleMessage), questo presenta oltre al comportamento riflessivo una parte di comunicazione con il receiver, necessaria per richiedere la computazione del metodo apply, e con il sender per fornirgli il risultato della computazione. Questa parte è comune ad ogni tipo di canale ed è completamente separata dal comportamento riflessivo, quindi per una migliore modularità si può progettare il metodo in modo che la parte di comportamento riflessivo sia incapsulata all'interno del metodo. Questa parte è definita dal progettista del canale, mentre la parte fissa (comunicazione) è incapsulata all'interno di un metodo (HandleMessageCommunication) che deve essere attivato dal metodo HandleMessage. Come è mostrato nel codice riportato nel riquadro [3.8].

```
main( int argc, char **argv )
{
 int bufid, bytes, msgtag, sender_tid, reply;
 refl_X rx( argv[1], ..., atoi( argv[argc-1]) );
 while( TRUE ) {

   bufid = pvm_recv( any_proc, any_tag );
   pvm_bufinfo( bufid, &byte, &msgtag, &sender_tid );
   switch( msgtag ) {
    case BLOCK_REQUEST:
          pvm_initsend( PvmDataRaw );
          pvm_send( sender_tid, ACKNOWLEDGE );
          pvm_recv( sender_tid, REPLY );
          pvm_upkint( &reply, 1, 1 );
          pvm_upkint( &sender_tid, 1, 1 );
          pvm_recv( sender_tid, APPLY_REQUEST );
    case APPLY_REQUEST:
          void *response, *(*f)( reflobj, int, void ** ), **args;
          int n;
          pvm_upkmeth( f );
          pvm_upkint( &n, 1, 1 );
          pvm_upkvoidp( args, n , 1 );
          response = rx.apply( f, n, args );
          pvm_initsend( PvmDataRaw );
          pvm_pkvoidp( response, 1, 1 );
          pvm_send( sender_tid, APPLY_RESPONSE );
          break;
    case SEND_REQUEST:
          int r_tid, n;
          void **args, *(*f)( reflobj, int, void ** ), *response;
          char *tt;
          pvm_upkint( &r_tid, 1, 1 );
          pvm_upkstr( tt );
          pvm_upkmeth( f );
          pvm_upkint( &n, 1, 1 );
          pvm_upkvoidp( args, n, 1 );
          response = rx.send( r_tid, tt, f, n, args );
          break;
   }
 }
}
```

Codice C++ e PVM 3.6: gestore degli oggetti riflessivi

3.6.7.1 Cooperazione tra Canali

I canali instaurati tra gli stessi oggetti attivi sono concettualmente simili, in quanto differiscono unicamente per il loro comportamento e non per le entità che mettono in contatto. Essendo essi stessi degli oggetti attivi possono formare un gruppo di oggetti identificato dai tid delle entità che uniscono (ad esempio "gruppoXXX.YYY" dove XXX e YYY sono rispettivamente i tid del sender e del receiver dei canali appartenenti al gruppo). Un gruppo di canali agevola la condivisione di informazioni tra meta-entità tramite lo scambio di messaggi contenenti le informazioni da condividere. Questo ovviamente, come si vede nell'esempio successivo, porta ad

```
void main( int argc, char **argv )
{
  int bufid, bytes, msgtag, tid, info, n_args;
  channel ch( atoi(argv[1]), atoi(argv[2]), argv[3] );
  void *(*f)( reflobj, int, void ** ), **args;

  while( TRUE ) {
    bufid = pvm_recv( -1, -1 );
    pvm_bufinfo( bufid, &bytes, &msgtag, &tid );
    switch ( msgtag ) {
      case KILL_CHANNEL:
           delete ch;
           break;
      case HANDLE_REQUEST:
           pvm_initsend( PvmDataRaw );
           pvm_pkint( &ch.reply, 1, 1 );
           pvm_send( tid, REPLY );
      case HANDLE_MESSAGE:
           pvm_upkmeth( f );
           pvm_upkint( &n_args, 1, 1 );
           pvm_upkvoidp( args, n_args, 1 );
           pvm_upkint( &info, 1, 1 );
           ch.set_info( info );
           ch.HandleMessage( f, n_args, args );
           break;
    }
  }
}
```

Codice C++ e Pvm 3.7: gestore del canale

una semplificazione nella progettazione di applicazioni riflessive ottenute usando il modello a materializzazione del canale, aumentando ancora le sue potenzialità ed eliminando uno dei suoi punti deboli nei confronti del modello a meta-oggetto.

3.6.8 Struttura ed Interazione di un'Applicazione Riflessiva

Ora si può descrivere nel suo complesso l'architettura di una generica applicazione riflessiva e dettagliare i passi eseguiti durante una generica computazione riflessiva. L'applicazione è composta da un certo numero di oggetti riflessivi distribuiti su diversi processori, per ogni processore si ha un dizionario dei canali, che tiene traccia dei canali, ed ovviamente un certo numero di canali, che sovraintendono il comportamentento riflessivo dell'applicazione. I dizionari sono raggruppati in un apposito gruppo d'oggetti ed i canali instaurati tra gli stessi oggetti riflessivi possono essere raggruppati in un insieme di canali (come visto nella precedente sezione).
Le entità che compongono un'ipotetica applicazione riflessiva possono essere classificate in tre categorie:

```
class channel {
  public:
    int info, reply;
    void HandleMessageCommunication( void *(*f)( reflobj, int, void ** ), int n, void **args );
    int sender_tid, receiver_tid;
    char *tag_type;
   // costruttore
    channel( int s, int r, char *tt ) : reply(0), sender_tid(s), receiver_tid(r), tag_type(tt) {};
   // distruttore
    ~channel() {pvm_exit();exit(1);};
   // metodi
    void  HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args );
    void set_info( int n ) { info = n; };
}

void channel::HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args )
{
  printf( ''Meta-Livello\n'' );
  HandleMessageCommunication( f, n, args );
}

void *channel::HandleMessageCommunication( void *(*f)( reflobj, int, void ** ), int n, void **args )
{
  pvm_initsend( PvmDataRaw );
  pvm_pkmeth( f );
  pvm_pkint( &n, 1, 1 );
  pvm_pkvoidp( args, n, 1 );
  int my_tid = pvm_mytid();
  pvm_pkint( &mytid, 1, 1 );
  pvm_send( receiver_tid, APPLY_REQUEST );
  int rbufid = pvm_recv( receiver_tid, APPLY_RESPONSE );
  if ( info  == 0 ) {
    pvm_setsbuf( rbufid );
    pvm_send( sender_tid, APPLY_RESPONSE );
  } else
    if ( reply ) pvm_send( sender_tid, ACK_REQUEST );
}
```

Codice C++ e PVM 3.8: Classe channel

- Le entità del livello base, cioè gli oggetti riflessivi, che caratterizzano il comportamento dell'applicazione, la cui struttura ed interazione è a completa discrezione del programmatore dell'applicazione.

- Le meta-entità, cioè i canali, che caratterizzano il comportamento riflessivo dell'applicazione, la cui struttura è a discrezione del progettista dei canali, mentre l'interazione con gli oggetti riflessivi è lasciata al programmatore dell'applicazione.

- Il dizionario dei canali (o gruppo di dizionari), la cui presenza, per non parlare della struttura, è completamente nascosta al programmatore dell'applicazione, e nota solo al progettista delle meta-entità.

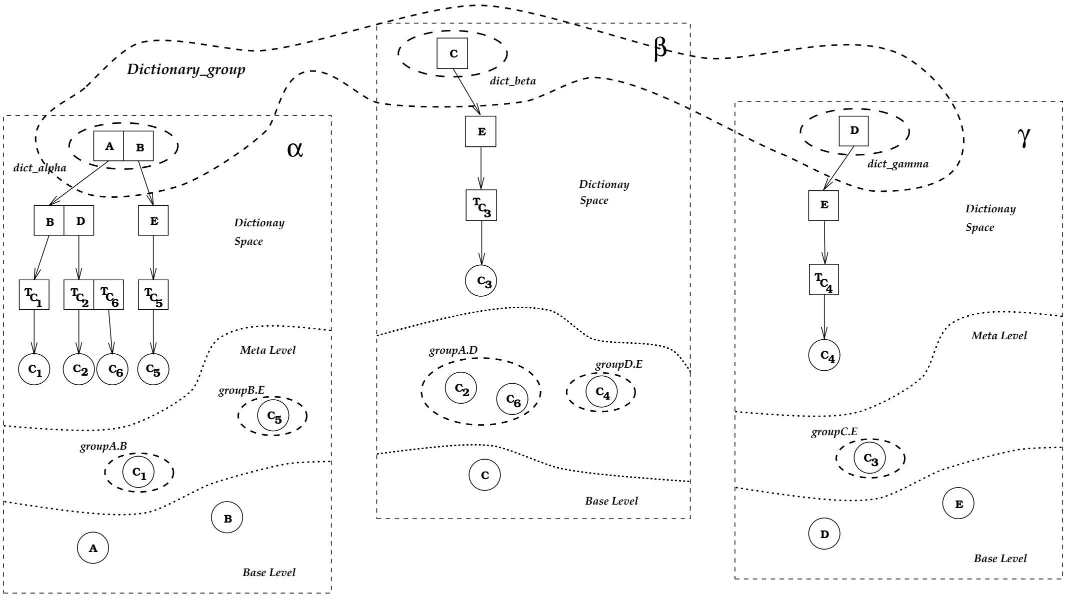


Figura 3.8: **Struttura tipo di una generica applicazione riflessiva**

Da questo schema risulta che un'applicazione riflessiva può essere sviluppata in tempi e da persone diverse; un gruppo progetta l'applicazione vera e propria, un altro le meta-entità (possono essere riusate da progetti precedenti) ed infine un gruppo, sfruttando le proprietà della modularità presentate nella sezione [2.5], compone, con il minimo sforzo produttivo, i due livelli assieme.

Per quanto riguarda l'interazione delle entità che compongono questa ipotetica applicazione, si supponga che $\mathcal{A}$ richieda a $\mathcal{D}$ un servizio f, il cui comportamento riflessivo deve essere di genere $\mathcal{T}_{\mathcal{C}_6}$:

① $\mathcal{A}$ tramite il metodo asend effettua la richiesta riflessiva del servizio f all'oggetto $\mathcal{D}$, specificando che il comportamento riflessivo deve essere di tipo $\mathcal{T}_{\mathcal{C}_6}$.

$\mathcal{A}$.asend(tid_$\mathcal{D}$, $\mathcal{T}_{\mathcal{C}_6}$, f, 0, NULL);[17]

② Il metodo asend verificato che il sender ed il receiver sono diversi, inoltra una richiesta di locking al receiver

```
pvm_send( receiver_tid, BLOCK_REQUEST );
```
[18]

[17] Per semplicità si suppone che la computazione richiesta sia non bloccante (asend), tid_$\mathcal{D}$ sia il tid dell'oggetto $\mathcal{D}$ e che f non abbia parametri.

③ L'oggetto $\mathcal{A}$ sospende la propria attività in attesa di ricevere l'acknowledge che il receiver è effettivamente bloccato. Questa è una sospensione a tempo, iterata con il controllo che l'oggetto $\mathcal{D}$ sia ancora attivo[19]

④ Il metodo asend, ricevuto l'acknowledge del receiver, consulta la propria memoria (known_handle), verificando la necessità di determinare un nuovo canale. In caso affermativo il metodo known_handle richiede al dizionario (dict_α) residente sull'host di $\mathcal{A}$ (α) di determinare|creare un canale[20] di genere $\mathcal{T}_{\mathcal{C}_6}$ ed instaurato tra gli oggetti $\mathcal{A}$ e $\mathcal{D}$

```
pvm_send( dictionary_tid, REFLECTIVE_REQUEST );[21]
```

Se invece si può usare il canale precedentemente usato, si evita la comunicazione con il dizionario, interpellando direttamente il canale

```
pvm_send( static_channel_tid, HANDLE_MESSAGE );[22]
```

Quindi $\mathcal{A}$ si pone in attesa della risposta del dizionario o del canale a seconda del caso. Si supponga, caso più generale, la necessità di interpellare il dizionario dei canali.

⑤ Interpellato dal sender, il dizionario dei canali determina il canale soddisfacente la computazione richiesta, creandolo nel caso non esista.

```
channel_tid = OpenChannel.get( sender_tid, receiver_tid, channel_type );[23]
```

⑥ Il dizionario gira la richiesta al canale determinato|creato

```
pvm_send( channel_tid, HANDLE_REQUEST );
```

e si pone in attesa di conoscere il tipo di computazione necessaria al canale e che deve poi girare al sender.

---

[18] La variabile receiver_tid contiene il tid dell'oggetto attivo $\mathcal{D}$.

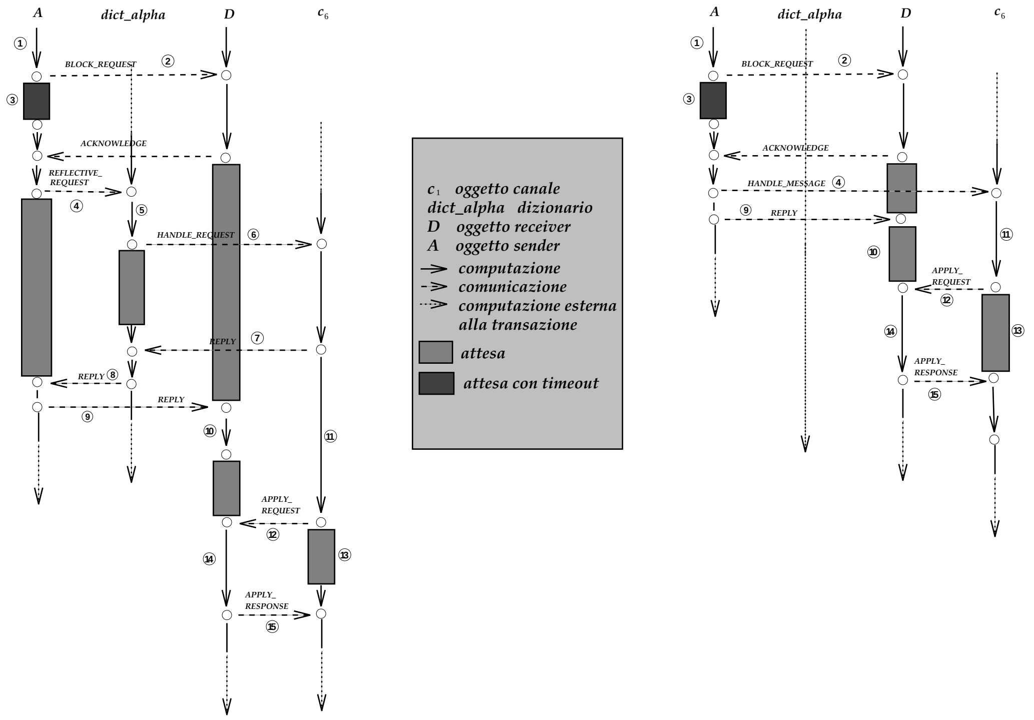
[19] I punti ② e ③ vengono saltati se il sender ed il receiver coincidono.

[20] Si suppone che si sia cambiato il tipo di comportamento riflessivo richiesto e che il canale adatto a soddisfare la richiesta sia già stato instaurato.

[21] La variabile dictionary_tid contiene il valore del tid del dizionario dei canali dict_α.

[22] La variabile static_channel_tid rappresenta il "ricordo" del tid del precedente canale utilizzato.

[23] Le variabili sender_tid, receiver_tid e channel_type contengono rispettivamente il tid degli oggetti attivi $\mathcal{A}$ e $\mathcal{D}$ ed il genere del canale richiesto, cioè $\mathcal{T}_{\mathcal{C}_6}$, mentre la variabile channel_tid contiene il tid del canale $\mathcal{C}_1$.



Viene Consultato il Dizionario dei Canali - Figura A

Figura B - Non Viene Consultato il Dizionario dei Canali

Figura 3.9: **Traccia di un Generica Computazione Riflessiva tra due Oggetti Attivi - A) Caso in cui si Interpella il Dizionario dei Canali. B) Caso in cui si Sfrutta il Principio di Località**

```
int rbufid = pvm_recv( channel_tid, REPLY );
```

⑦ Il canale, ricevuta la richiesta, invia al richiedente (dizionario) le proprie richieste computazionali (cioè se modifica o meno lo stato degli oggetti coinvolti). Si supponga che non modifichi lo stato di alcun oggetto.

```
pvm_send( tid, REPLY );
```
[24]

⑧ Il dizionario dei canali riceve la risposta dal canale e la gira al sender assieme al tid del canale.

```
pvm_setsbuf( rbufid );
```

[24] La variabile tid contiene il tid del richiedente sia che esso sia dict_α che $\mathcal{A}$.

```
pvm_pkint( &channel_tid, 1, 1 );
pvm_send( sender_tid, REPLY );
```

Il dizionario prosegue la propria computazione evadendo la successiva richiesta. Nel caso il canale precedentemente usato venga riusato, i passi da ⑤ a ⑧ non vengono effettuati come si può vedere in figura [3.9.B].

⑨ Il sender invia le informazioni necessarie (canale usato e tipo di computazione richiesta) al receiver e siccome, per ipotesi, il canale non necessita il locking del sender, questo passa ad evadere le successive richieste.

⑩ Il receiver riceve il messaggio del sender e si pone in attesa di ricevere una richiesta di apply da parte del canale.

```
pvm_recv( sender_tid, APPLY_REQUEST );
```
[25]

⑪ Il canale attiva il metodo HandleMessage.

```
ch.HandleMessage( f, n_args, args );
```
[26]

⑫ Il metodo HandleMessage, espletate le istruzioni caratterizzanti il suo comportamento riflessivo, richiede al receiver l'esecuzione effettiva del metodo f.

```
pvm_send( receiver_tid, APPLY_REQUEST );
```

⑬ Il canale si pone in attesa della terminazione della computazione del receiver.

```
pvm_recv( receiver_tid, APPLY_RESPONSE );
```

⑭ Il receiver riceve la richiesta di apply quindi evade tale richiesta

```
response = rx.apply( f, n, args );
```

⑮ Il receiver ritorna al canale il risultato della computazione

```
pvm_pkvoidp( response, 1, 1 );
pvm_send( sender_tid, APPLY_RESPONSE );
```

Quindi passa a gestire la successiva richiesta. Il canale ricevuto il valore di ritorno completa tutta la computazione riflessiva e passa alla gestione delle successive richieste.

La transazione, sia nel caso che si interpelli il dizionario dei canali sia che non lo si interpelli grazie al principio di località, è schematizzata nella figura [3.9].

[25] La variabile sender_tid contiene il genere del canale $\mathcal{C}_6$.

[26] La variabile f è il riferimento al metodo f, n_args vale zero e args ha valore NULL.

3.6.9 Un Esempio Completo

Si consideri la situazione descritta nella sezione [1.9], in cui si intende costruire una classe, le cui istanze gestiscono le operazioni base su un file (lettura, scrittura, ...) e si è interessati a fornire questa classe di un comportamento riflessivo che consenta, per ogni file creato da un'istanza di questa classe, di mantenere una copia di sicurezza consistente con il file originale. A questo proposito ogni operazione implementata dalla classe ha un comportamento riflessivo differente, che segue le seguenti direttive:

- All'atto della creazione del file, viene creato un secondo file, usato come copia di sicurezza dell'originale.
- L'operazione di scrittura di un carattere sul file scrive tale carattere sul file originale ed automaticamente anche sulla copia.
- L'operazione di lettura legge un dato dal file originale, verificando con la copia la correttezza del dato che è stato letto.

A queste operazioni, per la seguente trattazione, viene aggiunta l'operazione di chiusura del file il cui comportamento riflessivo, ovviamente, è quello di chiudere il file copia più alcune altre operazioni che si discutono successivamente. Il seguente stub di codice è la definizione della classe SafeDiskAccess:

```
% --- classe SafeDiskAccess ---

 class SafeDiskAccess {
   private:
     FILE *fp;
   public:
     void open( char *name ) { fp = fopen( name, ''wr'' ); };
     void read( char *ch ) { *ch = getc( fp ); };
     void write( char ch ) { putc( ch, fp ); };
     void close( void ) { fclose( fp ); };
 };
```

Si può facilmente osservare che questa è una normalissima classe C++, che non presenta alcun comportamento riflessivo. Questo esempio mostra, perciò, come rendere riflessive le istanze di una classe di cui non si vuole modificare il sorgente.
È nostro interesse costruire una classe le cui istanze abbiano un certo comportamento riflessivo; in accordo con quanto affermato nella sezione [2.2.1], la capacità riflessiva è supportata dalla classe ReflObj ed, in accordo con quanto affermato nella sezione [2.5], per una maggiore modularità si crea un'apposita classe che eredita dalla classe ReflObj e dalla classe SafeDiskAccess, come è mostrato nella seguente dichiarazione:

```
% --- classe Refl_SafeDiskAccess ---

 class Refl_SafeDiskAccess public: ReflObj, public: SafeDiskAccess {};
```

Da questo momento ogni istanza della classe Refl_SafeDiskAccess ha un comportamento riflessivo tale da garantire i requisiti di sicurezza proposti, invece le istanze della classe SafeDiskAccess non mantengono alcuna copia del proprio file.

La gestione del comportamento riflessivo è lasciata ai canali. La possibilità di associare più canali ad ogni oggetto e la granularità più fine offerta permettono di definire per ogni operazione (richiedente una computazione riflessiva differente) una sotto-classe di channel differente. Quindi essendo richiesti quattro comportamenti riflessivi differenti si hanno quattro sotto-classi di channel (SafeOpenChannel, SafeReadChannel, SafeWriteChannel e SafeCloseChannel), di conseguenza ogni istanza della classe Refl_SafeDiskAccess è supervisionata da quattro canali.
Come si è visto nell'esempio presentato nella sezione [1.9] il problema principale di questa scomposizione dei compiti tra i quattro canali è rappresentato dalla difficoltà di condividere le informazioni, nella fattispecie il file descriptor del file copia (si tralasciano i problemi di località dei file all'host su cui è memorizzato, di localizzazione e descrizione dei metodi di pack e unpack per i file descriptor). In SmallTalk si è risolto il problema introducendo un nuovo meta-livello in cui un meta-canale, quando necessario, fornisce tale informazione ai sottostanti canali aumentando ulteriormente la complessità dell'applicazione, con nuove classi ed istanze.
Pensare i canali, instaurati tra gli stessi oggetti, come facenti parte di un'unica entità (gruppo di canali) facilita questa condivisione di informazioni; infatti è possibile condividere delle informazioni tramite scambio di messaggi, richiedendole quando necessarie. Nel caso specifico l'informazione da condividere è solamente il file descriptor. Questa, però, è un'informazione in evoluzione, in quanto ogni operazione di lettura|scrittura modifica la posizione corrente all'interno del file associato al file descriptor. Inizialmente il file descriptor è recuperabile presso il canale che ne gestisce l'apertura (questo quando gli è richiesto invia la versione più recente del file descriptor) ed ogniqualvolta viene espletata un'operazione il file descriptor aggiornato, è inviato ad ogni elemento del gruppo. Nel momento in cui si riattiva un canale, vengono "consumati" i messaggi obsoleti (tramite il metodo updating) recuperando l'ultimo aggiornamento del file descriptor. Questa non pretende di essere la soluzione migliore, anzi è una soluzione che da origine ad un elevato traffico di messaggi sulla rete, però sottolinea bene le potenzialità espresse dalla cooperazione dei canali, relativamente alla condivisione di informazioni.

Le operazioni per gestire il gruppo possono essere inserite in un'apposita sotto-classe di channel (GroupChannel) che eredita il comportamento base dalla classe channel e supporta trasparentemente l'arruolarsi di un canale al gruppo identificato dai tid degli oggetti tra cui è instaurato ed il relativo dissociarsi. Ovviamente l'arruolamento ha luogo nel momento in cui il canale viene creato, mentre il dissociamento ha luogo nel momento della sua distruzione, la quale implica anche la sua cancellazione dal dizionario. Quindi queste operazioni sono eseguite rispettivamente dal costruttore e dal distruttore. A queste operazioni si aggiungono i metodi to_all, updating e share che consentono alle istanze di questa classe (o sue sotto-classi) di condividere (tramite scambio di messaggi) informazioni con gli altri oggetti appartenenti all'insieme. Di seguito è mostrato il codice della classe GroupChannel;

```
%% --- canale appartenente ad un gruppo ---

 class GroupChannel public: channel {
   private:
     int group_membership;
     int sbufid, rbufid;
   public:
    // costruttore inserisce l'oggetto nel gruppo
     GroupChannel( int s, int r, char *t );
    // distruttore l'oggetto lascia il gruppo e si suicida
     ~GroupChannel( void );
    // metodi per la condivisione e trasparenza dei gruppi
     void to_all( int msg_tag ) { pvm_bcast( "group"+sender_tid+"."+receiver_tid, msg_tag ); };[27]
     int share( int msg_tag );
     int updating( int msg_tag );
 }

  GroupChannel::GroupChannel( int s, int r, char *t ) : channel( s, r, t )
  {
    group_membership = pvm_joingroup( "group"+sender_tid+"."+receiver_tid );
    sbufid = pvm_initsend( PvmDataRaw );
  };

 GroupChannel::~GroupChannel( void )
 {
   pvm_initsend( PvmDataRaw );  // richiesta di cancellazione esplicita del canale
   pvm_pkint( &sender_tid, 1, 1 ); pvm_pkint( &receiver_tid, 1, 1 );
   pvm_pkstr( channel_tag ); my_tid = pvm_mytid(); pvm_pkint( &my_tid, 1, 1 );
   pvm_send( dictionary_tid, DELETE_EXPLICIT );
   pvm_lvgroup( "group"+sender_tid+"."+receiver_tid );
   pvm_exit();
   exit( Ok );
 };

 int GroupChannel::updating( int msg_tag ) {
   int updated = 0, rbufid;
   while( rbufid = pvm_nrecv( any_proc, msg_tag ) > 0 ) updated = rbufid;
   return updated;
 };

 int GroupChannel::share( int msg_tag ) {
   int open_channel_tid = pvm_gettid( "group"+sender_tid+"."+receiver_tid, 0 );
   pvm_send( open_channel_tid, SHARE );
   return pvm_recv( open_channel_tid, msg_tag );
 };
```

Come si può osservare la classe GroupChannel agevola la cooperazione tra canali permettendo, tramite un'interfaccia precisa e ben definita (share, updating e to_all) di condividere informazioni tra istanze delle sotto-classi della classe channel. In questo modo è evidente la semplificazione indotta nella costruzione dell'applicazione; si hanno meno classi da definire ex-novo, e si elimina il meta-meta-livello. Infine si

[27] In questo esempio si è supposta l'esistenza di una classe che implementi il tipo stringhe con l'operazione di concatenazione tra stringhe e tra stringhe ed interi espressa dal simbolo +.

riduce il numero di oggetti attivi interagenti. Di seguito è riportato il codice delle classi SafeOpenAccess, SafeReadAccess, SafeWriteAccess e SafeCloseAccess.

```
%% --- canale per la gestione della open ---

 class SafeOpenChannel public: GroupChannel {
   private:
     FILE *safefp;
     void sharing( void );
   public:
    // costruttore
    SafeOpenChannel( int s, int r, char *t, int d ) : GroupChannel( s, r, t, d ), safefp( NULL ) reply(1) {};
    // distruttore
     ~SafeOpenChannel( void ) : ~GroupChannel() {};
    // ridefinizione del metodo HandleMessage
     void HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args );
  }

 void SafeOpenChannel::HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args ) {
     // apre il file replica quindi chiede l'applicazione della open
     // all'entità del livello base
   safefp = fopen( args[0]+''.replica'', ''wr'' );
   pvm_pkfilep( safefp );
   HandleMessageCommunication( f, n, args );
   sharing();
 }

 void sharing( void ) {
   int bytes, msg_tag, tid;
   while( TRUE ) {
     sbufid = pvm_recv( any_proc, SHARE );
     pvm_bufinfo( sbufid, &bytes, &msg_tag, &tid );
     updating(FILE_FP);
     pvm_setsbuf( rbufid );
   }
   pvm_send( tid, FILE_FP );
 }


%% --- canale per la gestione della read ---

 class SafeReadChannel public: GroupChannel {
   private:
     FILE *safefp;
   public:
    // costruttore
    SafeReadChannel( int s, int r, char *t, int d ) : GroupChannel( s, r, t, d ), safefp( NULL ), reply(0) {};
    // distruttore
     ~SafeReadChannel( void ) {};
    // ridefinizione del metodo HandleMessage
     void HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args );
 }

 void SafeReadChannel::HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args ) {
     // se non ha il puntatore al file lo richiede alla SafeOpenChannel
     // controlla se il carattere letto da fp è uguale a quello letto da safefp
   if ( safefp == NULL ) { // inizializzazione
     share( FILE_FP );
     pvm_upkfilep( safefp );
```

```
    }
    else if ( updating( FILE_FP ) != 0 ) pvm_upkfilep( safefp );
    HandleMessageCommunication( f, n, args );
    char ch = getc( safefp );
    if ( ch != *(char *)args[0] ) {
      cout << ‘‘errore ’’ << ch << ‘‘è diverso da ’’ << *args << endl;
      pvm_send( receiver_tid, CLOSE );
    }
    pvm_initsend( PvmDataRaw ); pvm_pkfilep( filefp );
    to_all( FILE_FP );
  }

%% --- canale per la gestione della write ---

  class SafeWriteChannel public: GroupChannel {
    private:
      FILE *safefp;
    public:
     // costruttore
     SafeWriteChannel( int s, int r, char *t, int d ) : GroupChannel( s, r, t, d ), safefp( NULL ), reply(1) {};
     // distruttore
      ~SafeWriteChannel( void ) {};
     // ridefinizione del metodo HandleMessage
      void HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args );
   }

  void SafeWriteChannel::HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args ) {
      // se non ha il puntatore al file lo richiede alla SafeOpenChannel
      // scrive il carattere sia su fp che su safefp
    if ( safefp == NULL ) { // inizializzazione
      share( FILE_FP );
      pvm_upkfilep( safefp );
    }
    else if ( updating( FILE_FP ) != 0 ) pvm_upkfilep( safefp );
    HandleMessageCommunication( f, n, args );
    putc( *args, safefp );
    pvm_initsend( PvmDataRaw ); pvm_pkfilep( filefp );
    to_all( FILE_FP );
  }

%% --- canale per la gestione della close ---

  class SafeCloseChannel public: GroupChannel {
    private:
      FILE *safefp;
    public:
     // costruttore
      SafeCloseChannel( int s, int r, char *t, int d ) : GroupChannel( s, r, t, d ), safefp( NULL ) {};
     // distruttore
      ~SafeCloseChannel( void ) : ~GroupChannel() {};
     // ridefinizione del metodo HandleMessage
      void HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args );
   }

  void SafeCloseChannel::HandleMessage( void *(*f)( reflobj, int, void ** ), int n, void **args ) {
      // se non ha il puntatore al file lo richiede alla SafeOpenChannel
      // chiude il file safefp e ordina a tutti i canali del gruppo di finire le proprie attività
    if ( safefp == NULL ) { // inizializzazione
      share( FILE_FP );
```

```
    pvm_upkfilep( safefp );
  }
  else if ( updating( FILE_FP ) != 0 ) pvm_upkfilep( safefp );
  HandleMessageCommunication( f, n, args );
  fclose( safefp );
  to_all( CLOSE_UP );
  ~SafeCloseChannel();
}
```

I principali motiviper cui si è inserita un'operazione di chiusura del file con relativo comportamento riflessivo e relativo canale che la gestisce, sono:

- Completezza; un file che viene aperto deve anche essere chiuso, come pure devono essere eliminati i canali ad esso associati.

- Sicurezza; nel momento in cui si riscontra un errore nella gestione dell'operazione di lettura, si attiva la fase di chiusura, eliminando tutti i canali ad esso connessi.

Il motivo più importante per questa trattazione è comunque quello di mostrare la semplicità offerta dalla comunicazione in broadcast gestita tramite il gruppo di canali, in luogo di una comunicazione gestita dall'oggetto, che dovrebbe conoscere il tid di tutte le entità con cui vuole comunicare.

Come si può facilmente osservare le classi del livello base, a meno che non facciano esplicitamente uso della concorrenza, sono semplici classi C++, mentre le operazioni di comunicazione, sincronizzazione, eccetera sono incapsulate nella classe ReflObj ed in particolare nei metodi send e apply e nei canali, incapsulando così in pochi punti le aree interessate alla gestione della distribuzione, aree che possono rimanere nascoste al fruitore di una libreria di classi di canali.

Nota In questo esempio non si sono prese in considerazione le problematiche di mutua esclusione tra operazioni di lettura e scrittura perché gli oggetti attivi che si sono costruiti, sfruttando PVM, non possiedono i thread, quindi le richieste vengono schedulate secondo l'ordine di arrivo, perciò operazioni di lettura e scrittura non possono sovrapporsi. Sfruttando altri sistemi ad oggetti concorrenti che supportano i thread (al posto di PVM), il problema non può essere trascurato; per la risoluzione di tale problema si rimanda alle risoluzioni dei casi di studio (ad esempio il problema dei lettori-scrittori) riportati in [Ben-Ari 90].

Nota Grazie alla cooperazione tra canali l'uso di un meta-meta-livello è limitato esclusivamente ai casi in cui si necessita una computazione riflessiva sulle azioni dei canali.

3.7 Conclusioni

In questo capitolo si è esaminato l'impatto della riflessione sui sistemi concorrenti, in particolare si è esaminato un sistema che implementa il modello a meta-oggetto (OpenC++) e soprattutto come integrare PVM con il modello a materializzazione del canale.
A questo punto non è possibile concludere senza confrontarsi con uno standard de facto, tra i sistemi ad oggetti distribuiti come CORBA [Corba 93].

CORBA (Common Object Request Broker Architecture) è la specifica, cioè una descrizione concreta dell'interfaccia e dei servizi che devono essere forniti da un sistema ad oggetti distribuito per ottenere comunicazione tra oggetti, attivazioni di metodi e localizzazione degli oggetti in modo trasparente.
La specifica di CORBA è stata sviluppata da OMG (Object Management Group) che è un'associazione accademica ed industriale, di cui tra l'altro fanno parte *Digital*, *IBM* e *Sun*, dedita alla definizione di standard.
La specifica di CORBA o almeno parte delle idee che espone sono state implementate da diversi tool come OLE di *Microsoft* e SOM di *IBM* per citarne alcuni.

CORBA è composto da cinque componenti principali:

⋆ Object Request Broker (ORB)

⋆ Linguaggio di definizione delle interfacce (IDL)

⋆ Interfaccia per l'invocazione dinamica

⋆ Deposito delle Interfacce

⋆ Basic Object Adapter (BOA)

ORB: gli oggetti forniscono i servizi richiesti dai clienti. Scopo di ORB è quello di smistare le richieste dei clienti agli oggetti e di restituire il risultato dei servizi ai clienti. Il lavoro di ORB è completamente trasparente al cliente. I clienti non hanno la necessità di sapere dov'è l'oggetto o come è fatto, ma devono solamente conoscere il riferimento all'oggetto da cui si vuole il servizio. È ORB che tramite il riferimento posseduto dal cliente effettua la localizzazione. Le richieste agli oggetti possono essere inoltrate in due modi: invocazione statica (tramite IDL) e dinamica (tramite l'apposita interfaccia). In base alle modalità con cui viene inoltrata la richiesta, ORB localizza l'oggetto desiderato, attiva il servizio e restituisce i risultati.

IDL: affinché un cliente possa usufruire dei servizi offerti da un oggetto, occorre che conosca quali servizi l'oggetto offre. In CORBA l'interfaccia degli oggetti è descritta

in un linguaggio dichiarativo (IDL), la cui sintassi è simile a quella del C++.
Le interfacce sono composte da operazioni, che descrivono i servizi offerti dall'oggetto a loro volta composti dal nome e dal tipo dei valori manipolati dal servizio.
Le interfacce possono essere usate come tipi di valori manipolati dai servizi e possono essere ereditate.
Essendo IDL un linguaggio dichiarativo, l'implementazione degli oggetti è separata dall'interfaccia degli stessi. Questa separazione, concede, tra l'altro, la possibilità di implementare gli oggetti ed i servizi da loro offerti in qualsiasi linguaggio si desideri, purché esista un traduttore che effettui il mapping tra IDL e questo linguaggio; al momento OMG ha previsto un traduttore per il C e per il C++, ma si sta specificando anche il traduttore per SmallTalk.

Interfaccia per l'invocazione dinamica: il mapping delle dichiarazioni IDL in un altro linguaggio (ad esempio C++), permette ai clienti di richiedere servizi agli oggetti conosciuti, però alcune applicazioni devono essere in grado di richiedere servizi senza avere conoscenza, al momento della compilazione, dell'interfaccia degli oggetti che forniscono tali servizi. L'interfaccia per l'invocazione dinamica è in grado di "girare" qualsiasi richiesta effettuata da un cliente a qualunque oggetto. Questo è realizzato tramite l'interpretazione, al momento dell'esecuzione, dei parametri della richiesta e dell'identificatore dell'operazione.

Deposito delle interfacce: il deposito è concepito come estensione dell'interfaccia per l'invocazione dinamica, fornendo oggetti persistenti, che rappresentano informazioni sull'interfaccia di un oggetto. Con il deposito delle interfacce, un cliente è in grado di localizzare un oggetto sconosciuto in fase di compilazione, di chiedere la sua interfaccia e di costruire una richiesta da inoltrare tramite ORB.

BOA: è l'interfaccia usata per accedere alle funzionalità di ORB. Esporta le operazioni necessarie per creare dei riferimenti agli oggetti, per registrare ed attivare le implementazioni degli oggetti e per autenticare le richieste.

Esistono due modi per implementare il modello a materializzazione del canale utilizzando CORBA.
Il primo metodo consiste nello sfruttare IDL. Questa soluzione dal punto di vista dell'implementazione non si discosta molto da quanto presentato sinora (tralasciando la costruzione|gestione degli oggetti attivi, che è fornita direttamente da CORBA). Ovviamente si devono costruire le interfacce per i canali e per gli oggetti riflessivi. A tale proposito si sfrutta l'ereditarietà dell'interfacce offerta da IDL, ricordando che non necessariamente l'implementazione delle varie entità deve rispettare la relazione di ereditarietà esplicitata dalle interfacce, definendo l'interfaccia generale per gli oggetti istanza della classe ReflObj e per quelli istanza della classe channel e facendo in

modo che da queste interfacce ereditino tutte le interfacce dei vari oggetti riflessivi e dei vari tipi di canale.
A titolo di esempio si presenta l'interfaccia per gli oggetti istanza della classe Refl_Person.

```
interface person {
void incage();
void print_person();
};

interface refl_obj {
any send( inout reflobj s, inout reflobj r, in string kind, in string service, inout sequence<any> args );
any apply( in string service, inout sequence<any> args );
};

interface refl_person : person, refl_obj {};
```
[28]

Però la localizzazione degli oggetti attivi è lasciata ad ORB, i metodi send e apply non necessitano dei tid degli oggetti, come in PVM, ma usano direttamente dei riferimenti agli oggetti forniti da CORBA.
Inoltre, come si vede di seguito, il metodo apply sfrutta l'interfaccia per l'invocazione dinamica, quindi il riferimento al metodo riflessivo è rappresentato da una stringa che serve per determinare il servizio richiesto, mentre i parametri del metodo sono definiti di tipo *any*, che è un tipo generico offerto da IDL ed ha le stesse caratteristiche del *void ** del C++.

La maggioranza delle differenze con l'implementazione in PVM riguardano il servizio|metodo apply. Per la sua implementazione si sfrutta l'interfaccia per l'invocazione dinamica, in questo modo i canali possono eseguire i metodi dell'oggetto riflessivo senza conoscerli al momento dell'esecuzione. Ne consegue una semplificazione del metodo apply. Quest'ultimo può recuperare a run-time l'interfaccia dell'oggetto riflessivo, a cui appartiene, dal deposito delle interfacce e quindi valutare ed attivare nel modo corretto il metodo richiesto. Questo si effettua eseguendo i seguenti passi:

- apply richiede l'interfaccia dell'oggetto riflessivo di cui è parte (*get_interface*).
- Dall'interfaccia si ottengono le informazioni riguardanti le operazioni dell'oggetto (*describe_interface*).
- apply crea un oggetto richiesta per l'operazione desiderata (*create_request*), sfruttando l'interfaccia recuperata e la stringa che gli viene passata come richiesta di servizio.
- apply aggiunge alla richiesta ogni argomento necessario (*add_arg*) prelevandoli dal suo parametro args.

[28] Si sfrutta l'ereditarietà multipla delle interfacce, presente, con alcune restrizioni, dalla versione 1.2 di CORBA [Corba 93].

- apply effettua effettivamente la richiesta (*invoke*).

Il secondo meccanismo invece consiste, nell'incapsulare ORB in modo che ogni richiesta venga filtrata e quindi inoltrata ai canali, che effettuano la computazione riflessiva. Questo meccanismo è analogo al modificare la macchina virtuale di SmallTalk in modo che i messaggi vengano inoltrati direttamente ai canali.
Tale comportamento aumenta la trasparenza dell'uso delle riflessioni e la leggibilità delle applicazioni però va contro quanto ci si è prefissi, cioè di introdurre le riflessioni sull'architettura e non dentro la stessa. È riportata solo per sottolineare la flessibilità di CORBA e la facilità con cui può essere modificato.
La prima soluzione per introdurre le riflessioni ed, in particolare, il modello a materializzazione del canale in CORBA sottolinea la duttilità del meccanismo presentato che con poche modifiche, può essere facilmente introdotto in tutti i linguaggi, in particolare nei linguaggi concorrenti basati sul C++ (come visto per CORBA).

In questo capitolo si è esaminato come adeguare il meccanismo presentato nel capitolo [2] per introdurre le riflessioni, seguendo il modello a materializzazione del canale, in un linguaggio concorrente derivato dal C++. Si è esaminata l'implementazione di un linguaggio riflessivo, derivato dal C++, secondo il modello a meta-oggetto (OpenC++).
Con questo capitolo si è conclusa l'escursione nei meccanismi per introdurre le riflessioni, nella seconda parte si esamina un tipo di applicazioni (Fault Tolerance), in cui l'uso della riflessione può portare dei benefici.

Parte II
Riflessioni e Fault Tolerance

4 Fault Tolerance

4.1 Introduzione

Applicazioni sviluppate per determinate aree, come la difesa, i trasporti e la medicina necessitano di essere affidabili, cioè di garantire il conseguimento del risultato corretto, anche in presenza di situazioni anomale, dovute al verificarsi di errori sia hardware che software.
Applicazioni aventi tale proprietà sono dette applicazioni fault tolerant o tolleranti i guasti.

def. Un'applicazione fault tolerant è un sistema computazionale in grado di recuperare errori causati dalle proprie componenti, hardware o software, fornendo, così un servizio affidabile.

L'affidabilità di tali applicazioni è una proprietà molto importante; per rendersene conto è sufficiente immaginare a quali disastri può andare incontro un aereo affidato ad un sistema di controllo di volo non fault tolerant.
È importante sottolineare che la costruzione di software fault tolerant non è un mezzo sostitutivo per le tecniche di ingegneria del software, come le specifiche formali, le prove di correttezza ed il testing esaustivo, che sono usate per eliminare gli errori di progettazione (tecniche di faut avoidance), ma è loro complementare in quanto, serve a trattare situazioni insolite dovute ad errori non riscontrabili in fase di progettazione o di testing, cioè, tende ad arginare le mancanze di progettazione, rendendo stabile il software in attesa di una sua revisione.
Solitamente, applicazioni fault tolerant presentano al loro interno componenti duplicate, che adempiono allo stesso scopo; lo sforzo produttivo necessario per produrre queste componenti duplicate e per farle interagire sarebbe evitato se le tecniche di ingegnerizzazione del software fornissero uno strumento perfetto, in grado di eliminare tutti gli errori di un progetto.
Nel resto del capitolo, si classificheranno i tipi di errori in cui si può incorrere ed i possibili approcci usati nella costruzione di applicazioni fault tolerant.

4.2 Classificazione degli Errori

Il fallimento di un sistema è dovuto al comportamento erroneo di una sua componente. Questa componente può essere sia hardware che software.
Il primo passo per trattare applicazioni fault tolerant è quello di definire i comportamenti erronei e quelli corretti. A tale scopo si segue la classificazione dettata in [Ezhilchalvan-Shrivastava 86], che adotta i concetti di valore atteso, secondo le specifiche fornite, e di intervallo di risposta, cioè l'intervallo di tempo in cui, sempre secondo le specifiche, è accettabile la risposta.

def. Risposta corretta di una componente:
Sia v_j il risultato al tempo t_j di un input, che prevede una risposta non nulla, ricevuto al tempo t_i da una componente; tale responso è corretto se e solo se:

i) il valore è corretto:
$v_j = w_j$
dove w_j è il valore atteso, consistente con la specifica, e

ii) la risposta è corretta:
$t_j = t_i + t_d + \Delta t$
dove t_d è il minimo ritardo di risposta della componente, Δt è un ritardo impredicibile tale che $0 \leq \Delta t \leq t_{max}$ e t_{max} è il massimo ritardo impredicibile della componente.

def. una componente fallisce quando il suo comportamento devia da quello specificato.

In base alla definizione di risposta corretta ed ad alcune osservazioni, che si riportano di seguito, Ezhilchalvan e Shrivastava hanno classificato i diversi tipi di fallimento, in cui una componente può incorrere e, conseguentemente, le condizioni che hanno generato tale fallimento. Si noti che tali definizioni sono applicabili indifferentemente a componenti sia hardware che software. Gli autori non hanno fornito una definizione di risposta scorretta, in quanto tale definizione è la negazione di quella di risposta corretta.
Dalla definizione di risposta corretta si possono estrapolare alcune osservazioni:

- La risposta non è istantanea, ma avviene come minimo dopo un tempo t_d.
- Una componente non deve emettere risposte in modo arbitrario, ma deve essere stata sollecitata da una richiesta in input (con opportune eccezioni, quali i temporizzatori ed i watchdog, che una volta che iniziano a fornire risposte continuano a farlo senza ulteriori sollecitazioni, ma tali casi non sono contemplati da questo modello).

- Una risposta costituita da un valore corretto, ma fornito in ritardo o in anticipo rispetto a quando è atteso, oppure costituita da un valore sbagliato, fornito al momento richiesto, è considerata scorretta.

Di seguito è riportata la classificazione dei tipi di fallimento e delle condizioni di errore che li hanno generati:

- fallimento temporale: la componente produce il valore atteso, o troppo presto o troppo tardi, la causa è detta errore temporale

  i) $v_j = w_j$, e
  ii) $t_j < t_i + t_d$ o $t_j > t_i + t_d + t_{max}$

- fallimento puntuale: la componente produce, nell'intervallo di tempo specificato, un valore che non è quello atteso, la causa è detta errore puntuale

  i) $v_j \neq w_j$, e
  ii) $t_j = t_i + t_d + \Delta t$

- fallimento di commissione: la componente produce un valore scorretto e|o in ritardo|anticipo, la causa è detta errore di commissione

  i) $v_j \neq w_j$, e|o
  ii) $t_j \neq t_i + t_d + \Delta t$

- fallimento di omissione: la componente non produce alcuna risposta, la causa è detta errore di omissione

Risulta evidente che gli elementi di questa classificazione sono tra loro in relazione, in particolare, l'errore di omissione è sia un errore temporale che un errore puntuale, a loro volta, sia l'errore temporale che quello puntuale sono entrambi errori di commissione. Quindi un'applicazione fault tolerant in grado di trattare un fallimento di commissione è in grado di trattare qualsiasi errore.

Gli errori oltre ad essere classificati come descritto possono anche appartenere a due ulteriori classi, completamente ortogonali alle precedenti:

- errori permanenti: ritrovandosi successivamente nelle stesse condizioni in cui precedentemente si è verificato l'errore, questo continua a presentarsi (ad esempio un errore di divisione per zero, al ripresentarsi delle condizioni che lo hanno generato, cioè stesso dividendo e stesso divisore, continuerà a verificarsi).
- errori transienti: ritrovandosi successivamente nelle stesse condizioni in cui precedentemente si è verificato l'errore, questo non si verifica più (ad esempio un errore di lettura dovuto ad un calo improvviso di tensione, nella lettura successiva l'errore probabilmente sparirà).

Il comportamento delle componenti hardware è detto ***fail-silent*** quando l'unico tipo di fallimento che le può coinvolgere è il fallimento di omissione, ed è detto ***fail-uncontrolled*** o ***bizantino*** quando può essere soggetto ad uno qualsiasi dei tipi di fallimento descritti.

4.3 Aree di Interesse della Fault Tolerance

La fault tolerance dispone di due naturali campi di applicazione: l'hardware ed il software.
Questi ultimi danno luogo a due differenti discipline: l'hardware fault tolerance e la software fault tolerance.
Entrambe hanno come finalità la tolleranza dei guasti ed il loro recupero; tuttavia utilizzano tecniche e studiano situazioni d'errore assai differenti.
Le strategie che perseguono la tolleranza dei guasti sono implementate operando azioni sull'hardware e sul software.
È ragionevole che guasti a componenti hardware siano gestiti tramite dell'hardware aggiuntivo (circuiti ridondanti, memoria replicata, ...) e che guasti a componenti software siano gestiti tramite componenti software.
Dal momento che un sistema software è ospitato da una struttura hardware, fallimenti dell'hardware inficeranno anche l'elaborazione del software. Quindi la tolleranza a guasti hardware può, anche, essere ottenuta agendo sul software.
Ricapitolando la hardware fault tolerance si occupa di tollerare guasti prettamente hardware usando dell'hardware, mentre la software fault tolerance sfrutta il software per tollerare guasti sia software che hardware.

4.4 Classificazione della Software Fault Tolerance

Le tecniche di fault tolerance possono essere suddivise in due classi hardware e software fault tolerance.
L'hardware fault tolerance, coinvolgendo esclusivamente hardware (sia nel problema che nella soluzione), non interessa questo lavoro, quindi non viene ulteriormente trattata.
Di seguito si approfondisce la software fault tolerance presentando alcune delle tecniche più usate.

Randell [Randell 75] ha definito la <u>software fault tolerance</u> come <u>un insieme di tecniche, pensate per assicurare che un sistema adempia allo scopo per cui è stato costruito, nonostante errori software, hardware o chiamate illecite alle sue funzioni</u>.
Seguendo tale definizione la software fault tolerance può essere classificata in due categorie: system e software fault tolerance.

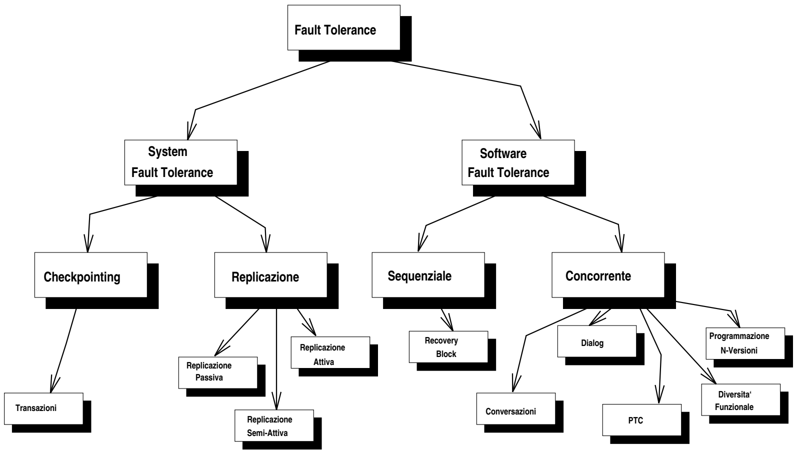


Figura 4.1: **Schema Classificazione Fault Tolerance**

Software Fault Tolerance: include i meccanismi necessari per trattare gli errori dovuti al software, solitamente i metodi usati si basano sulla ridondanza del software, cioè le componenti del software fault tolerant vengono sviluppate in più versioni differenti e tali versioni cooperano per il conseguimento del fine comune, tale meccanismo può essere adottato sia in un ambiente sequenziale che in uno parallelo|distribuito.

System Fault Tolerance: include meccanismi capaci di garantire l'esecuzione di un programma applicativo nonostante il verificarsi di errori hardware transienti o permanenti (gli errori software non vengono trattati). La gestione di tali situazioni può essere ottenuta per replicazione o per salvataggio dello stato; il primo caso consiste nell'esecuzione su macchine distinte, di copie identiche in modo da ridurre la probabilità che un crash di sistema impedisca alla computazione di giungere a compimento, mentre nel secondo metodo i cambiamenti di stato vengono trattati come delle azioni atomiche (cioè di tipo transazionale), che possono solo andare a buon fine o abortire.

Randell ha, inoltre, affermato che il principale veicolo per ottenere la tolleranza dei guasti è la ridondanza. Questa può presentarsi sotto diverse forme, le principali sono:

① Replica di software identico[1].

[1] Con software identico si intende software composto dalla stessa sequenza di codice.

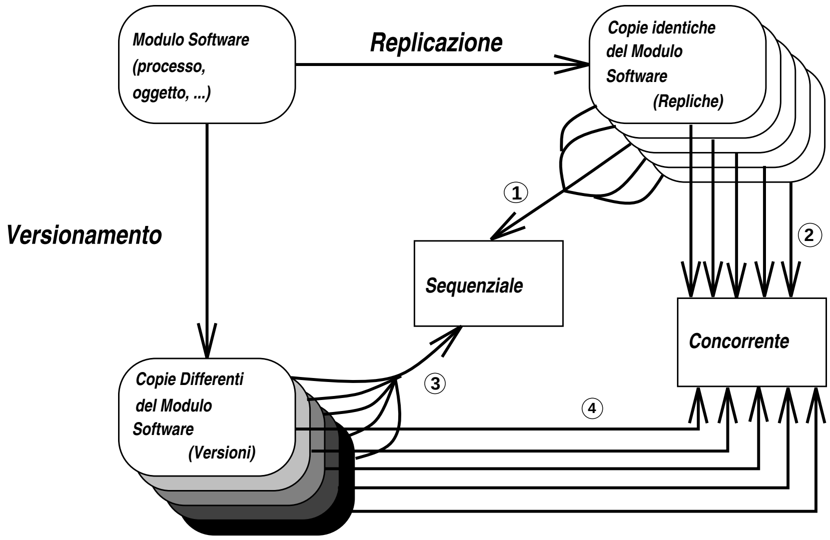


Figura 4.2: **Tassonomia della Software Fault Tolerance**

② Replica di software diverso[2].

Le attività svolte dalle repliche|versioni possono essere compiute sia concorrentemente, cioè con attività operanti contemporaneamente, che in modo sequenziale, cioè eseguendo in modo sequenziale le parti computazionali delle singole repliche|versioni. Dalla situazione descritta, che può essere schematizzata come in figura [4.2], si possono distinguere quattro approcci alla tolleranza ai guasti:

① Software identico eseguito in modo sequenziale sulla stessa macchina. Questo approccio è scarsamente interessante perchè moduli software uguali presentano gli stessi errori quando eseguiti sotto le stesse condizioni (ripristino dello stato e stesso hardware).
L'unico tipo di errori che si è in grado di gestire con questo approccio sono gli errori hardware transienti. Esempi di tale approccio sono rappresentati dagli schemi di recupero basati su checkpoint dello stato [Mullender 89].

② Software identico eseguito in modo concorrente su diverse macchine. Tecniche fault tolerant rappresentanti questo approccio sono la replicazione attiva ([4.4.1.2]), semi-attiva ([4.4.1.3]) e passiva ([4.4.1.1]).

③ Software diverso eseguito in modo sequenziale su un'unica macchina. Un esempio di questo approccio è il recovery block ([4.4.2.1.1]).

[2] Con software diverso si intende software composto da istruzioni, strutture dati ed algoritmi distinti, ma funzionamento equivalente.

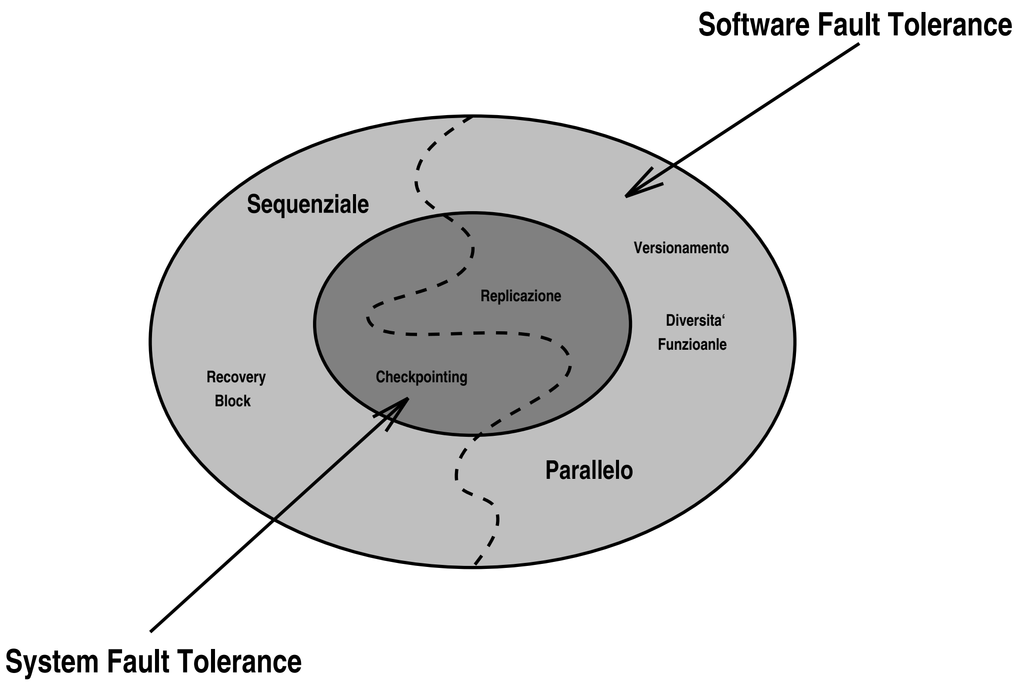


Figura 4.3: **Relazione tra Tecniche di System e Software Fault Tolerance**

④ Software diverso eseguito in modo concorrente su diverse macchine. Un esempio di questo approccio è la programmazione a N-Versioni ([4.4.2.2.2]).

Vediamo che tipi di errore risultano gestibili dagli approcci che sono stati classificati sopra.
I primi due approcci risultano adatti a recuperare tramite il software comportamenti erronei dovuti a fallimenti dell'hardware (system fault tolerance). In particolare il primo approccio, come già affermato, risulta essere utile per gestire errori hardware transienti, mentre il secondo approccio è usato per gestire errori di progettazione hardware.
Per gestire errori hardware permanenti occorre usare la seconda tecnica associata ad una diversificazione dell'hardware. Senza la diversificazione dell'hardware utilizzando tale tecnica non si ottiene alcun beneficio. Si consideri, ad esempio, i processori *pentium*, rilasciati l'anno scorso, che presentano un "bug" (errore permanente) nel processore matematico e che quindi non sono in grado di effettuare calcoli corretti su certi input. Usando un certo numero di tali *pentium* per replicare lo stesso gruppo di processi, se questi falliscono su uno di questi processori a causa di tale bug, allora tali processi falliscono su tutti i *pentium* perchè l'errore si ripresenta sistematicamente al ripresentarsi delle condizioni che lo hanno generato (cosa che accade sicuramente perchè il software e l'hardware che vengono utilizzati non cambiano).

I restanti due approcci si rivelano adatti a gestire errori software (software fault tolerance). Inoltre il quarto può essere adottato anche per gestire guasti hardware transienti (il guasto può coinvolgere una sola macchina permettendo alle rimanenti di raggiungere il risultato corretto).
Sia il terzo che il quarto approccio, comunque, possono essere adottati per gestire guasti hardware.
In particolare, il primo ed il secondo approccio possono essere visti come delle varianti, rispettivamente, del terzo e del quarto approccio, in cui la variante è rappresentata proprio dalla diversificazione del software che caratterizza la software fault tolerance.

Da quanto affermato, si osserva che le tecniche di software fault tolerance permettono di gestire errori non gestibili con tecniche di system fault tolerance (errori software), mentre gli errori gestiti dalle tecniche di system fault tolerance possono essere gestiti, con le stesse ipotesi (come ad esempio la diversificazione dell'hardware), da tecniche di software fault tolerance. Si può, quindi, classificare le tecniche di system fault tolerance, in base al tipo di errore che gestiscono, come un sottoinsieme proprio delle tecniche di software fault tolerance (come è evidenziato in figura [4.3]), rimarcando ulteriormente la maggiore duttilità della software fault tolerance.

4.4.1 System Fault Tolerance

Le principali tecniche di system fault tolerance sono raggruppabili in due classi: schemi di checkpointing e schemi a replicazione.
I primi consistono nel salvare periodicamente lo stato della computazione, dati ed operazioni effettuate (magari in memoria di massa, meglio ancora se in modo stabile) ed in caso di fallimento ripristinare lo stato dell'applicazione all'ultimo stato salvato, che viene, quindi, considerato corretto, e continuare così la computazione.
Un esempio di tecnica basata su questo schema è rappresentato dalle <u>transazioni</u> (vedere ad esempio il quinto capitolo di [Mullender 89]), questo schema, in questa sede, non viene ulteriormente approfondito.
La seconda classe, invece di basarsi sulla ripetizione della computazione, si basa sulla replicazione della stessa in modo concorrente, cioè la computazione che si vuole rendere fault tolerant viene effettuata su più macchine; per poter replicare la computazione occorre replicare le informazioni necessarie alla computazione stessa.
Esempi di tecniche basate sulla replicazione sono la replicazione passiva, attiva e semi-attiva. Questo schema di tolleranza ai guasti serve, soprattutto, per ovviare ad errori hardware.

Per determinare il numero di repliche (e quindi di macchine) necessario per tollerare un certo numero di fallimenti è necessario effettuare delle ipotesi sul tipo di fallimento che può coinvolgere una macchina. Se le macchine possono fallire solo in modo "silenzioso" (comportamento fail-silent) allora per tollerare k fallimenti è sufficiente

disporre di *k+1* repliche, questo perché l'unico modo con cui può fallire è dovuto ad un errore di omissione, cioè non viene inviata alcuna risposta, quindi con *k+1* repliche si riesce a tollerare il fatto che k componenti non inviino alcuna risposta.
Se le macchine possono fallire in modo incontrollato allora per tollerare k fallimenti sono necessarie *2k+1* repliche. Il fallimento può manifestarsi, infatti, con risultati scorretti e quindi per decidere qual è il risultato corretto occorre che la maggioranza delle repliche sia d'accordo su un valore. Quindi, con k fallimenti, si può determinare il valore corretto se si hanno *k+1* repliche "sane" (da cui *2k+1* repliche necessarie).

Di seguito si definiscono alcuni concetti legati ai checkpoint; i checkpoint sono uno dei concetti di base delle tecniche di fault tolerance ed in particolare delle tecniche di checkpointing e sono comunemente usati sia dalle tecniche di system che software fault tolerance.

def. Checkpoint è l'insieme delle informazioni necessarie al ripristino dello stato di una componente software.

Se, ad esempio, la componente è un processo, il checkpoint contiene almeno: i dati inizializzati, i dati BSS e quelli allocati dinamicamente, lo stack del processo ed il contenuto dei registri e dello stack pointer. Spesso tale insieme di informazioni viene costruito a partire da una situazione iniziale, usando dei mini-checkpoint.

def. Mini-Checkpoint o delta è un checkpoint ridotto contenente soltanto le informazioni che sono state modificate dopo l'ultima azione di checkpoint effettuata.

Con azione di checkpoint si intende l'insieme di operazioni necessarie a costruire un checkpoint ed ad inviarlo alle repliche oppure a memorizzarlo su memoria di massa o su memoria stabile.

4.4.1.1 Replicazione Passiva

La replicazione passiva è una tecnica fault tolerant, basata sulla replicazione delle componenti software critiche, in cui solamente una delle repliche[3], detta replica primaria, elabora i messaggi di input e fornisce dei messaggi di output, ovviamente in assenza di fallimenti. Le altre repliche o repliche secondarie non elaborano i messaggi di input e non producono messaggi di output; il loro stato è, comunque, aggiornato regolarmente tramite azioni di checkpoint dalla copia primaria.
In caso di fallimento della copia primaria, una delle copie secondarie (ovviamente non fallita) viene promossa a copia primaria e può, così, completare la computazione iniziata dalla copia primaria.

[3] Qui, come nel prosieguo, con replica si intende un'unità software che partecipa alla computazione e può essere ad esempio un processo, un oggetto od un gruppo di oggetti|processi.

Di seguito viene presentato un algoritmo per la gestione della replicazione passiva; per approfondimenti si rimanda a [Speirs-Barret 89].
Si supponga di avere due componenti $\mathcal{C}$ e $\mathcal{S}$ che si vogliono rendere fault tolerant, replicandole rispettivamente n e m volte. Una richiesta di servizio da parte di $\mathcal{C}$ ad $\mathcal{S}$ viene evasa come segue:

① $\mathcal{C}$ costruisce un messaggio contente il suo stato e lo invia alle sue repliche $\mathcal{C}_1, \ldots, \mathcal{C}_n$ (azione di checkpoint) e quindi attende un segnale di acknowledge dalle repliche (figura [4.4.a]).

② $\mathcal{C}$ inoltra la richiesta di servizio ad $\mathcal{S}$ ed alle sue repliche $\mathcal{S}_1, \ldots, \mathcal{S}_m$. La copia primaria $\mathcal{S}$ nel momento in cui riceve la richiesta, inizia a soddisfarla, mentre le repliche si limitano, ovviamente una volta ricevuta, ad immagazzinare la richiesta in una coda di richieste (figura [4.4.b]).

③ Al completamento del servizio richiesto, $\mathcal{S}$ invia un messaggio contenente il proprio stato alle proprie repliche (azione di checkpoint) ed attende il loro messaggio di acknowledge, come fa $\mathcal{C}$ nel punto ① (figura [4.4.c]).

④ $\mathcal{S}$ invia a $\mathcal{C}$ ed alle sue repliche un messaggio contenente il risultato del servizio. Le repliche $\mathcal{C}_1, \ldots, \mathcal{C}_n$, ricevuto il messaggio lo immagazzinano in una coda contenente i risultati ricevuti dall'ultimo checkpoint (figura [4.4.d]).

Quando fallisce la replica primaria, sia che essa sia $\mathcal{C}$ o $\mathcal{S}$, una delle repliche secondarie viene promossa a replica primaria ed a partire dall'ultimo checkpoint ricevuto dalla copia primaria, grazie alle code dei servizi richiesti e dei risultati ottenuti, riesce a riportarsi nello stato in cui era la copia primaria prima di fallire.
Ad ogni azione di checkpoint dello stato della copia primaria, anche le code dei servizi richiesti e dei risultati ottenuti, vengono aggiornate in accordo con il contenuto del checkpoint (questo è reso possibile dal fatto che tale messaggio contiene anche le informazioni per identificare i messaggi obsoleti contenuti nelle code). La presenza delle code delle richieste dei servizi e dei risultati ottenuti permette alle repliche secondarie di riprendere la computazione senza dover richiedere nuovamente ad altre entità i servizi già richiesti dalla copia primaria dopo l'ultima azione di checkpoint. In questo modo è possibile risparmiare tempo di esecuzione, evitare di chiedere nuovamente computazioni la cui azione non può essere facilmente ripetuta (come la richiesta di un input esterno) e di evitare l'effetto domino (vedere la sezione [4.4.2.2]).

4.4.1.2 Replicazione Attiva

La replicazione attiva è una tecnica di fault tolerance, basata sulla replicazione delle componenti software critiche, in cui tutte le repliche elaborano tutti i messaggi di

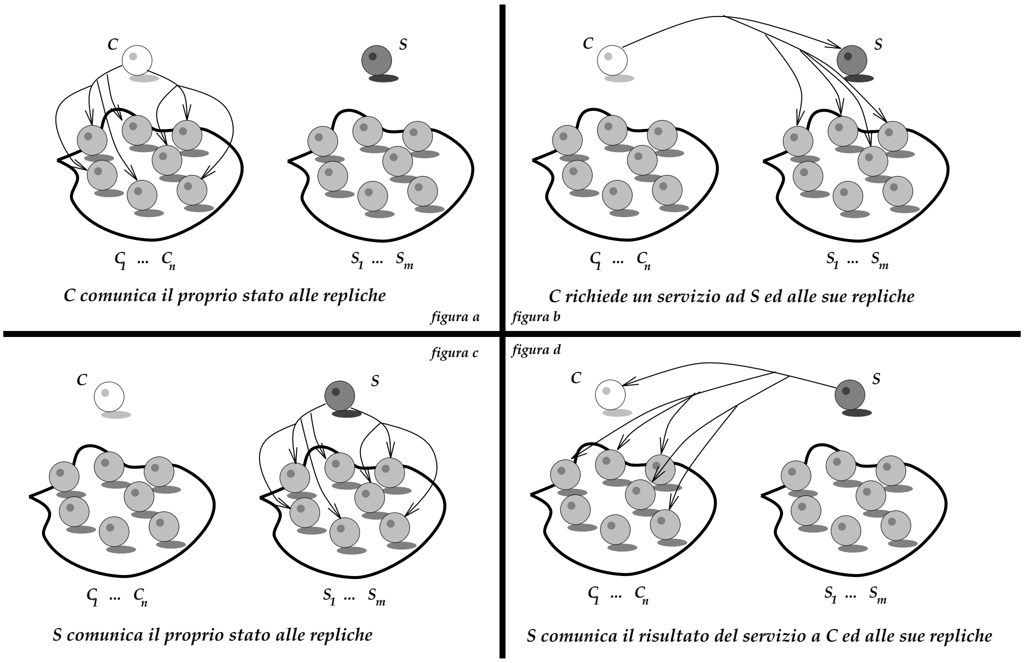


Figura 4.4: **Replicazione Passiva applicata ad una richiesta di servizio**

input concorrentemente, così che lo stato interno delle repliche, in assenza di fallimenti, sia sempre consistente con quello della copia primaria (a tale proposito si rende necessario un meccanismo di sincronizzazione inter-replica).
In assenza di fallimenti tutte le repliche forniscono lo stesso risultato, come tale si può scegliere in diversi modi quale risultato considerare. Se le macchine che si considerano hanno un comportamento fail-silent, allora come output si può considerare l'output generato da una replica qualsiasi (se la macchina fallisce, allora la replica non rende alcun output). Se, invece, le macchine che si considerano possono fallire in modo incontrollato, allora l'output viene scelto in seguito ad un confronto, come accade nella programmazione a N-Versioni (vedere la sezione [4.4.2.2.2]).
In entrambi i casi lo schema prevede un'ulteriore entità che gestisce gli output delle repliche attive, fornendoli, quindi, al cliente (questa entità, a seconda delle ipotesi, restituisce il primo risultato ottenuto od il risultato su cui concordano, ad esempio, la maggioranza delle repliche).
Di seguito si illustra un possibile algoritmo per la gestione della replicazione attiva (per maggiori dettagli vedere [Chérèque-Powell-Reynier-Richier-Voiron 92] e [Fabre-Nicomette-Pérennou-Wu 94]).
Si supponga di avere un'entità $\mathcal{C}$ che richiede un servizio ad un'entità critica $\mathcal{S}$, questa viene replicata, secondo lo schema a replicazione attiva, n volte. Un'entità $\mathcal{V}$ ha la funzione di voter. Per semplicità si trascura la possibilità che le entità $\mathcal{C}$ e $\mathcal{V}$

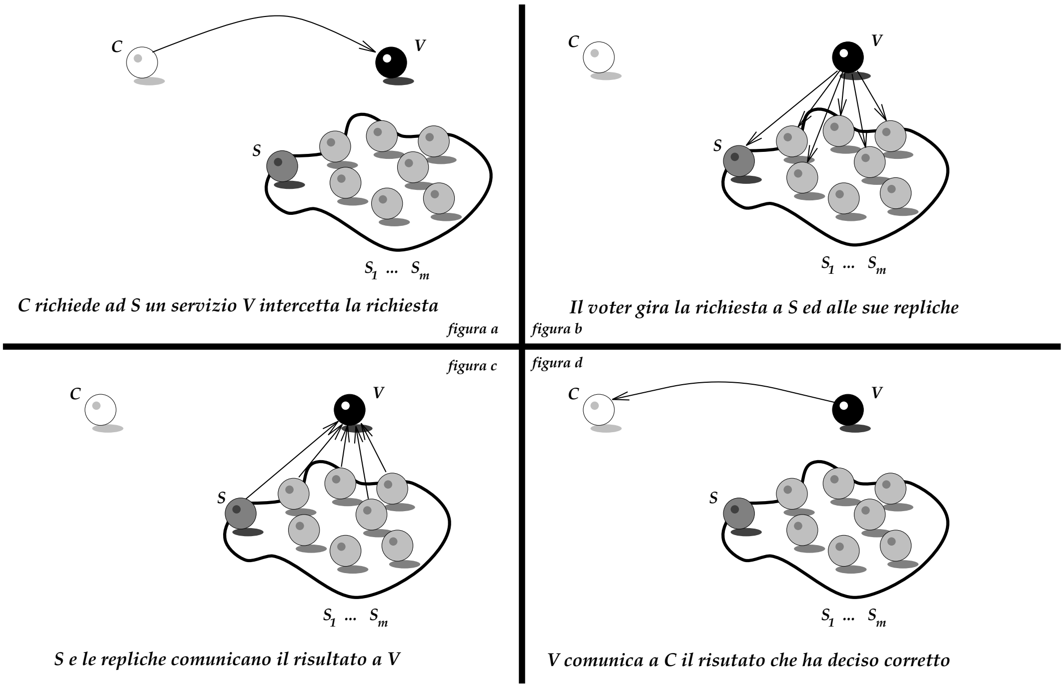


Figura 4.5: **Replicazione Attiva applicata ad una richiesta di servizio**

possano fallire (la tolleranza al loro fallimento, comunque, può essere ottenuta con una qualsiasi delle tecniche presentate in questo capitolo).
Quindi la richiesta da parte di $\mathcal{C}$ di un servizio ad $\mathcal{S}$ viene svolta come segue:

① $\mathcal{C}$ inoltra ad $\mathcal{S}$ la sua richiesta. Questa viene però intercettata da $\mathcal{V}$ (figura [4.5.a]).

② $\mathcal{V}$ inoltra la richiesta di $\mathcal{C}$ ad $\mathcal{S}$ ed alle sue repliche $\mathcal{S}_1, \ldots, \mathcal{S}_n$ (va notato che con la replicazione attiva il comportamento dell'oggetto critico è identico a quello delle repliche) (figura [4.5.b]).

③ Tutte le repliche attive di $\mathcal{S}$ ($\mathcal{S}$ compreso) elaborano la richiesta pervenuta, quindi, inviano a $\mathcal{V}$ il risultato di tale elaborazione (figura [4.5.c]).

④ $\mathcal{V}$ decide qual è il risultato corretto e lo invia a $\mathcal{C}$.

Terminata una computazione, le diverse repliche, che hanno manifestato un comportamento erroneo, dovrebbero essere ripristinate ad uno stato consistente con la decisione presa dal voter. Questo, normalmente, viene fatto tramite protocolli di interazione tra repliche, che non vengono ulteriormente esaminati, rimandando il loro approfondimento alla letteratura sull'argomento.
In questo schema tutte le repliche, compresa l'entità critica, sono uguali; le repliche,

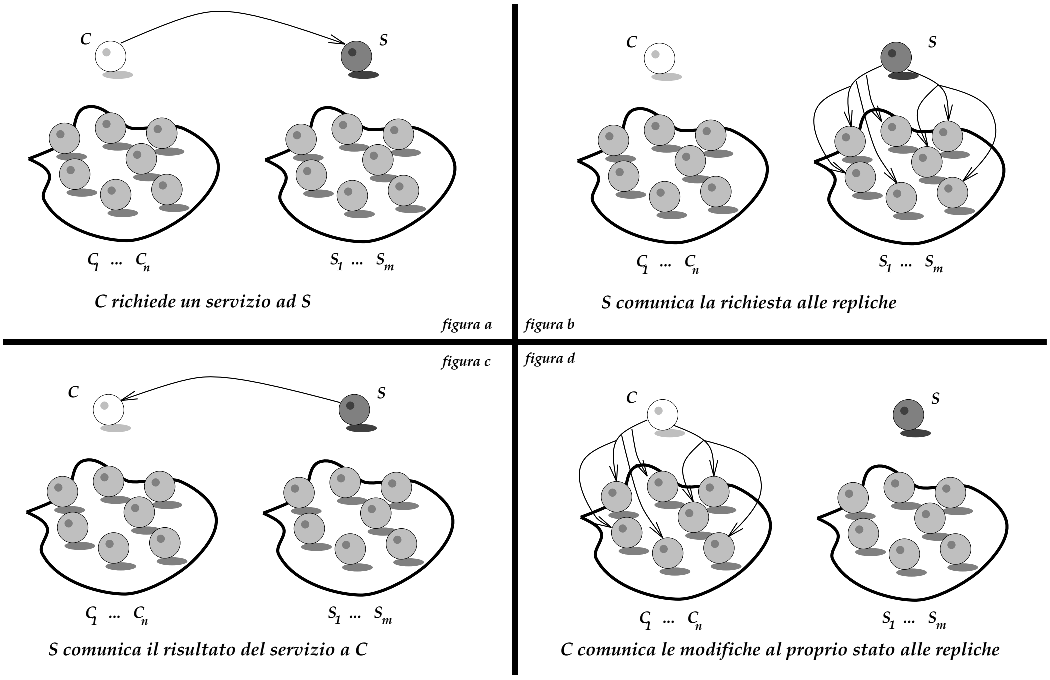


Figura 4.6: **Replicazione Semi-Attiva applicata ad una richiesta di servizio**

essendo attive ed identiche, avanzano tutte contemporaneamente ed, in assenza di errori, nello stesso modo.
Eventuali comportamenti erronei vengono rilevati dal voter, il quale decide le azioni da intraprendere per il corretto funzionamento della richiesta successiva (può ad esempio sostituire le repliche con nuove repliche, il cui stato è inizializzato allo stato delle repliche che hanno avuto un comportamento corretto oppure può semplicemente effettuarne l'aggiornamento).

4.4.1.3 Replicazione Semi-Attiva

La replicazione semi-attiva è una tecnica di fault tolerance basata sulla replicazione, che può essere pensata come un ibrido della replicazione attiva con quella passiva. Solo una delle repliche, la replica primaria, elabora i messaggi di input e fornisce dei messaggi di output, ovviamente in assenza di errori, le altre repliche, repliche secondarie, non producono messaggi di output; il loro stato interno è aggiornato tramite la diretta elaborazione delle richieste. I risultati dei servizi richiesti ad altre entità vengono fornite alle repliche secondarie tramite il mini-checkpoint da parte della replica primaria.
Questo schema consiste, praticamente, nel tenere delle repliche passive delle entità critiche, il cui stato è mantenuto consistente con quello della replica primaria in

modo “attivo”, cioè facendo elaborare, ad ogni replica, ogni messaggio.
Per soddisfare una richiesta di un servizio inoltrata da un'entità $\mathcal{C}$ ad un'altra entità $\mathcal{S}$ replicate rispettivamente m e n volte, si seguono i seguenti passi:

① $\mathcal{C}$ inoltra la propria richiesta di servizio a $\mathcal{S}$ (figura [4.6.a]).

② $\mathcal{S}$ inoltra la richiesta di $\mathcal{C}$ alle sue repliche, $\mathcal{S}_1, \ldots, \mathcal{S}_\mathrm{n}$ (figura [4.6.b]).

③ $\mathcal{S}$, elaborata la richiesta, invia a $\mathcal{C}$ il risultato. In parallelo con la computazione di $\mathcal{S}$, $\mathcal{S}_1, \ldots, \mathcal{S}_\mathrm{n}$ elaborano anch'esse la richiesta elaborata da $\mathcal{S}$. Questa fase potrebbe necessitare una sincronizzazione tra le repliche $\mathcal{S}$, $\mathcal{S}_1, \ldots, \mathcal{S}_\mathrm{n}$ in modo che la loro esecuzione avanzi di pari passo (figura [4.6.c]).

④ $\mathcal{C}$ ricevuto da $\mathcal{S}$ il risultato del servizio richiesto, costruisce un mini-checkpoint contenente solo il valore ricevuto e lo invia alle proprie repliche (figura [4.6.d]).

Si può osservare che l'algoritmo è analogo a quello presentato per la replicazione passiva, unica variazione è il progredire dello stato delle repliche secondarie concorrentemente a quello della replica primaria e nell'uso di mini-checkpoint, che sostituiscono i checkpoint, che devono per forza essere utilizzati quando lo stato viene modificato in seguito a computazioni svolte da altre entità (come si è visto nel caso di $\mathcal{C}$).
Lo schema di replicazione semi-attiva risulta adatto a gestire comportamenti erronei di tipo fail-silent. Questo meccanismo può gestire anche comportamenti erronei non-fail-silent, però in questo caso, si dovrebbe introdurre un voter per determinare la correttezza del risultato, rendendolo, quindi, completamente analogo allo schema a replicazione attiva. In assenza del voter le repliche evolverebbero in modo eventualmente incontrollato, introducendo durante le elaborazioni errori di cui si perdono le tracce.
Il motivo che fa preferire l'uso di questo schema a quello a replicazione passiva è l'efficienza, infatti spesso lo stato di un oggetto o comunque le informazioni contenute in un checkpoint sono molto grosse e quindi comunicarle alle repliche implica un overhead che inficia le prestazioni del sistema.

4.4.2 Software Fault Tolerance

Nella software fault tolerance la tecnica base, come già nella system fault tolerance, consiste nell'introdurre nel sistema la ridondanza delle componenti, in modo da ottenere un'esecuzione corretta nonostante il fallimento di una componente. Naturalmente, il concetto di ridondanza, come riproduzione di copie perfette dello stesso oggetto, può funzionare solo con errori hardware, mentre nel caso di errori software, sfocierebbe semplicemente nella ripetizione dello stesso errore (se non si cambia alcunché, le condizioni logiche che hanno portato all'errore si riproporranno invariate).

Quindi al concetto di ridondanza si deve affiancare quello di diversità computazionale, cioè non si hanno più semplici copie identiche, ma copie simili, cioè utilizzabili nello stesso modo, ma che raggiungono lo scopo tramite algoritmi diversi; in questo modo l'esecuzione di un'alternativa non dovrebbe più incorrere nell'errore in cui era incorsa la componente primaria.
Le tecniche usate nella costruzione di software fault tolerant sono state sviluppate partendo dall'osservazione pessimistica che: "grandi progetti, sicuramente, presentano degli errori di progettazione residui"; conseguentemente le tecniche usate si basano sulla ridondanza, in virtù dell'osservazione, questa volta ottimistica, che: "applicazioni progettate indipendentemente, per risolvere lo stesso problema, presentano errori differenti". La sensatezza di tali ipotesi è stata oggetto di molti studi, a volte anche con risultati contrastanti. In questa sede non si è interessati ad approfondire tale questione, rimandando per approfondimenti alla letteratura ed in particolare a [Avižienis 85], [Kersken-Saglietti 92] e [Kim 82a].
La software fault tolerance può essere utilizzata sia in ambiente sequenziale che parallelo|distribuito. Esempio di una tecnica di software fault tolerance adottata in ambiente sequenziale è il *recovery block*, mentre esempi di tecniche adottate in ambiente parallelo|distribuito sono le *conversazioni* e la *programmazione a N-Versioni* (NVP).

4.4.2.1 Software Fault Tolerance: caso sequenziale

Una routine fault tolerant in un ambiente sequenziale, solitamente, può essere suddivisa in tre fasi[4]:

① rilevazione dell'errore.

② ripristino di uno stato corretto, coincidente con lo stato, che si aveva prima dell'inizio della computazione.

③ esecuzione di routines alternative.

che solitamente danno origine ad uno schema simile a quello mostrato in figura [4.7], tale figura fa subito notare che il metodo, usato per risolvere situazioni errate, è

[4] Seguendo quanto detto in [Kim 82a], le fasi sarebbero quattro, cioè una volta rilevata la presenza di un errore nella computazione, si dovrebbe determinare se questo è dovuto all'hardware o al software, qui si considera esclusivamente che l'errore sia di origine software, quelli di origine hardware si suppongono trattati con tecniche di system fault tolerance (vedi sezione [4.4.1]).
La principale differenza tra errore hardware e software è che la routine alternativa nel primo caso può essere la stessa che ha fallito, nel secondo no, pena il ricadere nello stesso errore.

basato sulla ridondanza delle componenti (la routine principale e le i routines alternative eseguono lo stesso compito).
L'uso di questo meccanismo in un ambiente sequenziale, non è privo di complicazioni: innanzitutto porta ad un costo di sviluppo maggiore, in quanto si devono incaricare teams differenti (uno per ogni alternativa) di sviluppare il proprio algoritmo per risolvere il problema, i teams devono essere diversi e senza punti di contatto, questo nella speranza che persone diverse progettino soluzioni con bug diversi; inoltre un programma fault tolerant richiede una quantità di memoria maggiore (il codice delle alternative si va ad aggiungere al codice della routine principale), nonché un tempo di esecuzione più alto (si effettuano più test, ed in caso di comportamento errato si deve eseguire l'operazione di ripristino e la routine alternativa).
Il problema più evidente, comunque, è la scarsa leggibilità di un programma fault tolerant e di conseguenza la sua difficile mantenibilità.

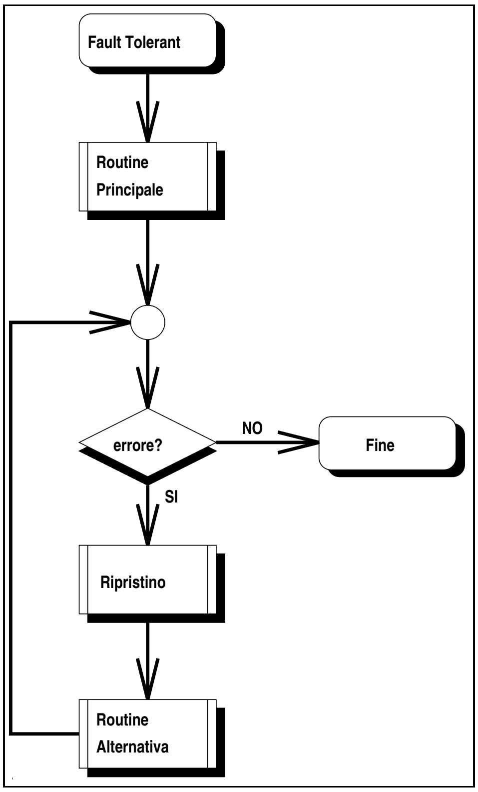


Figura 4.7: Schema routine Fault Tolerant

Un programma fault tolerant segue il seguente schema generale:

```
<routine principale>

if errore then
  begin
     <ripristino>
     <routine alternativa 1>
     if errore then
       begin
          <ripristino>
          <routine alternativa 2>
          if errore then
            begin
                 ⋮
              <ripristino>
              <routine alternativa n>
            end
       end
  end
```

Per chiarire i problemi che derivano dall'impiego di una tecnica di questo tipo si consideri la costruzione di una soluzione fault tolerant per il seguente problema:

dato un sistema di equazioni lineari Ax = b, trovare il vettore soluzione x.
I metodi usati sono i metodi classici dell'analisi numerica basati sulle fattorizzazioni LU e QR, che modificano la matrice dei coefficienti, quindi l'eventuale ripristino consiste nel riportare tale matrice nello stato precedente a quello dell'esecuzione del primo metodo. Come si può osservare dallo pseudocodice in figura [4.1], l'algoritmo è alquanto macchinoso; innanzitutto, occorre memorizzare l'ultimo stato corretto prima dell'inizio dell'esecuzione (nella fattispecie occorre conservare la matrice A nella matrice C), poi il ripristino deve essere effettuato manualmente (recupero di A da C), introducendo nell'algoritmo vero e proprio codice inutile al fine della risoluzione del problema; per ogni metodo alternativo occorre inserire il codice relativo al test, aumentando la dimensione dell'eseguibile e diminuendo la leggibilità del codice sorgente (con tante alternative si avrebbe un discreto numero di if annidati).

pseudo-codice 4.1 Algoritmo fault tolerant per la risoluzione di sistemi lineari

```
C := A                            → Salvataggio di A per il successivo ripristino
<metodo QR>                       → metodo principale
 if (|Cx-b| >ε) then              → test di accettazione, dove ε è il vettore tolleranza
    begin
      A := C                      → ripristino di A e x
      x := 0
      <metodo LU>                 → metodo alternativo
      if (|Cx-b| >ε) then         → test di accettazione, dove ε è il vettore tolleranza
           <sistema intrattabile>
    end
```

Quindi per migliorare la programmazione di applicazioni fault tolerant si è rivelato necessario un costrutto, pensato appositamente per migliorare la leggibilità dello stesso. Ci sono state diverse proposte, ma quella che ha avuto maggiore seguito è stata quella dello schema recovery block fatta da Randell in [Randell 75].

4.4.2.1.1 Schema Recovery Block Un recovery block è un costrutto linguistico, che supporta l'incapsulamento della ridondanza nel codice di un programma fault tolerant in modo coinciso e facile da leggere.
La sua sintassi base è riportata di seguito:

```
ensure <test di accettazione>
by <metodo primario>
else by <primo metodo alternativo>
  ⋮
else by <n-esimo metodo alternativo>
else error
```

Il test di accettazione è un'espressione logica atta a verificare la correttezza|accettabilità dei valori ottenuti, buoni test possono essere le funzioni inverse (se il metodo calcola $x = \sqrt{x}$ allora il test può essere $x^2 = \mathrm{Prior}(x)$ dove con Prior(x) si intende il valore che aveva x prima dell'inizio della computazione), controlli sui tempi di

risposta, o relazioni tra i valori vecchi e quelli nuovi, comunque, il test deve essere semplice e rapido in modo da influenzare il meno possibile il tempo di esecuzione. Il metodo primario ed i metodi alternativi sono tutti algoritmi destinati alla risoluzione dello stesso problema. Possono essere pensati con diverse strategie, ad esempio, come miglioramenti l'uno dell'altro (ogni metodo successivo raffina l'elaborazione a scapito del tempo di esecuzione, ottenendo risultati sempre più accurati) o come degradamento l'uno dell'altro (cioè in presenza di problemi particolarmente ostici, ogni metodo successivamente applicato fornisce soluzioni sempre più grossolane in tempi sempre più rapidi).
L'esecuzione di tale costrutto consiste nell'eseguire il metodo primario, e successivamente il test di accettazione, se questo non va a buon termine, si esegue il primo metodo alternativo, si rieffettua il test di accettazione, se questo nuovamente fallisce si esegue il metodo alternativo successivo, e così via finché un test di accettazione non viene passato, o non ci sono più metodi alternativi da applicare.
Questo schema apporta notevoli miglioramenti al codice in termini di leggibilità come si può osservare dall'esempio relativo alla risoluzione di sistemi lineari presentato in precedenza.

pseudo-codice 4.2 Sistemi Lineari con Recovery Block

```
ensure |Prior(A)x-b| > ε
by <metodo QR>
else by <metodo LU>
else error
```

Come si può notare in figura [4.2] il codice risulta più compatto; è lo schema di recovery block che si occupa del salvataggio e del ripristino dello stato, fornendo i meccanismi necessari per accedere, nel test di accettazione, ai vecchi valori (Prior(A)).
Questo esempio sottolinea la notevole astrazione del costrutto recovery block, che può essere anche annidato entro i metodi (sia primari che alternativi).
Il recovery block presenta alcuni limiti come il fatto che esistono alcune azioni i cui effetti non possono venire ripristinati automaticamente (ad esempio il lancio di un missile o molte applicazioni real-time) o l'impossibilità di effettuare controlli sullo stato in ingresso; queste sono limitazioni per il cui approfondimento si rimanda a [Kim 82a]. Il vero problema consiste nell'impossibilità di applicarlo in ambiente parallelo|distribuito a causa dell'effetto domino, discusso nel paragrafo seguente.

4.4.2.2 Software Fault Tolerance: caso concorrente

Si supponga di avere un'applicazione costituita da un insieme di processi cooperanti. La distribuzione e la concorrenza introducono nuove condizioni di errore e quindi

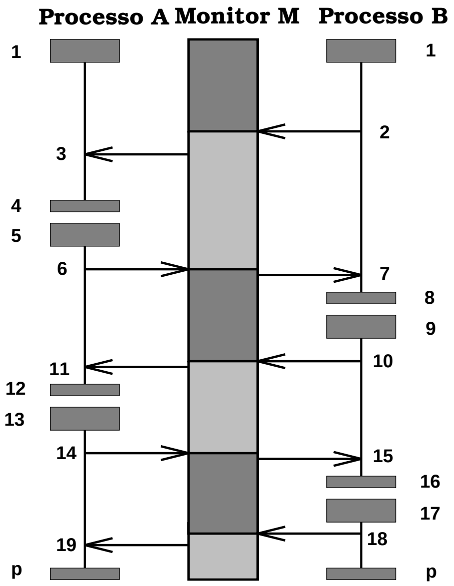


Figura 4.8: **Esempio di Contrabbando di Informazioni che Provoca l'Effetto Domino**

di fallimento, in particolare la possibilità di interazioni scorrette tra processi; il controllo di validità e l'eventuale ripristino devono essere ottenuti dal contributo "collettivo" di tutti i processi coinvolti.

Effetto Domino: Una trasposizione diretta, in ambiente concorrente, dello schema recovery block consiste nell'applicare lo schema ad ogni processo che è coinvolto nell'applicazione o per lo meno nella parte dell'applicazione che si vuole rendere fault tolerant.
Si considerino due processi $\mathcal{A}$ e $\mathcal{B}$ che progrediscono concorrentemente e comunicano tramite un monitor $\mathcal{M}$ [Hoare 74]. La loro interazione è raffigurata in figura [4.8]. In figura un cambiamento dell'ombreggiatura del monitor rappresenta un cambiamento dello stato dello stesso. Una freccia proveniente da un processo ed entrante nel monitor rappresenta un aggiornamento (ad esempio $\mathcal{B}.2$), mentre una freccia uscente dal monitor ed entrante in un processo rappresenta l'utilizzo da parte del processo delle informazioni contenute nel monitor (ad esempio $\mathcal{A}.3$). Una linea verticale limitata da due rettangoli, uno più sottile e l'altro più spesso rappresenta l'esecuzione di un recovery block; ogni rettangolo spesso rappresenta il punto di recupero, mentre il rettangolo sottile rappresenta il punto in cui si effettua il controllo sul fallimento del blocco (test di accettazione).
Si supponga che il processo $\mathcal{A}$ passi con successo il test di accettazione $\mathcal{A}.4$. L'esecuzione del recovery block che termina con il test di accettazione $\mathcal{A}.4$ non si può ancora

definire completamente valida, in quanto le informazioni ricevute in $\mathcal{A}.3$ dal monitor possono essere successivamente invalidate se il processo $\mathcal{B}$ non passa con successo il test di accettazione $\mathcal{B}.8$. Questo comporta che il punto di recupero stabilito in $\mathcal{A}.1$ deve essere mantenuto finché $\mathcal{B}$ non raggiunge e supera il test di accettazione in $\mathcal{B}.8$. Supposto, invece, che il processo $\mathcal{A}$ non passi il test di accettazione $\mathcal{A}.4$, questo può essere dovuto ad un errore proprio di $\mathcal{A}$ od ad un errore commesso dal processo $\mathcal{B}$ tra $\mathcal{B}.1$ e $\mathcal{B}.2$ e propagato ad $\mathcal{A}$. Se l'errore di $\mathcal{A}$ è dovuto al processo $\mathcal{B}$, allora $\mathcal{A}$ ha riscontrato un comportamento anomalo di $\mathcal{B}$, però non è in grado di discernerlo da un suo comportamento anomalo, perciò il fallimento in $\mathcal{A}.4$ comporta il rollback[5] di entrambi i processi.
Si consideri il caso in cui il processo $\mathcal{A}$ fallisca il test di accettazione $\mathcal{A}$.p. In questo caso $\mathcal{A}$ viene riavvolto al punto di recupero $\mathcal{A}.13$, però $\mathcal{A}$ aveva aggiornato il monitor in $\mathcal{A}.14$ e $\mathcal{B}$ aveva usato tali dati in $\mathcal{B}.15$, introducendo quindi nella sua computazione un errore potenziale, questo causa il rollback del processo $\mathcal{B}$ al punto di recupero $\mathcal{B}.9$. La propagazione dei rollback continua, per gli stessi motivi che hanno generato i primi, finché entrambi i processi non sono stati riavvolti al loro punto di partenza ($\mathcal{A}.1$ e $\mathcal{B}.1$). Una simile propagazione incontrollata dei rollback è nota come *effetto domino*. L'effetto domino è dovuto all'introduzione scoordinata del recovery block in processi cooperanti, che permette il propagarsi dell'errore tramite lo scambio di informazioni "non validate". Questo fenomeno è detto *contrabbando di informazioni*.
Ovviamente tale situazione è improponibile; i modi che tentano di risolvere tale problema sono sostanzialmente raggruppabili secondo due approcci: controllo dell'interazione tra processi, limitando i ripristini a situazioni intermedie dell'esecuzione, o evitare il problema alla radice, eseguendo in parallelo le copie e decidendo successivamente, quale risposta è quella più corretta (non si ha più la fase di ripristino). Esempi del primo approccio sono le <u>conversazioni</u>, il <u>dialogo</u> e il <u>Programmer Transparent Coordination (PTC)</u>, in questa sede si tratteranno esclusivamente le conversazioni rimandando a [Kim 82a] per gli altri modelli, mentre un esempio del secondo approccio è rappresentato dal <u>n-version programming o programmazione a n-versioni</u>.

4.4.2.2.1 Conversazioni Lo schema a conversazioni è essenzialmente un'estensione concettuale alla concorrenza dello schema a recovery block che, come visto, è un costrutto progettato per supportare il rilevamento e la gestione degli errori di un solo processo in modo strutturato.
Nel momento in cui un processo entra in una conversazione ne viene costruito il checkpoint che costituisce il punto di recupero di quel processo (l'insieme dei punti di recupero costituisce la linea di recupero dell'intero sistema). Per prevenire il

[5] Rollback: con tale termine si indica l'azione di ripristino ad uno stato sicuro dell'esecuzione di un processo.

contrabbando di informazioni il processo può comunicare esclusivamente con processi che sono già coinvolti nella conversazione. Quando tutti i processi hanno completato le loro operazioni, un test di accettazione globale (cioè un test di accettazione per ogni processo) viene eseguito (linea di controllo). Se esso fallisce (cioè se almeno un test di accettazione fallisce) tutti i processi sono ripristinati e si eseguono le rispettive alternative successive, se invece va a buon fine, tutti i processi escono sincronamente dalla conversazione. Tale meccanismo garantisce che nessun errore possa provocare un ripristino ad un punto antecedente ad un test di accettazione che è stato passato con successo. In questo modo, quando tutti i processi vengono ripristinati alla linea di recupero, non possono esserci ulteriori propagazioni dell'operazione di ripristino, né verso altri processi, né verso situazioni precedenti, evitando, così, l'effetto domino.
Ricapitolando lo schema delle conversazioni è uno schema a recovery block a cui si sono aggiunti i seguenti concetti:

- **linea di recupero:** insieme dei punti di ingresso nella conversazione dei processi, tali punti rappresentano gli stati corretti a cui i processi possono essere ripristinati

- **linea di controllo:** insieme di punti rappresentanti i test di accettazione di ogni processo

- **muri laterali:** linea immaginaria pensata per inglobare tutti i processi interagenti; i processi non possono superare tali linee comunicando con processi che non sono nel recovery block

L'esecuzione di una conversazione consiste nell'esecuzione dei processi al suo interno; nel momento in cui il flusso esecutivo, per ogni processo, raggiunge il test d'accettazione si effettua tale test. La differenza dal caso precedente è che il test è passato solo se tutti i test sono passati, nel caso un test fallisca tutti i processi che "conversano" sono riavvolti ognuno al proprio punto di recupero, appartenente alla linea di recupero e non è possibile risalire oltre tale linea perché è uno stato corretto; dopo il ripristino si ricomincia l'esecuzione usando per ogni processo le alternative definite.
Nella proposta originale dello schema a conversazioni, fatta da Randell in [Randell 75], è stata presentata solo la semantica, da allora diverse implementazioni sono state presentate (vedere, ad esempio, [Kim 82b] e [Kim 84]). Il meccanismo è comunque stato applicato pochissimo perché risulta essere complesso da implementare e riduce notevolmente il grado di concorrenza del sistema.

4.4.2.2.2 N-Version Programming (NVP) La programmazione a N-Versioni è una tecnica di software fault tolerance, sviluppata presso l'UCLA (Università della California e Los Angeles) dal gruppo di ricerca di Avižienis, come si può leggere in [Avižienis 85].

La programmazione a N-Versioni consiste nello sviluppo indipendente di $N \geq 2$ moduli software[6], detti *versioni*, a partire dalla stessa specifica.

Con sviluppo indipendente si intende che la fase di produzione delle versioni deve essere portata avanti da individui o gruppi di individui con scarsi o nulli punti di contatto. Ovviamente il modo migliore per sviluppare le versioni è quello di sfruttare per ognuna, algoritmi, linguaggi di programmazione, strumenti e ambienti di sviluppo differenti.
Lo scopo dello sviluppo indipendente delle versioni è quello di produrre versioni con una probabilità minore di presentare gli stessi errori.
La specifica del problema oltre a fornire una descrizione completa e non ambigua dei requisiti delle varie versioni, lasciando, al contempo, ampio spazio decisionale agli sviluppatori di ogni singola versione, deve descrivere come le diverse versioni possono cooperare per il conseguimento del risultato. Quindi la specifica deve descrivere sia l'aspetto funzionale[7] dell'applicazione, cioè l'opera espletata da ogni versione sia l'aspetto non funzionale[8] dell'applicazione, cioè deve stabilire come, quando e chi effettua il controllo sulla correttezza dell'elaborazione svolta dalle versioni.
Per questo motivo fa parte dell'applicazione anche un'altra entità logica il cui compito è quello di coordinare l'operato delle diverse versioni. Questa entità, detta *adjdicator* o *voter*, ogniqualvolta l'applicazione giunge ad un punto di controllo, elabora il proprio algoritmo di decisione[9] e stabilisce qual è il risultato corretto e come questo influenza il resto della computazione. Fa parte delle specifiche dell'applicazione anche il momento in cui questo giudizio viene espresso: ad esempio può essere dopo l'elaborazione di un certo numero di istruzioni, dopo il completamento di una transazione, dopo una comunicazione, dopo l'attivazione di un metodo od in altre situazioni ancora.

Confrontando la programmazione a N-Versioni con il recovery block o con il suo equivalente concorrente saltano fuori due differenze sostanziali:

① **Test di accettazione:** nell'approccio a recovery block è specifico dell'applicazione che deve essere implementata, mentre nella programmazione a N-Versioni, l'algoritmo di decisione è un algoritmo "generico" di votazione (ad esempio per maggioranza).

[6] Con modulo software si intende una componente dell'applicazione, come, ad esempio, un processo od un oggetto.

[7] Con funzionale si intende il fatto che la sua presenza è fondamentale per la risoluzione del problema.

[8] Con non funzionale si intende il fatto che la sua presenza serve per dotare l'applicazione di un certo comportamento (come la fault tolerance), marginale al fine della risoluzione del problema.

[9] Un algoritmo di decisione comunemente usato è quello per maggioranza. Questo non è l'unico, per avere una panoramica sugli algoritmi di decisione si rimanda a [Di Giandomenico-Stringini 90].

② **Esecuzione delle alternative:** nell'approccio a recovery block (ed analogamente nello schema a conversazione) le alternative vengono eseguite solo nel caso l'entità primaria fallisca, mentre, con la programmazione a N-Versioni, le alternative vengono eseguite sempre tutte concorrentemente.

Da queste osservazioni risulta che la programmazione a N-Versioni favorisce il riuso dell'entità che effettua il voting e quindi degli algoritmi che implementa. Inoltre si ha un risparmio di tempo in esecuzione, in quanto, nell'approccio a recovery block|conversazioni, se si incorre in un errore, occorre eseguire la fase di ripristino e quindi le alternative, mentre nella programmazione a N-Versioni si ha solo la fase di voting per stabilire la risposta corretta in quanto le alternative sono già state elaborate tutte.

4.4.2.2.3 Diversità Funzionale Sperimentalmente si è visto che il partire da un'unica specifica può condurre agli stessi errori di progettazione. In seguito a questa osservazione risulta che la programmazione ad N-Versioni non garantisce il raggiungimento del livello di affidabilità richiesto dal sistema. A tale proposito al versionamento si affianca il concetto di *diversità funzionale* [Burlando-Gianetto-Mainini 92]. La diversificazione funzionale consiste nello sviluppare le versioni in modo che producano lo stesso risultato, considerando informazioni ed algoritmi diversi per ogni versione.
La principale differenza introdotta dalla diversificazione funzionale è il punto di partenza delle varie versioni. Mentre nel caso precedente le versioni elaborano gli stessi input, in questo caso ogni versione elabora i propri input.
Il principale problema della diversificazione funzionale, che ne limita l'applicabilità, è rappresentato dal fatto che non tutti i problemi presentano più punti di vista differenti.

Vediamo di chiarire la programmazione a N-Versioni e la diversificazione funzionale con il seguente esempio: si consideri una galleria molto trafficata e si voglia costruire un sistema che segnali su di un tabellone esterno che nella galleria in quel momento il traffico è piuttosto elevato, sconsigliandone l'uso per limitare il rischio di incidenti. Si può costruire il sistema sfruttando dei rilevatori posti all'ingresso ed all'uscita del tunnel per contare il numero di macchine all'interno del tunnel e si possono, quindi, sviluppare due[10] moduli software, con lo scopo di segnalare l'eccessivo traffico, che prendono come input, esclusivamente, i segnali dei rilevatori. Questi moduli devono implementare algoritmi differenti come ad esempio:

[10] Per semplicità si sono considerate solo due versioni e non tre trascurando i problemi che avrà il voter a mettere d'accordo le due versioni (non si può ad esempio determinare una maggioranza).

① Calcolare il numero di macchine presenti nel tunnel.

② Se il numero di macchine nel tunnel è maggiore di un certo numero, segnalare il pericolo.

① Calcolare il numero di macchine presenti nel tunnel.

② Calcolare la distanza media tra le macchine.

③ Se la distanza media è minore di una distanza di sicurezza prefissata, segnalare il pericolo.

È chiaro che lo stesso input porta anche in algoritmi diversi ad effettuare passi simili, nell'esempio il calcolo del numero di macchine nel tunnel, quindi ad aumentare la probabilità di fallimento delle due versioni sullo stesso input.
Se si ritiene che il grado di affidabilità della precedente soluzione non sia soddisfacente si può ricorrere alla diversificazione funzionale.
Si consideri una seconda fonte di informazione oltre ai rilevatori di movimento posti all'ingresso ed all'uscita della galleria, come ad esempio dei rilevatori del grado di produzione di monossido di carbonio all'interno del tunnel. Si possono, quindi, sviluppare le due versioni in modo che una elabori gli input ricevuti dai rilevatori di movimento e l'altra le informazioni ricevute dal rilevatore del tasso di inquinamento.
A questo punto le due versioni sfrutteranno i seguenti algoritmi:

① Calcolare il numero di macchine presenti nel tunnel.

② Se il numero di macchine nel tunnel è maggiore di un certo numero, segnalare il pericolo.

① Se il tasso di monossido di carbonio presente nella galleria supera una certa percentuale per la quale si è calcolata la presenza di un numero eccessivo di macchine nel tunnel (in modo statistico, empirico, etc ...), segnalare il pericolo.

Risulta evidente che con la diversificazione degli input le fasi simili nei due algoritmi si sono ridotte, riducendo, quindi, le probabilità di errori di progettazione comuni.

4.5 Motivazioni all'uso delle Riflessioni nella Costruzione di applicazioni Fault Tolerant

Nelle sezioni precedenti è stata presentata la tolleranza ai guasti e diverse strategie per ottenerla. Presentando le diverse strategie, si è, volutamente, evitato ogni accenno agli approcci utilizzati per introdurre queste strategie nelle applicazioni.
La tolleranza ai guasti, solitamente, viene introdotta in un'applicazione seguendo uno dei seguenti tre approcci:

- Gli aspetti fault tolerant sono forniti dal sistema sottostante.

- Gli aspetti fault tolerant sono forniti da funzioni predefinite facenti parte di apposite librerie.

- Gli aspetti fault tolerant sono introdotti nell'applicazione, sfruttando le peculiarità proprie del linguaggio di programmazione adottato.

Di seguito vengono esaminati i tre approcci, sottolineandone pregi e limitazioni.

4.5.1 Fault Tolerance dovuta al sistema sottostante

In questo approccio, il sottostante run-time system offre un insieme di protocolli per la gestione degli errori, ad esempio basati sulla replicazione come in Delta-4 [Powell 91].
Il vantaggio di questo approccio consiste nel fornire la gestione degli errori in modo trasparente al programmatore dell'applicazione. Il suo principale svantaggio è dato dalla necessità di uno specifico supporto da parte del run-time system sottostante.

4.5.2 Fault Tolerance dovuta a Librerie Predefinite

Questo approccio è basato sull'utilizzo di librerie di funzioni e primitive predefinite. Un buon esempio di questo approccio è rappresentato da ISIS [ISIS 90].
La principale differenza con il precedente approccio è che la gestione degli errori e l'implementazione dell'applicazione sono fatti entrambi allo stesso livello, utilizzando specifici costrutti. Questo significa che chiamate a determinate funzioni devono essere introdotte nel codice dell'applicazione in punti appropriati per ottenere la gestione degli errori desiderata. In altre parole, questo approccio fornisce una gestione degli errori visibile nel codice dell'applicazione.
Il vantaggio di un simile approccio è rappresentato dalla duttilità del sistema, cioè il programmatore può definire, ottimizzandolo per le proprie esigenze, il meccanismo di gestione degli errori. Il principale svantaggio è rappresentato dal fatto che l'aspetto funzionale e quello non funzionale del codice sono mescolati. Questa promisquità dell'aspetto funzionale con quello non funzionale ostacola, tra l'altro, la riusabilità del codice. Questo approccio, ovviamente, non è trasparente al programmatore dell'applicazione.

4.5.3 Fault Tolerance dovuta al Linguaggio di Programmazione

I precedenti due approcci non si affidano ad alcuna particolare proprietà del linguaggio di programmazione adottato, bensì si basano su meccanismi forniti dal sottostante sistema operativo o da uno specifico ambiente.
L'approccio descritto in questa sezione sfrutta le proprietà proprie del linguaggio di programmazione adottato, in particolare le peculiarità (come l'ereditarietà) dei linguaggi di programmazione object-oriented.

L'approccio, basato sull'ereditarietà, consiste nel progettare le tecniche per la fault tolerance in classi di sistema predefinite che vengono usate per l'implementazione di una certa soluzione. L'idea è quella di usare la nozione di ereditarietà per derivare un'implementazione fault tolerant degli oggetti critici. Questa soluzione consiste nel rendere ereditabili le caratteristiche prettamente non funzionali da appropriate classi di sistema. Questo tipo di soluzione è stata utilizzata con successo in Arjuna [Arjuna 92].
Si può sfruttare l'ereditarietà per implementare diverse tecniche di gestione degli errori. Alcune strategie possono richiedere la creazione di diverse repliche e soprattutto l'esecuzione di metodi replicati (cioè eseguiti per ogni replica oltre che per l'oggetto principale). Queste operazioni non possono essere ottenute trasparentemente usando l'ereditarietà.
Comunque la principale difficoltà, che si incontra utilizzando questo approccio, è che l'ereditarietà non concede accessi alla struttura interna degli oggetti e neppure la ridefinizione dei meccanismi fondamentali di un linguaggio object-oriented, come ad esempio il meccanismo di attivazione dei metodi.

In ognuno dei tre approcci presentati il ruolo del programmatore è differente, in accordo con il grado di trasparenza e di separazione tra il livello funzionale e quello non funzionale fornito dall'approccio.
Nel primo caso, i meccanismi di gestione degli errori sono forniti dal sottostante sistema ottenendo, quindi, sia la trasparenza che la separazione del livello funzionale da quello non funzionale. Purtroppo questo approccio manca di flessibilità.
Nel secondo caso, l'ambiente fornisce funzioni di libreria che mettono in grado il programmatore di definire il proprio meccanismo di gestione degli errori. Però in questo modo si mischiano il livello funzionale con quello non funzionale ed ovviamente il processo non è trasparente al programmatore.
Con l'ultimo approccio, l'ereditarietà può essere usata per aggiungere proprietà di fault tolerance ad applicazioni object-oriented. Si ha la separazione del livello funzionale da quello non funzionale, però non si ha una completa trasparenza a causa di convenzioni di programmazione che sono richieste.
Come affermato in [Fabre-Nicomette-Pérennou-Stroud-Wu 95], con la riflessione si riesce ad ovviare agli inconvenienti di questi approcci, ottenendo trasparenza e la separazione del livello funzionale da quello non funzionale, oltre ad una flessibilità intriseca, dovuta alle proprietà delle riflessioni.
La trasparenza è una delle proprietà base delle riflessioni, la struttura a torre delle applicazioni riflessive può essere utilizzata per separare le componenti funzionali, che operano nel livello base, da quelle non funzionali, che operano nei meta-livelli. Si possono implementare nuove strategie di gestione degli errori, definendo nuove meta-entità a partire da quelle esistenti. Si ricorda che le meta-entità sono definite da classi e come tali sono raffinabili. A queste caratteristiche si aggiunge la possibilità di riusare le meta-entità e quindi le varie componenti delle strategie di fault tolerance, ad

esempio il voter, come affermato in [Ancona-Clematis-Dodero-Gianuzzi-Lisboa 95b].

4.6 Conclusioni

In questo capitolo si sono esaminate le tecniche di software fault tolerance, fornendone una breve panoramica. Inoltre si sono descritte le principali tecniche per dotare un'applicazione di un comportamento fault tolerant. I tre principali approcci (uso delle caratteristiche del sottostante sistema, uso di librerie predefinite ed uso di caratteristiche proprie del linguaggio di programmazione) si sono rivelati o poco trasparenti o poco flessibili per poter garantire un'agevole introduzione della gestione degli errori in un'applicazione.

La riflessione, grazie alle sue proprietà intrinseche di trasparenza ed estendibilità, risulta essere una soluzione adeguata per dotare un'applicazione di comportamento fault tolerant, separandolo da quello proprio dell'applicazione in modo trasparente.

L'idea di utilizzare un meta-livello per nascondere l'implementazione di aspetti non funzionali non è nuova. Esistono diverse proposte per l'aggiunta di aspetti non funzionali, come la persistenza e la distribuzione, tramite l'uso delle riflessioni [Attardi-Bonini-Boscotrecase-Flagella-Gaspari 89, Agha-Frølund-Panwar-Sturman 93], mentre Stroud in [Stroud 93] ha affermato che la riflessione è un metodo appropriato per introdurre trasparentemente, in un'applicazione, aspetti non funzionali alla stessa.

Nel seguito si presentano le implementazioni di alcune delle tecniche di fault tolerance presentate, sfruttando la riflessione secondo il classico modello a meta-oggetto ed il modello a materializzazione del canale, che è stato presentato in questa tesi. Si presenta un confronto tra le soluzioni, rimarcando i vantaggi ottenuti dall'uso delle riflessioni rispetto agli approcci tradizionali.

5 Riflessioni Applicate alla Fault Tolerance

5.1 Introduzione

In questo capitolo, si analizza la possibilità di utilizzare l'approccio riflessivo per realizzare alcune delle politiche di tolleranza ai guasti presentate nel capitolo [4].
Si sottolinea che lo scopo di questo capitolo non è quello di individuare la migliore politica di fault tolerance tra quelle presentate o di caratterizzarne pregi e difetti, ma semplicemente di verificare l'applicabilità dei modelli di riflessione presentati, alla costruzione di software tollerante ai guasti.
Si evidenziano, inoltre, i vantaggi che si ottengono dall'utilizzo dell'approccio riflessivo: trasparenza, flessibilità e separazione degli aspetti funzionali e non funzionali.

5.2 Tipologia dei Sistemi Software ed Errori con cui Devono Convivere

Si considera un sistema software composto da oggetti attivi distribuiti su più macchine (separate e senza memoria condivisa) che cooperano tra loro per espletare l'aspetto funzionale del sistema.
Gli oggetti attivi cooperano secondo il modello client|server, cioè un oggetto attivo (client) richiede un servizio ad un altro oggetto attivo (server), il quale per elaborare la richiesta che gli viene inoltrata può richiedere servizi ad altri oggetti attivi (diventando quindi client di altri server).
Si è interessati a gestire errori dovuti all'hardware, in particolare si suppone che le macchine possano fallire solo in modo fail-silent. Quindi gli errori che coinvolgono gli oggetti attivi sono errori di omissione (vedere la sezione [4.2]) e si manifestano con il mancato invio di dati in seguito a sollecitazioni (rilevabili, ad esempio, tramite time-out). Si suppone che l'omissione sia dovuta ad un fallimento della macchina e non ad un fallimento dei mezzi di comunicazione che collegano le diverse macchine.
Si suppone che gli host non falliti siano sempre raggiungibili da un qualunque altro

host non fallito. Inoltre macchine diverse hanno probabilità indipendente di fallire, cioè il fallimento di un host non provoca il fallimento di altri host.

Nei sistemi software che si considerano, si suppone che per un oggetto attivo critico sia fondamentale che ogni servizio che richiede ad altri oggetti attivi venga soddisfatto. Si suppone, inoltre, che per l'oggetto critico sia fondamentale che elabori tutte le richieste di servizi che gli vengono inoltrate. Si effettuano le due ipotesi precedenti perché ogni interazione tra oggetti determina dei risultati che possono modificare sia lo stato dell'oggetto client che lo stato dell'oggetto server. Quindi sotto questa ipotesi per un oggetto critico è fondamentale per una elaborazione corretta che ogni interazione con gli altri oggetti sia corretta.
Quindi per garantire il corretto funzionamento di un oggetto non è sempre sufficiente considerare come critico solo quest'ultimo (dotandolo di comportamento tollerante ai guasti) in quanto la correttezza del suo operato è dipendente dalla correttezza dell'operato degli oggetti con cui interagisce. Un fallimento di questi ultimi impedisce di garantire l'operato dell'oggetto critico.
Chiariamo su un esempio, si supponga di avere gli oggetti $\mathcal{A}, \mathcal{B}$ e $\mathcal{C}$ e che si voglia che il comportamento di $\mathcal{B}$ sia fault tolerant mentre quello di $\mathcal{A}$ e $\mathcal{C}$ no. Si consideri una computazione del tipo

$$\mathcal{A}.\mathrm{a} \Longrightarrow \mathcal{B}.\mathrm{b} \Longrightarrow \mathcal{C}.\mathrm{c}$$

in cui il metodo a di $\mathcal{A}$ attiva il metodo b di $\mathcal{B}$ che attiva a sua volta il metodo c di $\mathcal{C}$. Si consideri un fallimento che coinvolge l'oggetto $\mathcal{C}$. Tale fallimento non è recuperabile in quanto $\mathcal{C}$ non è fault tolerant, quindi $\mathcal{B}$ non può completare la propria azione e non si può, pertanto, garantirne il corretto operato.
Quindi nei sistemi software che si considerano gli oggetti attivi richiedono servizi solo ad oggetti attivi che hanno un grado di sicurezza maggiore od al più uguale al proprio, dove il grado di sicurezza (o *k-resiliency*) di un oggetto attivo rappresenta il numero di fallimenti a cui tale oggetto può sopravvivere ed è legato al suo grado di criticità, più l'oggetto è critico più la sua computazione deve essere sicura. Da questo segue che un oggetto critico, per essere sicuro di condurre a termine le proprie elaborazioni, non deve richiedere servizi ad un oggetto considerato non critico.
Inoltre nei sistemi software che si considerano lo stesso servizio richiesto più volte allo stesso oggetto, ma con stati differenti, fornisce risultati differenti. Questo fa capire che in caso di recupero da un fallimento l'oggetto recuperato non deve richiedere nuovamente servizi già ottenuti pena il dare origine ad una computazione diversa da quella originale.

5.3 Tolleranza ai Guasti Ottenuta per Replicazione

Data la natura degli errori considerati, le politiche più adatte a gestire la loro tolleranza sono le tecniche di system fault tolerance (sezione [4.4.1]) ed in particolare le politiche basate sulla replicazione.

Prima di addentrarsi nella descrizione dei diversi approcci per integrare la replicazione con la riflessione si esaminano, nel caso generale, quali sono le azioni che si devono effettuare per sfruttare le tecniche di fault tolerance basate sulla replicazione:

① La prima azione consiste nella creazione delle repliche secondarie. Le repliche solitamente sono create quando viene creato l'oggetto critico.

② Le successive azioni consistono nel far evolvere lo stato delle repliche consistentemente con lo stato della copia primaria. L'algoritmo di aggiornamento dello stato ed il momento in cui effettuare tale aggiornamento dipendono dal tipo di replicazione adottata e dal livello di criticità del sistema software.

③ In concomittanza alle azioni descritte ai punti ① e ②, si hanno azioni di rilevamento di eventuali condizioni di errore ed azioni necessarie per il conseguente ripristino dello stato corretto della computazione.

Parlando di fault tolerance ottenuta per replicazione il grado di sicurezza (o k-resiliency) di un oggetto coincide con il numero di repliche secondarie che vengono tenute dello stesso. Ad esempio se l'oggetto non è critico (o 0-resiliency) allora non ha repliche, se ha grado di sicurezza pari a 1 (o 1-resiliency) ha una sola replica e così via.

5.4 Replicazione e Meta-Oggetti (Modellazione)

Adottando il modello a meta-oggetto, l'applicazione che si intende rendere tollerante ai guasti, secondo lo schema a replicazione, può essere strutturata in livelli (torre riflessiva) dove le entità in ogni livello cooperano per espletare un certo aspetto funzionale o meno all'applicazione (separazione degli aspetti).
Comunque si intenda strutturare l'applicazione, i meta-oggetti hanno il compito di gestire l'aspetto fault tolerant dell'applicazione, mentre gli oggetti del livello base svolgono l'aspetto funzionale dell'applicazione. Le differenze possibili nella struttura riguardano il numero dei meta-oggetti, le entità a cui sono associati e le modalità con cui si garantisce la tolleranza ai guasti dei meta-oggetti.
Di seguito si esaminano due possibili modellazioni della replicazione utilizzando i meta-oggetti.

5.4.1 Un Unico Meta-Oggetto per ogni Oggetto Critico e le sue Repliche

L'applicazione viene strutturata in modo che ogni oggetto critico e le sue repliche condividano lo stesso meta-oggetto, in questo modo ogni azione effettua-

ta dall'oggetto critico viene monitorata dal meta-oggetto, che provvede ad aggiornare in modo consistente le repliche. Il meta-oggetto, in caso di fallimento di un'entità del livello base, provvede a ripristinarla.
I meta-oggetti vengono replicati. Un meta-oggetto e le sue repliche monitorizzano mutuamente il proprio stato. In caso di fallimento di un meta-oggetto, una delle repliche provvede a sostituirlo con uno nuovo.

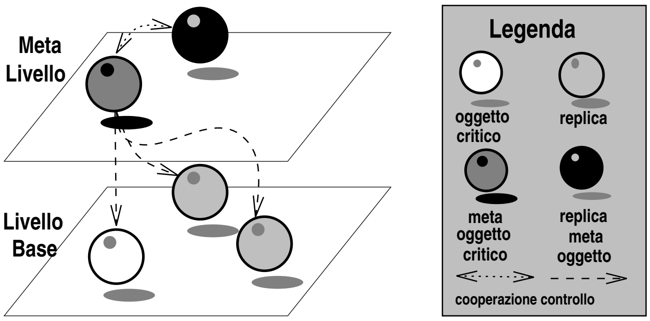


Figura 5.1: Applicazione Riflessiva Fault Tolerant in cui un Meta-Oggetto è Associato ad un Oggetto Critico ed alle sue Repliche

Tale strutturazione (schematizzata in figura [5.1]) presenta il vantaggio di usare un numero limitato di meta-oggetti per gestire gli oggetti critici e le loro repliche. Se un'applicazione coinvolge più oggetti critici distinti, i rispettivi meta-oggetti, cooperano con la propria replica e con nessun altro meta-oggetto, riducendo al minimo le comunicazioni tra meta-oggetti.
Una simile struttura presenta lo svantaggio che per poter essere implementata si deve utilizzare un linguaggio riflessivo che permetta di associare lo stesso meta-oggetto a più oggetti riflessivi (l'oggetto critico e le sue repliche) e che permetta di cambiare il meta-oggetto associato ad un oggetto riflessivo durante l'esecuzione (per effettuare il recupero del meta-oggetto).

5.4.2 Meta-Oggetti per ogni Oggetto Critico e per ogni Replica

L'applicazione viene strutturata in modo che ogni oggetto critico ed ogni sua replica abbiano un loro proprio meta-oggetto distinto. Ogni azione intrapresa dall'oggetto critico viene monitorata dal suo meta-oggetto, il quale provvede ad informare sull'azione svolta dal suo referente i meta-oggetti delle repliche, i quali effettuano le operazioni necessarie per far evolvere le repliche in modo consistente con l'evoluzione dell'oggetto critico.
Il fallimento di un'entità del livello base viene rilevata dal rispettivo meta-oggetto ed è proprio questa ultima entità che provvede al recupero di tale entità. Il fallimento di un meta-oggetto viene rilevato dagli altri meta-oggetti che provvedono al suo recupero.

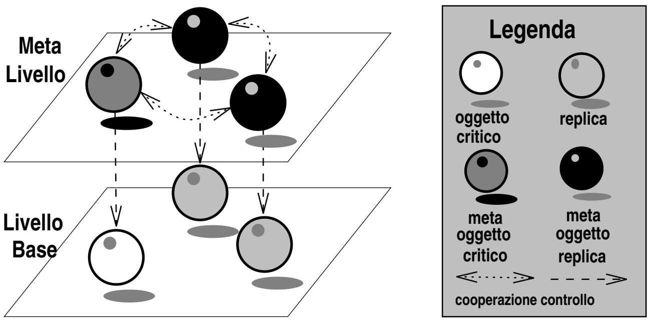


Figura 5.2: Applicazione Riflessiva Fault Tolerant in cui sia l'Oggetto che le sue Repliche Hanno un loro Meta-Oggetto

Tale strutturazione (schematizzata in figura [5.2]) presenta il vantaggio che i meta-

oggetti delle repliche e dell'oggetto critico sono tutti uguali ed evolvono di pari passo, come tale ricoprono anche il ruolo di repliche degli altri meta-oggetti. Inoltre siccome si ha un meta-oggetto sia per l'oggetto critico che per le repliche, allora l'oggetto ed il rispettivo meta-oggetto possono risiedere sullo stesso processore, riducendo il numero dei controlli necessari per verificare il fallimento di oggetto e meta-oggetto. In pratica, procedendo in questo modo, un meta-oggetto controllando se un altro meta-oggetto è fallito verifica anche se è fallito l'oggetto riflessivo a cui è associato. Il recupero può comunque essere effettuato in quanto il meta-oggetto ricopre il ruolo di replica dell'altro meta-oggetto, mentre l'oggetto a cui è associato ricopre il ruolo di replica dell'oggetto a cui è associato il meta-oggetto fallito.
Gli svantaggi sono rappresentati dall'aumento del numero di meta-oggetti e della conseguente necessità di coordinamento.
La modellazione descritta è stata presentata in [Fabre-Nicomette-Pérennou-Wu 94].

Queste sono solo due delle molteplici strutturazioni possibili. Ogni strutturazione presenta vantaggi e svantaggi e la loro applicazione va valutata caso per caso.

5.4.3 Replicazione Passiva e Modello a Meta-Oggetto

Finora si sono descritti, in astratto, alcune possibili strutturazioni per un sistema software reso fault tolerant, usando la replicazione. Adesso si approfondisce tale approccio scegliendo la replicazione passiva (la replicazione attiva e quella semi-attiva sono ottenute in maniera analoga a partire dagli schemi generali, modificando solo gli algoritmi usati per pilotare l'evoluzione delle repliche, per maggiori dettagli vedere [Fabre-Nicomette-Pérennou-Wu 94]). L'algoritmo che si segue è quello presentato da Powell in [Powell 91] e riportato nella sezione [4.4.1.1].

La tecnica di replicazione passiva consiste nel mantenere una replica completa dell'oggetto critico, aggiornandone lo stato consistentemente con quello dell'oggetto critico ad ogni azione di checkpoint. In caso di fallimento di un oggetto critico la replica consente il ripristino della computazione. Occorre quindi stabilire tre cose:

- Quando si effettuano le azioni di checkpoint.
- Cosa fanno i meta-oggetti ad ogni azione di checkpoint
- Come e quando vengono effettuate le operazioni di rilevamento e ripristino da condizioni di errore.

Checkpoint: il momento in cui si deve aggiornare lo stato della replica è dipendente dalla criticità dell'applicazione, più questa è critica più le azioni di checkpoint devono essere frequenti (ovviamente in corrispondenza di cambiamenti di stato).

In questa trattazione si sceglie di effettuare un'operazione di checkpoint ogniqualvolta l'oggetto critico termina l'elaborazione di un servizio (metodo) che modifica il proprio stato.

Meta-Oggetto e Checkpoint: i meta-oggetti hanno due compiti: il primo è quello di rilevare e quindi recuperare da una eventuale condizione di errore, il secondo è quello di mantenere consistente lo stato delle repliche con quello dell'oggetto di cui sono repliche.
Attivata un'azione di checkpoint, il meta-oggetto dell'oggetto critico comunica il nuovo stato del suo referente, che conosce grazie alla riflessione strutturale, al meta-oggetto della replica dell'oggetto critico, il quale, sempre tramite la riflessione strutturale, si occupa di aggiornare lo stato della replica.

Rilevamento e Ripristino da una Condizione di Errore: con operazione di rilevamento di un errore si intende quell'insieme di azioni atte a determinare la presenza o meno di una condizione di errore.
Sfruttando la torre riflessiva presentata, il rilevamento viene effettuato dai meta-oggetti nel seguente modo: il meta-oggetto di un oggetto critico, ad ogni computazione riflessiva, verifica lo stato del meta-oggetto della replica dell'oggetto critico. Siccome gli errori che si intende gestire sono errori di omissione, il rilevamento dei fallimenti può consistere nello scambio di messaggi ed attesa del relativo acknowledge.
In caso si rilevi un errore, cioè nel caso in cui un oggetto critico non fornisca alcuna risposta entro un periodo di tempo t prestabilito a partire dal momento in cui è stata sottoposta la richiesta, occorre recuperare la situazione d'errore. Il recupero viene effettuato nel seguente modo: il meta-oggetto che rileva l'errore, crea una nuova replica con relativo meta-oggetto e li inizializza entrambi, rispettivamente, con lo stato dell'oggetto che monitorizza, informazioni che può ottenere grazie alla riflessione strutturale, ed il proprio stato.
Un comportamento simmetrico viene tenuto dal meta-oggetto della replica dell'oggetto critico. In questo modo è possibile recuperare dal crash sia della replica che dell'oggetto replicato e dei relativi meta-oggetti.

5.4.4 Codice OpenC++ per Implementare la Replicazione Passiva

Adesso, tenendo presente le osservazioni esposte in precedenza, si presenta una possibile implementazione in OpenC++ dei meta-oggetti che gestiscono la replicazione passiva.

La natura di preprocessore del linguaggio OpenC++, come visto nella sezione [3.5], lega inscindibilmente un oggetto riflessivo ad un ben preciso meta-oggetto, perciò non è possibile cambiare il meta-oggetto ad un certo oggetto durante l'esecuzione.

Inoltre la relazione tra oggetto e meta-oggetto è biunivoca, quindi non è possibile associare due meta-oggetti diversi allo stesso oggetto né due oggetti diversi allo stesso meta-oggetto.
Da queste affermazioni segue che delle strutturazioni presentate nella sezione [5.4] l'unica implementabile in OpenC++ è quella che adotta un meta-oggetto per ogni replica distinto da quello dell'oggetto critico. Si può comunque osservare che questa limitazione è dovuta al linguaggio OpenC++, in quanto, ad esempio, con il meccanismo sviluppato nella prima parte di questo lavoro entrambe le strutturazioni descritte sono implementabili. Infatti il metodo set_meta permette sia di cambiare il meta-oggetto di un oggetto durante l'esecuzione sia di associare lo stesso meta-oggetto a due oggetti distinti.

Definizione della Classe degli Oggetti Critici

Innanzitutto si analizzano le modifiche che si devono apportare al codice delle classi le cui istanze sono critiche per renderle tolleranti ai guasti. A tale proposito si considera una generica classe X, che definisce un unico attributo x_state ed un metodo set_x_state che modifica tale attributo ed un altro metodo read_x_state che non modifica il valore dell'attributo x_state e si suppone che le istanze della classe X siano oggetti critici.
Per garantire la tolleranza ai guasti delle istanze della classe X, occorre dotarle di meta-oggetti istanza di una particolare classe pr_metaobj, che si descrive di seguito, ed occorre inoltre effettuare delle meta-computazioni. In particolare è necessaria una meta-computazione nei seguenti casi:

- Subito dopo che l'oggetto critico è stato creato, il meta-oggetto deve creare la replica ed il relativo meta-oggetto.
- Ogniqualvolta l'oggetto critico esegue un metodo che ne modifica lo stato, il suo meta-oggetto deve verificare che la replica non sia fallita ed effettuare le azioni di checkpoint.
- Quando si distrugge l'oggetto riflessivo critico si deve provvedere a distruggere anche le sue repliche.

Per eseguire meta-computazioni dopo l'attivazione del costruttore e prima di quella del distruttore OpenC++ dota i suoi meta-oggetti di due metodi (Meta_Startup e Meta_Cleanup) attivati proprio in tali occasioni. Si sottolinea che la meta-computazione elaborata dal metodo Meta_Startup non è un'azione riflessiva sulla creazione di un oggetto, in quanto la prima è effettuata dal meta-oggetto dell'oggetto riflessivo, mentre la seconda dovrebbe essere effettuata dalla classe del meta-oggetto come visto nella sezione [2.4]. In OpenC++ quest'ultima è ottenuta definendo i costruttori come metodi riflessivi ed è realizzata tramite i metodi Meta_Create e Meta_HandleCreate. In OpenC++ una meta-computazione sull'attivazione di un metodo è gestita dal metodo

del meta-oggetto Meta_MethodCall ed è possibile discernere quali metodi richiedono una computazione riflessiva o meno, definendoli come metodi riflessivi grazie alla direttiva //MOP reflect.
Nella classe X utilizzata per la spiegazione solo il metodo **set_x_state** modifica lo stato delle istanze. Quindi viene definito come riflessivo facendolo precedere dalla direttiva //MOP reflect. Il metodo **read_x_state**, invece, non ha side-effect sullo stato dell'oggetto e quindi non è un metodo riflessivo.

Di seguito è riportato il codice OpenC++ relativo alla classe X.

```
class X {
  public:
    X( void );
    X( int init_value );
    ~X( void ) {};
    int read_x_state( void ) { return x_state }; // metodo non riflessivo
   //MOP reflect:
    void set_x_state( int new_value );            // metodo riflessivo
  protected:
   //MOP reflect:
    int x_state;
};

//MOP reflect class X: pr_metaobj;

X::X( void ) : x_state( 0 ) {};

X::X( int init_value ) : x_state( init_value ) { };

void X::set_x_state( int new_value ) {
  x_state += new_value;
};
```

Osservando la definizione della classe X si nota subito che questa classe definisce esclusivamente il codice necessario per espletare l'aspetto funzionale dell'applicazione. L'aspetto non funzionale viene aggiunto in modo trasparente dalla direttiva

//MOP reflect class X: pr_metaobj

sono, quindi, le istanze della classe pr_metaobj associate alle istanze della classe X che si occupano di gestirne la tolleranza ai guasti.

Definizione della Classe dei Meta-Oggetti che Gestiscono la Replicazione Passiva

La classe pr_metaobj definisce i meccanismi per ottenere la tolleranza ai guasti, e questi meccanismi sono indipendenti dal tipo di oggetti riflessivi a cui vengono applicati. Questo significa che la classe pr_metaobj può essere utilizzata non solo per dotare le istanze della classe X di comportamento fault tolerant secondo lo schema a replicazione passiva, ma anche le istanze di una qualsiasi altra classe.

Di seguito si riporta la definizione in OpenC++ della classe pr_metaobj

```
class pr_metaobj : public OcCoreMetaObj {
  private:
    const int MetaObj'sCritical = 0, MetaObj'sReplica = 1, dummy_method_id= -1;
    int replicas_number, mo_status   //assume i valori MetaObj'sCritical o MetaObj'sReplica
    char replica_exe[25];
  public:
   // costruttore
    pr_metaobj( int init_value, int OcId, char *replica_exe, rn = 1 );
   // metodi per la gestione della fault tolerance
    void create_replicas( void );
    int fault_detection( void );
    void send_checkpoint( void );
    void fault_recovery( void );
    void make_checkpoint( ArgPac &Checkpoint );
    void update_status( ArgPac &Checkpoint );
   // ridefinizione di metodi ereditati da OcCoreMetaObj
    void Meta_Startup( void );
    void Meta_EventCallbackBody( Id method_id, ArgPac &args, ArgPac &reply );
    void Meta_MethodCall( Id method_id, Id category, ArgPac &args, ArgPac &reply );
    void Meta_Cleanup( void );
}
```

Nella definizione della classe pr_metaobj si sfrutta la libreria di classi offerta da OpenC++ ed in particolare la classe OcCoreMetaObj per concedere agli oggetti critici, monitorati da meta-oggetti istanza della classe pr_metaobj, di evolvere concorrentemente alle repliche.
La classe OcCoreMetaObj fornisce le primitive per gestire le comunità di oggetti (vedere la sezione [3.5.2]), questo permette agli oggetti critici ed alle rispettive repliche di far parte della stessa comunità, consentendo così ai loro meta-oggetti di interagire facilmente.

Le istanze della classe pr_metaobj sovraintendono sia gli oggetti critici che le loro repliche e devono garantire la continuità d'operato dell'oggetto critico, ciò è ottenuto cooperando per far evolvere le repliche concorrentemente e consistentemente con l'evoluzione dell'oggetto critico e sostituendo l'oggetto critico con una replica quando rilevano un fallimento.
In caso di fallimento la computazione riprende ripetendo dall'inizio l'esecuzione del servizio che l'oggetto fallito stava fornendo.
Per evitare di fornire più volte lo stesso servizio (sempre che non sia effettivamente richiesto), i meta-oggetti tengono le code dei servizi richiesti e dei risultati forniti, come nell'algoritmo presentato da Powell in [Powell 91], e quando viene nuovamente richiesto un servizio già fornito il meta-oggetto se ne avvede grazie alla coda dei servizi richiesti ed evita di ricalcolarlo. I servizi sono identificati unicamente tramite, ad esempio, un identificatore composto dal "nome del client" e dal valore che ha il "program counter" del client al momento in cui inoltra la richiesta (in caso di fallimento anche il program counter viene ripristinato al valore che aveva quando è stato effettuato l'ultimo checkpoint).

Fornito un servizio, le repliche modificano il proprio stato in accordo con lo stato dell'oggetto critico.

Il costruttore della classe pr_metaobj, oltre a creare il meta-oggetto, si occupa di creare, se non è già stato fatto, la comunità di oggetti e di fare unire a tale comunità l'oggetto (critico o replica) di cui è parte integrante.

```
pr_metaobj::pr_metaobj( int init_value = MetaObj'sCritical, int OcId = nil, char *rexe, int rn ) :
        mo_status( init_value ), replica_exe( rexe ), replica_number( rn ) {
  BOOL init = FALSE;
  ArgPac dummy_args;

  if ( OcId == nil ) Meta_CreateOc();
  else this_oc_id = OcId;   // this_oc_id è definito in OcCoreMetaObj
  Meta_Join( this_oc_id, &init, &dummy_args );
}
```

I metodi definiti (o ridefiniti) nella classe pr_metaobj, in base allo scopo, possono essere distinti in due classi: metodi legati alla riflessione e metodi legati alla fault tolerance.

Metodi per la gestione delle Riflessioni

I metodi legati alla riflessione sono la ridefinizione di metodi ereditati dalla classe OcCoreMetaObj e sono automaticamente attivati in certe occasioni per effettuare il cambio di contesto tra livello base e meta-livello dell'applicazione e quindi delle meta-computazioni.

Meta_Startup: questo metodo viene attivato (automaticamente) successivamente alla creazione dell'oggetto e del suo meta-oggetto ed a seconda dell'entità che lo attiva (meta-oggetto dell'oggetto critico o meta-oggetto della replica dell'oggetto critico), discriminata grazie al contenuto della variabile mo_status. Se è attivato dal meta-oggetto dell'oggetto critico si occupa di creare le repliche (attivando il metodo create_replicas). Invece se è attivato dal meta-oggetto di una replica dell'oggetto critico entra in un ciclo in cui verifica sistematicamente lo stato dell'oggetto primario.

```
void pr_metaobj::Meta_Startup( void ) {
 // Metodo attivato dopo la creazioni di un oggetto e del suo meta-oggetto
  if ( mo_status == MetaObj'sCritical ) // meta-oggetto oggetto critico
    for( int i=0; i<replicas_number; i++ ) create_replicas();
  else {   // mo_status == MetaObj'sReplica
    while( TRUE )
      if ( fault_detection() ) break;
    fault_recovery();
  }
}
```

Sorge spontaneo chiedersi come mai queste azioni non vengono effettuate come una meta-computazione sulla creazione degli oggetti. Sebbene, in OpenC++, sia possibile effettuare una meta-computazione sulla creazione degli oggetti (tramite Meta_Create e Meta_HandleCreate) tale operazione viene svolta da un'entità diversa dal meta-oggetto dell'oggetto riflessivo, in quanto quest'ultimo, giustamente, non è ancora stato creato e quindi non si è in grado di discernere quale entità si sta creando (l'oggetto critico o la sua replica) e conseguentemente quale attività effettuare.

Meta_MethodCall: questo metodo viene attivato (automaticamente) nel momento in cui un oggetto riflessivo tenta di attivare un metodo riflessivo (i metodi sono definiti come riflessivi tramite la direttiva //MOP reflect).
Per sfruttare al meglio lo schema a replicazione passiva implementato dalla classe pr_metaobj si devono definire come riflessivi quei metodi la cui azione modifica lo stato dell'oggetto. Nel momento in cui si tenta di attivare un metodo riflessivo si ha l'intercettazione della computazione da parte del meta-oggetto che attiva il metodo Meta_MethodCall. Meta_MethodCall, verificato che il servizio non sia già stato fornito tramite le code dei servizi forniti, permette all'oggetto referente di eseguire effettivamente il metodo (Meta_HandleMethodCall), quindi controlla lo stato della replica, se questa è fallita ne effettua il ripristino (fault_recovery), comunque costruisce ed invia alla replica il checkpoint dello stato dell'oggetto critico (send_checkpoint).

```
void pr_metaobj::Meta_MethodCall( Id method_id, Id category, ArgPag &args, ArgPac &reply ) {
 // Questo metodo viene attivato solo dal meta-oggetto dell'oggetto critico,
 // quindi non è necessario testare il contenuto di mo_status.
  if ( !deja_requested() ) {
    Meta_HandleMethoCall( method_id, args, reply ); // esegue il metodo richiesto
    if ( fault_detection() ) fault_recovery();
    send_checkpoint(); //costruisce ed invia il checkpoint alle repliche
    set_deja_requested();
  } else recovery_reply( &reply );
}
```

I metodi deja_requested, set_deja_requested e recovery_reply si limitano a visitare ed a modificare i dati relativi alla gestione dei servizi già forniti, quindi non si dettaglia ulteriormente il loro codice.

Meta_Cleanup: questo metodo viene attivato (automaticamente) immediatamente prima del momento in cui l'oggetto riflessivo cessa la propria attività (ad esempio tramite una delete o la chiamata al distruttore). La distruzione di un oggetto critico deve portare alla distruzione di tutte le repliche. Compito della Meta_Cleanup, quando attivata in seguito alla richiesta di distruzione dell'oggetto critico, è proprio quello di richiedere a tutte le repliche, membri della comunità di oggetti di cui fa parte tale replica, l'attivazione del distruttore.

```
void pr_metaobj::Meta_Cleanup( void ) {
 // richiamato prima di terminare
  if ( mo_status == MetaObj'sCritical ) {
    ArgPac dummy_args; char method_name[50];
    int method_id = 0;
    while( TRUE ) {
      strcpy( method_name, Meta_GetMethodName( method_id++ ) );
```

```
      if ( method_name[0] == '~' ) break;
    }
    Meta_WaitForEvent( Meta_EventNotify( --method_id, dummy_args ), dummy_args );
    Meta_Disjoin( dummy_args );
    OcMetaObj::KillOcManager();
  }
  else Meta_Disjoin( dummy_args );
}
```

I metodi e le variabili, di una classe le cui istanze sono riflessive in OpenC++ sono disponibili al meta-oggetto sotto forma di indici e tramite quest'ultimi è possibile risalire risalire al nome tramite i metodi Meta_GetMethodName e Meta_GetVarName. Utilizzando tali metodi, cercando un metodo il cui nome inizi per '∼', si risale al codice del distruttore di cui poi viene richiesta la computazione alle repliche. Procedendo in questo modo risulta necessario definire per ogni classe, le cui istanze devono essere rese riflessive, un distruttore.
Questa routine è indipendente dalla classe dell'oggetto base e dal numero delle repliche, in quanto viene ricercato il distruttore in base alla presenza del simbolo '∼' nel nome e si notifica la richiesta della sua attivazione a tutte le repliche, che sono tutte membri della stessa comunità d'oggetti, di cui per altro non se ne deve conoscere il numero.

Meta_EventCallbackBody: mentre gli altri ridefiniscono metodi della classe MetaObj, da cui eredita la classe OcCoreMetaObj, e che quindi non hanno niente a che fare con la gestione delle comunità di oggetti, il metodo Meta_EventCallbackBody viene definito nella classe OcCoreMetaObj ed è utilizzato per gestire la ricezione di una richiesta da parte di un altro oggetto attivo.
Il metodo Meta_EventCallbackBody viene attivato (automaticamente) quando viene notificato, da un membro della comunità di oggetti, agli altri membri un messaggio (tramite il metodo Meta_EventNotify) ed è ridefinito in modo che effettui l'aggiornamento dello stato di una replica (tramite update_status) quando è attivato in seguito alla notifica di un checkpoint (riconoscibile tramite l'identificatore dummy_method_id). Se invece è attivato in seguito ad una richiesta di servizio attiva il metodo Meta_MethodCall che gestisce la computazione effettiva del servizio.

```
void pr_metaobj::Meta_EventCallbackBody( Id method_id, ArgPac &args, ArgPac &reply ) {
 // viene attivata (automaticamente) in seguito alla ricezione di
 // un messaggio notificato tramite Meta_EventNotify
  if ( method_id != dummy_method_id ) Meta_MethodCall( method_id, args, reply );
  else update_status( args ); // richiesta di checkpoint
}
```

Metodi per la Gestione della Tolleranza ai Guasti

I metodi legati alla fault tolerance sono metodi propri della classe pr_metaobj, cioè vengono definiti in questa e non ereditati da altre classi ed ivi ridefiniti, ed effettuano azioni prettamente inerenti la tecnica di fault tolerance implementata dalla classe, come la creazione delle repliche o le azioni di checkpoint. I metodi per la gestione della replicazione possono essere suddivisi in due classi: rilevamento e recupero del fallimento e gestione della consistenza dello stato delle repliche. Per primi si descrivono i metodi necessari alla gestione dello stato delle repliche.

send_checkpoint: questo metodo costruisce il checkpoint[1] (tramite il metodo make_checkpoint), ed invia a tutte le repliche tale checkpoint (usando il metodo Meta_EventNotify). Ed attende la serializzazione della richiesta per non essere interrotto successivamente.

```
void pr_metaobj::send_checkpoint( void ) {
  ArgPac Checkpoint, dummy_args;
  make_checkpoint( Checkpoint );    // costruzione tramite riflessione strutturale del checkpoint
    // invia a tutti i membri della comunità il checkpoint
  Meta_WaitForEvent( Meta_EventNotify( dummy_method_id, CheckPoint ), dummy_args );
}
```

Il fatto che oggetto critico e repliche appartengono ad un'unica comunità di oggetti consente di inviare il checkpoint a tutte le repliche senza conoscerne il numero.

make_checkpoint: questo metodo costruisce un checkpoint istanza della classe ArgPac, che è una struttura dati simile agli stack usata in OpenC++ per passare i parametri dei metodi riflessivi ai metodi del meta-oggetto. Il checkpoint viene costruito sfruttando la riflessione strutturale (il metodo Meta_ReadValue recupera il contenuto della variabile specificata dall'identificatore passatogli) e contiene per ogni identificatore il corrispondente valore.

```
void pr_metaobj::make_checkpoint( ArgPac &Checkpoint ) {
  Checkpoint = new ArgPac;
  Checkpoint.BeginSettingArgs(); // inizializzazione del checkpoint
  Id i = 0; ArgPac value; void *valp;
  while( strcmp(Meta_GetVarName( i ), ``\ 0'') != 0 ) {
    Meta_ReadValue( i, value ); // legge il valore della variabile identificata da i e lo mette in value
    valp = value.PopPtr(); // estrae il contenuto della variabile riferita da i
    Checkpoint.PushBin( *valp, sizeof(*valp) ); // memorizza valp nel checkpoint
    Checkpoint.PushInt( i ); // memorizza i nel checkpoint
  }
   // inserimento nel checkpoint di altre informazioni utili per il ripristino della computazione
}
```

[1] Per semplicità nel codice presentato non si fa distinzione sul tipo di modifica in seguito alla quale si effettua un'operazione di checkpoint. Si utilizza quindi sempre lo stesso metodo anche se la modifica riguarda soltanto una variabile.

Altre informazioni inerenti l'oggetto ed il meta-oggetto, come ad esempio il program counter, devono essere inserite nel checkpoint, però queste informazioni hanno poco o niente a vedere con l'integrazione della fault tolerance con la riflessione, come tale non vengono ulteriormente dettagliate.

update_status: questa routine viene richiamata dal metodo Meta_EventCallbackBody, in seguito alla notifica del ricevimento di un checkpoint da parte della copia primaria. Come nel caso del metodo make_checkpoint, l'aggiornamento dello stato della replica consistentemente con lo stato della copia primaria è fatto sfruttando la riflessione strutturale (tramite il metodo Meta_HandleAssign che assegna il valore contenuto in args alla variabile specificata da i).

```
void pr_metaobj::update_status( ArgPac &checkpoint ) {
  Id var_id; void *valp;
    // aggiornamento delle informazioni esterne allo stato
  do {
    var_id = checkpoint.PopInt();
    valp = (void *)checkpoint.PopBin();
    ArgPac args = new ArgPac;
    args.BeginSettingArgs();
    args.PushPtr( valp );
    Meta_HandleAssign( var_id, args );
    delete args;
  } while( var_id != 0 );
}
```

Ovviamente anche le altre informazioni che si è detto essere contenute nel checkpoint devono essere estratte in modo analogo da quest'ultimo.

Il secondo gruppo di metodi riguarda il rilevamento di un fallimento ed il conseguente recupero.

create_replicas: questo metodo crea una nuova replica e può essere utilizzato sia per creare una replica che sostituisca una replica fallita sia per creare per la prima volta le repliche dell'oggetto critico.
La comunità di oggetti fornite con OpenC++ forniscono la possibilità di scambiare informazioni con altri oggetti|processi residenti anche su macchine diverse utilizzando i socket, però non forniscono la possibilità di creare tali processi su altre macchine oltre a quella di partenza. Questo è un problema facilmente risolvibile utilizzando altre librerie integrabili con OpenC++ come ad esempio PVM; però non essendo essenziale integrare OpenC++ con altre librerie al fine di questa trattazione si suppone semplicemente l'esistenza di un metodo *spawn* che implementi od incapsuli il codice necessario ad effettuare la creazione di un processo su un'altra macchina. Tale processo rappresenta un oggetto replica e deve eseguire un codice simile al seguente, che è pensato appositamente per replicare delle istanze della classe X.

```
void main( int argc, char **argv ) {
```

```
    if ( atoi( argv[1] ) == 0 ) pr_metaobj::InitOcManager( "Replica_X" );
    Refl_X obj( atoi( argv[1] ), atoi( argv[2] ), "main_replica" );
    while( TRUE ) {
      obj.get_request(); // attende una richiesta
      obj.perform_request(); // elabora la richiesta ricevuta
    }
  }
```

Il metodo **create_replicas**, creata la replica, provvede ad inviarle un checkpoint (**send_checkpoint**) inizializzandola così consistentemente con lo stato della copia primaria.

```
void pr_metaobj::create_replicas( void ) {
  char temp_status = ((mo_status == MetaObj'sCritical)?'1':'0');
  char temp_oc_id[10];
  sprintf( temp_oc_id, "%d", this_oc_id );
    // spawn esegue un nuovo processo (la replica) su un'altra macchina
  spawn( "", replica_exe, temp_status, temp_oc_id, replica_exe, NULL );
  send_checkpoint(); // ne inizializza lo stato
}
```

fault_recovery: questo metodo viene richiamato nel caso si rilevi un fallimento della replica o dell'oggetto critico. Quindi se fallisce il processo contenente l'oggetto critico, il metodo **fault_recovery** viene attivato dal meta-oggetto della replica, mentre se fallisce il processo contenente la replica, allora il metodo **fault_recovery** viene attivato dal meta-oggetto dell'oggetto critico. Se è richiamato dal meta-oggetto della replica si promuove la replica secondaria a replica primaria e la si connette al sistema. In entrambi i casi si crea una nuova replica con **create_replicas** con stato consistente con quello dell'oggetto il cui meta-oggetto ha rilevato il fallimento.

```
void pr_metaobj::fault_recovery( void ) {
  int old_status = mo_status;
  mo_status = MetaObj'sCritical; // promozione della replica a copia primaria
  if ( old_status == MetaObj'sReplica )
    connect_newprimary(); // inserimento nel sistema della nuova copia primaria
  create_replicas(); // crea la nuova replica secondaria
}
```

Per quanto riguarda i metodi **fault_detection** e **connect_newprimary** si limitano a fare ciò che intuitivamente suggerisce il loro nome. Il primo metodo verifica se l'altra componente (ad esempio l'oggetto critico controlla la replica e viceversa) è in uno stato di fallimento, mentre il secondo inserisce nel sistema, in caso di fallimento, una nuova copia primaria facendo si che le richieste giungano a quest'ultima e non alla copia fallita. Non si approfondisce ulteriormente il loro codice sia perché dal punto di vista dell'integrazione tra riflessione e politiche di fault tolerance non aggiungono nulla di nuovo sia perché in letteratura è possibile reperirne ottime versioni.

5.4.5 Vantaggi Ottenuti dall'Uso del Modello a Meta-Oggetto

La classe **pr_metaobj** implementa la politica di replicazione passiva senza effettuare ipotesi sulla struttura della classe degli oggetti riflessivi. La generalità della classe pr_metaobj comporta la possibilità di associare delle sue istanze ad oggetti istanza di qualsiasi classe. Inoltre gli oggetti riflessivi sono completamente ignari dell'esistenza dei meccanismi di gestione della fault tolerance. Quindi senza alcuno sforzo ulteriore si può replicare, utilizzando la replicazione passiva, in modo trasparente qualsiasi oggetto critico.

Il meccanismo della riflessione risulta essere anche molto flessibile, infatti a partire da poche classi di meta-oggetti che implementano le principali politiche di gestione della tolleranza ai guasti, è possibile definire (grazie all'ereditarietà) una nuova classe che implementa la politica di gestione desiderata.
Se ad esempio fosse necessario, per ragioni di efficienza, avere una politica di fault tolerance basata sulla replicazione passiva che inserisce nel checkpoint solo le modifiche apportate dall'ultima azione di checkpoint (delta) è sufficiente definire una sottoclasse di pr_metaobj in cui si ridefiniscono soltanto alcuni metodi (make_checkpoint e send_checkpoint) per ottenere il comportamento desiderato.

La decomposizione dell'applicazione in più livelli evidenzia la separazione dell'attività funzionale, riservata alle entità del livello base, da quella non funzionale, riservata alle meta-entità operanti nei meta-livelli, permettendo un agevole riuso delle politiche di fault tolerance.

5.4.6 Comportamento di una Applicazione Fault Tolerant

Per descrivere il comportamento di una applicazione resa fault tolerant utilizzando la riflessione ed in particolare la classe pr_metaobj descritta, si utilizza un esempio.
Per semplicità si considera un'applicazione composta da soli due oggetti attivi, un client ed un server. L'operato dell'oggetto client $\mathcal{A}_{Normale}$ non è critico mentre l'operato dell'oggetto server $\mathcal{A}_{Critico}$ è critico con un grado di criticità pari ad uno. $\mathcal{A}_{Critico}$ è dotato di un meta-oggetto $\mathcal{MA}_{Critico}$ istanza della classe pr_metaobj e di un'unica replica $\mathcal{A}_{Replica}$, a sua volta dotata di un meta-oggetto $\mathcal{MA}_{Replica}$.
Nell'esempio si considera come critico il server, fermo restando che un oggetto critico richiede servizi solo ad oggetti considerati più critici di lui, si può considerare il client od entrambi come critici e la descrizione del comportamento rimane analoga.

Il caso in cui si verifica un fallimento durante la creazione dell'oggetto critico $\mathcal{A}_{Critico}$ è gestito a parte. Poiché il fallimento occorre prima della creazione della replica, l'unico modo in cui può essere rilevato è tramite un'opportuna ridefinizione del metodo Meta_Create che rilevi e recuperi un eventuale errore durante la creazione dell'oggetto riflessivo.

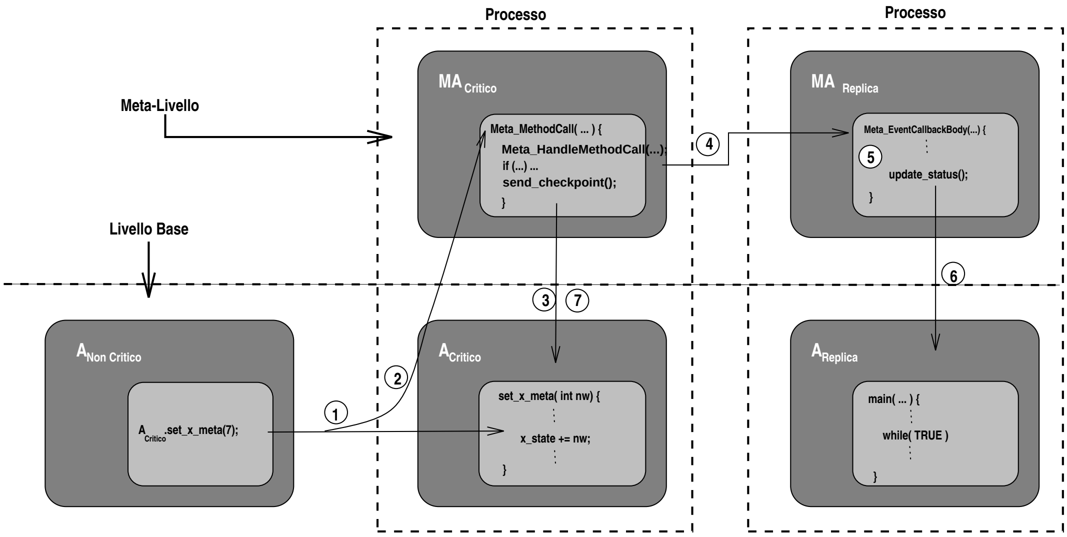


Figura 5.3: **Esempio di Computazione in Assenza di Fallimenti**

5.4.6.1 Attività di $\mathcal{A}_{Critico}$ senza Fallimenti

Si supponga che $\mathcal{A}_{Normale}$ richieda ad $\mathcal{A}_{Critico}$ l'esecuzione del servizio set_x_state sul valore 7. Si supponga, inoltre, che inizialmente lo stato dell'oggetto critico $\mathcal{A}_{Critico}$ e della sua replica siano corretti (ad esempio $\mathcal{A}_{Critico}$.x_state = $\mathcal{A}_{Replica}$.x_state = 0). Si esaminano i passi effettuati per elaborare il servizio richiesto nel caso in cui non si incorra nel fallimento di alcuna entità.

① $\mathcal{A}_{Normale}$ inoltra ad $\mathcal{A}_{Critico}$ la richiesta per il servizio set_x_state ($\mathcal{A}_{Critico}$.x_state = $\mathcal{A}_{Replica}$.x_state = 0).

② Il metodo set_x_state è un metodo riflessivo, pertanto si ha l'inizio di una meta-computazione svolta da $\mathcal{MA}_{Critico}$ (Meta_MethodCall).

③ $\mathcal{MA}_{Critico}$ attiva il metodo Meta_HandleMethodCall che permette ad $\mathcal{A}_{Critico}$ di effettuare il servizio richiesto (set_x_state), modificando quindi il contenuto di x_state ($\mathcal{A}_{Critico}$.x_state = 7, $\mathcal{A}_{Replica}$.x_state = 0).

④ $\mathcal{MA}_{Critico}$ verificato che la replica $\mathcal{A}_{Replica}$ non è fallita, costruisce ed invia il checkpoint alla replica (send_checkpoint).

⑤ Il meta-oggetto della replica $\mathcal{MA}_{Replica}$, ricevuto il checkpoint, attiva il metodo Meta_EventCallbackBody.

⑥ Il metodo Meta_EventCallbackBody gestisce, tramite la riflessione strutturale l'aggiornamento effettivo dello stato della nuova replica $\mathcal{A}_{Replica}$ ($\mathcal{A}_{Critico}$.x_state = $\mathcal{A}_{Replica}$.x_state = 7).

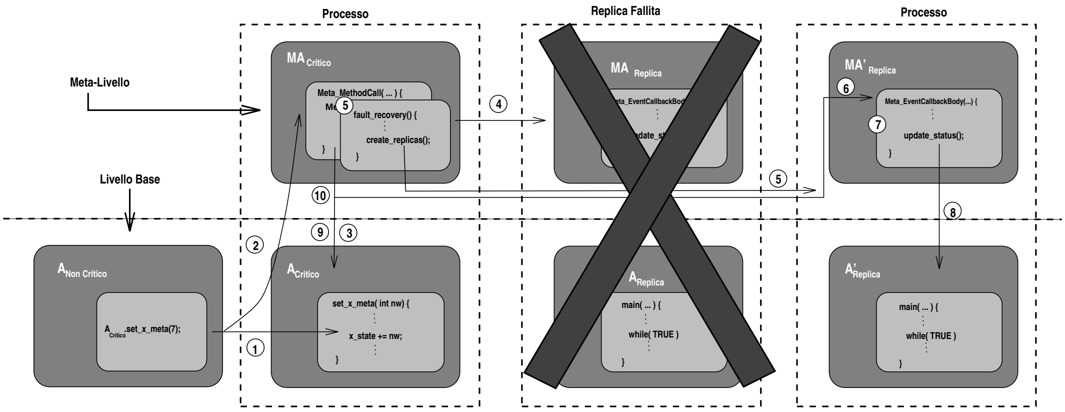


Figura 5.4: **Esempio di Computazione in cui si Verifica il Fallimento dell'Host su cui Risiede la Replica**

⑦ Avvenuto l'invio del checkpoint, $\mathcal{MA}_{Critico}$ termina la sua meta-computazione.

Terminata la computazione, schematizzata in figura [5.3], lo stato di $\mathcal{A}_{Critico}$ è stato modificato e consistentemente con tale modifica anche lo stato di $\mathcal{A}_{Replica}$ è stato modificato.

5.4.6.2 Fallimento della Replica $\mathcal{A}_{Replica}$

Si supponga che l'oggetto non critico $\mathcal{A}_{Normale}$ richieda all'oggetto critico $\mathcal{A}_{Critico}$ l'esecuzione del servizio set_x_state sul valore 7. Si supponga, inoltre, che inizialmente lo stato dell'oggetto critico $\mathcal{A}_{Critico}$ e della sua replica $\mathcal{A}_{Replica}$ siano corretti (ad esempio $\mathcal{A}_{Critico}.\mathsf{x\_state} = \mathcal{A}_{Replica}.\mathsf{x\_state} = 0$). Si esaminano i passi effettuati per elaborare il servizio richiesto in caso di fallimento della replica $\mathcal{A}_{Replica}$ dell'oggetto critico $\mathcal{A}_{Critico}$.

① $\mathcal{A}_{Normale}$ inoltra ad $\mathcal{A}_{Critico}$ la richiesta per il servizio set_x_state ($\mathcal{A}_{Critico}.\mathsf{x\_state} = \mathcal{A}_{Replica}.\mathsf{x\_state} = 0$).

② Il metodo set_x_state è un metodo riflessivo, pertanto si ha l'inizio di una meta-computazione svolta da $\mathcal{MA}_{Critico}$ (Meta_MethodCall).

③ $\mathcal{MA}_{Critico}$ attiva il metodo Meta_HandleMethodCall che permette ad $\mathcal{A}_{Critico}$ di effettuare il servizio richiesto (set_x_state), modificando quindi il contenuto di x_state ($\mathcal{A}_{Critico}.\mathsf{x\_state} = 7$, $\mathcal{A}_{Replica}.\mathsf{x\_state} = 0$).

④ Quando $\mathcal{A}_{Critico}$ ha terminato l'elaborazione del servizio set_x_state, $\mathcal{MA}_{Critico}$ controlla lo stato di $\mathcal{A}_{Replica}$ rilevando (fault_detection) che la replica è fallita e quindi inizia le operazioni di recupero (fault_recovery).

⑤ Il metodo fault_recovery, siccome è stato attivato dal meta-oggetto dell'oggetto critico $\mathcal{MA}_{Critico}$, si limita a creare una nuova replica $\mathcal{A}'_{Replica}$, con lo stato consistente con quello di $\mathcal{A}_{Critico}$, per sostituirla a quella fallita.

⑥ $\mathcal{MA}_{Critico}$, recuperato $\mathcal{A}_{Replica}$, costruisce ed invia il checkpoint alla nuova replica (send_checkpoint).

⑦ Il meta-oggetto della replica $\mathcal{MA}'_{Replica}$, in seguito al ricevimento del checkpoint, attiva il metodo Meta_EventCallbackBody.

⑧ Il metodo Meta_EventCallbackBody gestisce, tramite la riflessione strutturale l'aggiornamento effettivo dello stato della replica $\mathcal{A}'_{Replica}$ consistentemente con quello di $\mathcal{A}_{Critico}$ ($\mathcal{A}_{Critico}$.x_state $=$ $\mathcal{A}'_{Replica}$.x_state $= 7$).

⑨ Avvenuto l'invio del checkpoint, $\mathcal{MA}_{Critico}$ termina la sua meta-computazione.

Terminata la computazione, schematizzata in figura [5.4], si ha che lo stato dell'oggetto critico è stato modificato come si voleva inoltre si è recuperato il fallimento della replica creandone una nuova, il cui stato è consistente con quello della copia primaria.
Si osserva che il fallimento della replica viene rilevato solamente nel momento in cui l'oggetto critico effettua una computazione che richiede la supervisione da parte del meta-oggetto.
Siccome, nell'esempio, si è considerato un sistema con grado di sicurezza pari a uno, un fallimento della replica non rilevato seguito da un fallimento della copia primaria non è più recuperabile. Costruendo il sistema con un grado di sicurezza maggiore si ha che le repliche secondarie finché non vengono interrotte per gestire l'aggiornamento del loro stato, sono intente a rilevare eventuali fallimenti delle altre repliche (replica primaria ed altre repliche secondarie), in questo modo con almeno due repliche secondarie (cioè con grado di sicurezza del sistema pari a due) si rileva in qualsiasi momento il fallimento di un'altra replica o dell'oggetto critico.
Siccome le repliche degli oggetti critici, finché svolgono il ruolo di replica, si limitano a subire l'aggiornamento dello stato, il loro ripristino è indipendente dal momento in cui si è effettivamente verificato il fallimento. Questo significa che la nuova replica può venire creata con uno stato (purché consistente con lo stato dell'oggetto critico) successivo o precedente a quello che aveva la replica fallita senza problemi.

5.4.6.3 Fallimento dell'oggetto critico $\mathcal{A}_{Critico}$

I meta-oggetti delle repliche dal momento in cui vengono creati sono dediti al rilevamento di eventuali fallimenti delle altre repliche o della replica primaria e vengono distolti dal loro compito esclusivamente per aggiornare lo stato del proprio oggetto riflessivo in seguito al ricevimento di un checkpoint dello stato della copia primaria. Se la copia primaria fallisce, i meta-oggetti delle repliche lo rilevano ed una di esse intraprende le operazioni di recupero, promuovendo il proprio oggetto riflessivo a copia primaria e creando, quindi, una replica secondaria sostitutiva.

Tornando all'esempio illustrato in cui il sistema ha grado di sicurezza pari a uno, si suppone che l'oggetto critico $\mathcal{A}_{\mathsf{Critico}}$ fallisca, vediamo allora nel dettaglio quali operazioni la replica $\mathcal{A}_{\mathsf{Replica}}$ ed il suo meta-oggetto $\mathcal{MA}_{\mathsf{Replica}}$ devono intraprendere per recuperare il fallimento di $\mathcal{A}_{\mathsf{Critico}}$.

① $\mathcal{MA}_{\mathsf{Replica}}$ è intento a controllare se $\mathcal{A}_{\mathsf{Critico}}$ fallisce (fault_detection).

② Rilevato il fallimento, $\mathcal{MA}_{\mathsf{Replica}}$ attiva le operazioni di recupero (fault_recovery) promuovendo $\mathcal{A}_{\mathsf{Replica}}$ a copia primaria e se stesso a meta-oggetto della copia primaria, a questo punto collega il nuovo oggetto critico al sistema, cioè fa si che le richieste indirizzate ad $\mathcal{A}_{\mathsf{Critico}}$ pervengano ad $\mathcal{A}_{\mathsf{Replica}}$. Quindi crea una nuova replica secondaria $\mathcal{A}'_{\mathsf{Replica}}$.

Procedendo in questo modo si ha il ripristino dell'oggetto critico in modo che possa continuare la computazione.

La computazione svolta dal nuovo oggetto critico riprende dall'inizio del servizio (metodo) che l'oggetto critico stava elaborando prima di fallire.
Nell'esempio presentato, la computazione del metodo set_x_state viene ripetuta dall'inizio. Il fatto che le repliche vengono aggiornate solo in seguito al completamento del servizio che si sta elaborando, consente in caso di fallimento di ripetere la computazione del servizio senza preoccuparsi di modifiche parzialmente effettuate che vengono ripetute.
Eventuali servizi richiesti ad altri oggetti da un oggetto fallito, quando vengono nuovamente richiesti dal nuovo oggetto critico se erano già stati soddisfatti (cioè il loro identificatore unico è contenuto nella coda dei servizi svolti) non vengono nuovamente elaborati evitando così la replicazione delle richieste di servizi ed eventuali effetti collaterali indesiderati.

Richieste di Servizi e Fallimento degli Oggetti Fornitori

Esiste un gap temporale durante il quale l'oggetto critico non è disponibile perché è fallito e non è stato ancora recuperato. Ovviamente in tale periodo, l'oggetto critico non può gestire alcuna richiesta di servizio. Quindi, se un oggetto richiede un servizio all'oggetto critico fallito, tale richiesta non viene soddisfatta neppure successivamente.
Questo perché la richiesta giunge all'oggetto fallito e questo non può inoltrarla alla replica, quindi quando la replica recupera il fallimento dell'oggetto critico ignora l'esistenza di quest'ultima richiesta ed ovviamente non la ripristina.
Questo problema non è dovuto alla strutturazione scelta e neppure agli algoritmi

descritti, bensì al modello di riflessione adottato. Infatti nel modello a meta-oggetto ogni meta-oggetto supervisiona esclusivamente i propri referenti, ignorando l'esistenza e l'operato degli altri oggetti. Affinché un meta-oggetto possa evitare la perdita di richieste deve avere come referenti tutti gli oggetti che possono richiedere servizi all'oggetto critico (quindi ogni meta-oggetto controlla tutti gli oggetti del sistema). Oppure il problema può essere risolto se il client, al momento della richiesta del servizio, verifica lo stato del server, ed in caso di fallimento attende il suo ripristino per risottomettergli la sua richiesta. Se la verifica viene effettuata direttamente dall'oggetto la politica di fault tolerance non risulta più incapsulata nei meta-oggetti impedendone un agevole riuso. Tale soluzione può essere implementata in modo trasparente, preservandone l'incapsulazione della politica di fault tolerance, dotando ogni oggetto, anche non critico, di meta-oggetto che per ogni richiesta di servizi inoltrata ad un altro oggetto ne verifica lo stato attendendone il ripristino. La conseguenza di questa soluzione è che si deve dotare ogni oggetto di meta-oggetto, introducendo l'uso delle meta-computazioni per ogni computazione, per il fatto che a priori non si può determinare quali oggetti richiedono servizi ad oggetti critici.

5.4.6.4 Fallimento dei Meta-Oggetti $\mathcal{MA}_{Critico}$ o $\mathcal{MA}_{Replica}$

In OpenC++ il meta-oggetto e l'oggetto a cui è associato condividono lo stesso processo e siccome i fallimenti che si vuole recuperare sono dovuti all'hardware, è praticamente impossibile distinguere il fallimento di un oggetto dal fallimento del suo meta-oggetto ed analogamente per quanto riguarda rilevamento e recupero. Perciò il caso del fallimento di $\mathcal{MA}_{Critico}$ è riconducibile al caso del fallimento di $\mathcal{A}_{Critico}$ ed il caso del fallimento di $\mathcal{MA}_{Replica}$ è riconducibile al caso del fallimento di $\mathcal{A}_{Replica}$.

5.5 Replicazione e Canali

Adottando il modello a materializzazione del canale l'approccio alla replicazione risulta differente rispetto al caso in cui si utilizza il modello a meta-oggetto (sezione [5.4]).

Nel modello a meta-oggetto i meta-oggetti sovraintendono esclusivamente l'interazione tra oggetto critico e repliche.
Nel modello a materializzazione del canale i canali sovraintendono l'intera evoluzione del servizio richiesto.

5.5.1 Tolleranza ai Guasti del Dizionario

Per poter garantire la tolleranza ai guasti di un sistema software, realizzato utilizzando il modello a materializzazione del canale, occorre che il dizionario dei canali stesso sia tollerante ai guasti.

La tolleranza ai guasti del dizionario dei canali può essere ottenuta in diversi modi. In funzione del tipo di errori che si intende gestire ed, inoltre, supponendo che l'implementazione del dizionario sia priva di errori logici, è sufficiente usare la replicazione per renderlo fault tolerant.
Un possibile meccanismo consiste nel replicare, in modo passivo, il dizionario stesso. Essendo il dizionario nascosto al programmatore dell'applicazione, l'uso di questo meccanismo comporta la modifica del sistema riflessivo.
Un altro possibile meccanismo si basa sulla duplicazione, all'interno del canale, delle informazioni contenute nel dizionario. Non tutte le informazioni devono essere replicate per ogni canale. Per ogni canale è sufficiente mantenere i riferimenti agli altri canali instaurati con lo stesso sender. Come si è visto nella sezione [3.6], il metodo send ha la possibilità di recuperare l'ultimo canale utilizzato. Quindi, in caso di fallimento del dizionario, può reperire facilmente un canale e da quest'ultimo riuscire a risalire al canale da utilizzare. Questa soluzione comporta la costruzione di una classe discendente da ReflObj, in cui si ridefinisce il metodo send in modo che gestisca questa estensione.
Questi meccanismi non vengono ulteriormente approfonditi, essendo più interessante esaminare come si può usare la riflessione per costruire applicazioni fault tolerant che rendere fault tolerant un sistema riflessivo. Nel seguito della trattazione si suppone che il dizionario non fallisca o che comunque sia tollerante ai guasti.

5.5.2 Modellazione della Replicazione

Sfruttando la riflessione, un'applicazione è strutturabile in più livelli (torre riflessiva) e le entità operanti in ogni singolo livello cooperano per espletare un certo aspetto, funzionale o meno, dell'applicazione.
In particolare, adottando il modello a materializzazione del canale, la tolleranza ai guasti ottenuta per replicazione è il risultato della cooperazione di diversi canali.
La struttura della torre riflessiva cambia al variare del ruolo che assumono i canali all'interno dell'applicazione, in particolare in base a come ed a quanti canali vengono utilizzati per ottenere i risultati desiderati.
Di seguito vengono descritti tre possibili modelli per la replicazione ottenuti utilizzando i canali.

5.5.2.1 Partizionamento Orizzontale della Gestione della Tolleranza ai Guasti

L'applicazione viene strutturata in una torre riflessiva composta da due livelli, il livello base ed il meta-livello.

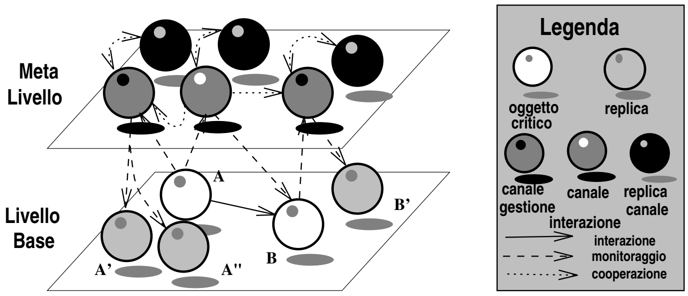


Figura 5.5: Applicazione Riflessiva con Aspetto Fault Tolerant Partizionato Orizzontalmente

Nel livello base operano gli oggetti che compongono l'applicazione e le repliche degli oggetti critici. Nel meta-livello operano i canali, il cui compito è quello di gestire la tolleranza ai guasti dell'applicazione.
Si distinguono i canali, in base al compito che svolgono ed agli oggetti che connettono, in due generi: *canali di gestione* e *canali di interazione*.
I canali di gestione sono utilizzati per creare le repliche, per mantenere la consistenza dello stato delle repliche con lo stato dell'oggetto critico e per rilevare e quindi recuperare un eventuale fallimento dell'oggetto critico o delle sue repliche.
I canali di interazione gestiscono l'interazione che coinvolge uno o più oggetti critici (e quindi anche le loro repliche). Il loro compito è quello di supervisionare la corretta esecuzione dei servizi che coinvolgono, sia nel ruolo di client che di server, degli oggetti critici. Ogni oggetto critico può avere più canali di interazione instaurati con diversi altri oggetti critici o meno. Ognuno di questi canali coordina il proprio operato con il canale di gestione, richiedendogli informazioni sull'oggetto critico ed informandolo che lo stato dell'oggetto critico, che supervisiona, ha subito delle modifiche che devono essere propagate alle repliche.
La richiesta inoltrata da un oggetto critico ad un altro viene intercettata dal rispettivo canale di interazione che provvede ad inoltrare al receiver la richiesta del sender. Il canale di interazione segnala al canale di gestione del receiver la terminazione della computazione. Quest'ultimo comunica al canale di gestione del receiver l'accaduto, il quale provvede ad aggiornare le repliche del receiver; analogamente i valori di ritorno vengono forniti al sender del canale di interazione ed al canale di gestione del sender, che provvede a fornirli alle repliche del sender. Quando oggetto fallisce tutti i canali (sia di interazione che di gestione) ad esso collegati vengono automaticamente rimossi (in base al ciclo di vita dei canali). Le repliche dei canali rilevano che i canali sono stati rimossi e provvedono al recupero della computazione. Le repliche dei canali di gestione promuovono una replica secondaria a primaria e si connettono. Le repliche dei canali di interazione attendono il recupero del fallimento da parte della replica del canale di gestione, quindi si connettono con il nuovo oggetto critico e recuperano la computazione interrotta.
Le operazioni di gestione della fault tolerant sono ripartite orizzontalmente tra più canali che devono comunicare e sincronizzarsi tra loro per effettuare le operazioni di recupero correttamente.
Una simile strutturazione (schematizzata in figura [5.5]) presenta il vantaggio di non avere un meta-meta-livello, riducendo le operazioni di shift-up e di shift-down, e lo

svantaggio di introdurre delle comunicazioni tra canali che obbligano a definire opportune politiche di sincronizzazione tra canali. Inoltre se un oggetto ha un grado di criticità tale da richiedere diverse repliche, il corrispondente canale di gestione associato deve avere la possibilità di gestire un gruppo di oggetti come receiver o più receiver contemporaneamente per poter far evolvere tutte le repliche.

5.5.2.2 Partizionamento Verticale della Gestione della Tolleranza ai Guasti

L'applicazione viene strutturata in una torre riflessiva composta da tre livelli: il livello base, il meta-livello ed il meta-meta-livello.

Nel livello base operano gli oggetti che espletano l'aspetto funzionale dell'applicazione e le repliche degli oggetti critici. Nel meta-livello operano i canali che supervisionano direttamente la computazione degli oggetti operanti nel livello base e le loro repliche. Nel meta-meta-livello operano dei canali (chiamati, per chiarezza, meta-canali) che supervisionano l'operato dei sottostanti canali.

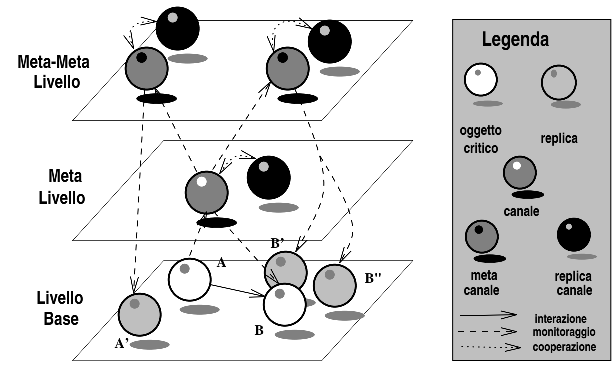


Figura 5.6: Applicazione Riflessiva con Aspetto Fault Tolerant Partizionato Verticalmente

La gestione della tolleranza ai guasti è ripartita (verticalmente) tra il meta-livello ed il meta-meta-livello ed è il risultato della cooperazione tra canali e meta-canali.
I canali sono instaurati tra oggetti critici, e si occupano di supervisionare le richieste di servizi che coinvolgono i due referenti in qualità di client (il sender) e di server (il receiver), garantendo il completamento di ogni richiesta inoltrata ad un oggetto critico, anche in caso di fallimento. Ogni canale ha una replica che tiene traccia del progredire della computazione.
I meta-canali sono instaurati tra un canale (in qualità di sender) e le repliche dei referenti di tale canale (in qualità di receiver). I meta-canali intercettano le modifiche effettuate dal canale loro referente allo stato dell'oggetto critico, la cui replica è il loro altro referente, e le effettuano anche sulla replica dell'oggetto critico che monitorizzano, facendolo evolvere concorrentemente all'oggetto critico.
Si ha, quindi, che quando un oggetto richiede ad un oggetto critico un servizio, questa richiesta viene intercettata da un canale instaurato tra di loro. Il canale inoltra effettivamente la richiesta al receiver. Le modifiche allo stato del receiver vengono intercettate dai meta-canali che le riportano alle repliche.
In caso di fallimento di un oggetto critico, in accordo con il ciclo di vita proprio del modello a materializzazione del canale, i canali instaurati con l'oggetto riflessivo fallito ed i suoi meta-canali vengono eliminati. Il recupero della computazione viene

effettuato dalle repliche dei canali (tenute aggiornate durante la normale computazione), il cui compito è quello di controllare sistematicamente se il canale di cui sono repliche sia stato rimosso (se il canale è stato rimosso vuol dire che si è verificato un fallimento). Se rilevano la rimozione dei canali, le repliche si agganciano ad una replica dell'oggetto fallito ed al vecchio referente non fallito e riprendono la computazione dal punto in cui è stata interrotta.
La modellazione della replicazione a partizionamento verticale, schematizzata in figura [5.6], è stata presentata in [Ancona-Cazzola-Dodero-Gianuzzi 95] e presenta il vantaggio che i canali non interagiscono (cioè non comunicano) tra loro, limitando, quindi, le comunicazioni a quelle previste dal modello a materializzazione del canale (ed in particolare dalle sue implementazioni). In questo modo non è necessario coordinare|sincronizzare l'operato delle diverse meta-entità.

5.5.2.3 Replicazione dell'Interazione

Questo modello considera un'interazione nel suo complesso come un'unica entità e non come composta da tante entità che interagiscono. Una simile concezione conduce ad una situazione in cui la replica di un canale instaurato tra due oggetti critici è a sua volta instaurata tra le repliche di tali oggetti critici. In questo modo è come se si replicasse l'intera interazione.

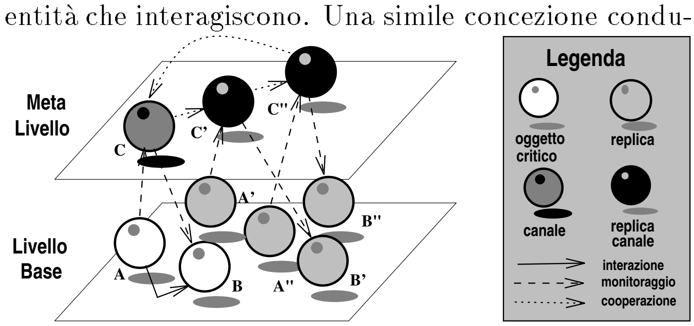


Figura 5.7: Applicazione Riflessiva Fault Tolerant in cui viene replicata l'interazione

L'applicazione viene strutturata in due livelli: il livello base ed il meta-livello. Nel livello base operano le entità che svolgono l'aspetto funzionale dell'applicazione e le repliche degli oggetti critici. Nel meta-livello operano i canali, il cui scopo è quello di supervisionare la computazione e di fare evolvere le repliche consistentemente con gli oggetti critici.
Per ogni coppia di oggetti critici che interagiscono, vi è un canale (primario) che monitorizza la loro computazione, mentre ogni replica del canale primario monitorizza una coppia di repliche dei referenti del canale primario.
La richiesta di un servizio inoltrata da un oggetto ad un altro viene intercettata dal canale instaurato tra di essi. Il canale supervisiona la computazione del servizio e quindi provvede ad informare la proprie repliche di aggiornare di conseguenza le repliche dei suoi referenti.
I canali e le loro repliche sono logicamente ordinate in modo circolare (cioè l'ultimo canale dell'ordinamento precede il primo dell'ordinamento) ed ogni canale controlla quello successivo nell'ordinamento.
Se un referente fallisce, i canali ad esso associati vengono rimossi (in accordo con il ciclo di vita del canale), interrompendo la catena di controlli. Se un canale fallisce

la catena viene automaticamente interrotta. In questo modo il canale (o la replica del canale) precedente nell'ordinamento rileva il fallimento e procede al recupero. Il recupero consiste nel ricreare una terna sender, receiver e canale (riutilizzando l'entità sopravissuta) con lo stato della terna del canale che provvede al recupero.
Una strutturazione simile (schematizzata in figura [5.7]) presenta il vantaggio di limitare il numero di meta-entità che supervisionano un'interazione, inoltre canali diversi non devono, necessariamente, cooperare tra loro per ottenere la tolleranza ai guasti ed i diversi compiti sono svolti tutti da entità analoghe, ottenendo la gestione della replicazione tramite un unico genere di canali. Gli svantaggi sono rappresentati dal fatto che le repliche dei canali ricoprono un ruolo attivo e la loro elaborazione richiede di essere sincronizzata con quella delle altre repliche e del canale primario. Inoltre, a causa della peculiare concezione della replicazione che adotta, questa organizzazione può essere utilizzata solo nel caso in cui gli oggetti critici che interagiscono siano replicati lo stesso numero di volte. Questa limitazione può essere superata instaurando i canali tra le repliche ed uno pseudo-oggetto.

5.5.3 Replicazione Passiva e Modello a Materializzazione del Canale

Confrontando le diverse strutturazioni presentate, ricordando che sono solo alcune di quelle possibili, si osserva che l'approccio a replicazione dell'interazione utilizza, per gestire un'interazione, un numero di canali inferiore a quello degli altri approcci e che i canali che gestiscono l'aspetto fault tolerant dell'applicazione sono tutti dello stesso genere, riducendo il codice da dover scrivere, essendo i canali tutti istanza della stessa classe.
In base a queste osservazioni si illustra come realizzare la replicazione passiva, adottando la modellazione a replicazione dell'interazione.

La tecnica di replicazione passiva consiste nel mantenere una replica completa dell'oggetto critico, aggiornandone lo stato consistentemente con quello dell'oggetto critico ad ogni azione di checkpoint. In caso di fallimento di un oggetto critico la replica consente il ripristino della computazione.

Per caratterizzare il modello generale in modo da implementare la replicazione passiva si devono raffinare i seguenti concetti:

- Azioni di aggiornamento, in particolare da chi, come e quando devono essere effettuate.
- Azioni di rilevamento e recupero dei fallimenti, in particolare da chi, come e quando devono essere effettuate.

Si deve effettuare il raffinamento di tali concetti perché sono questi i concetti fondamentali che differenziano i diversi approcci alla replicazione. L'algoritmo per la replicazione passiva adottato è dovuto a Powell [Powell 91] ed è illustrato nella sezione [4.4.1.1].

Azioni di Aggiornamento: Il mantenimento dello stato delle repliche consistente con quello degli oggetti critici è effettuato dai canali. I canali inerenti la stessa interazione sono virtualmente connessi ad anello ed ognuno di essi comunica con quello precedente e quello successivo nell'anello. Il canale primario, subito dopo che il suo receiver ha effettuato la computazione, costruisce il checkpoint dello stato del receiver e lo invia al successivo canale dell'anello, che aggiorna lo stato della replica che monitorizza. Quindi lo invia al successivo, che si comporta in modo analogo finché il checkpoint non giunge nuovamente al canale primario che riprende la computazione riflessiva, riportando i risultati della computazione al sender; quindi costruisce il checkpoint dello stato del sender ed effettua la relativa azione di aggiornamento per le repliche del sender, analoga a quella descritta per il receiver.
Gli aggiornamenti e la costruzione del checkpoint dello stato degli oggetti critici e delle repliche possono essere effettuati sfruttando la riflessione strutturale oppure utilizzando apposite primitive fornite dal sottostante sistema operativo.

Azioni di Rilevamento e Recupero dei Fallimenti: Il rilevamento e l'eventuale recupero dei fallimenti è gestito dai canali. I canali facenti parte di un certo anello sono repliche l'uno dell'altro. L'algoritmo di rilevamento è basato sul ciclo di vita del modello a materializzazione del canale, secondo il quale un canale esiste finché entrambi gli oggetti tra cui è instaurato continuano ad esistere. Quindi, quando un oggetto cessa la propria attività, il sistema provvede ad eliminare tutti i canali ad esso associati (vedere i capitoli [1], [2] e [3]).
Ogni canale controlla il canale successivo nell'anello. Quando il canale rileva la rimozione del canale che controlla, vuol dire che è occorso un fallimento che ha coinvolto uno dei due referenti o lo stesso canale. Quando un canale rileva un fallimento, determina, al fallimento di quale entità (sender o receiver) è dovuta la rimozione del canale che controlla. In base a tale informazione crea una nuova replica (se non è già stata creata da un altro canale) ed instaura un canale tra queste entità. Il canale e la replica vengono creati con lo stato aggiornato in base allo stato del canale che provvede al recupero ed agli oggetti che controlla. Il canale così creato sostituisce il canale rimosso prendendone il posto anche all'interno dell'ordinamento imposto dall'anello.
Sfruttare il ciclo di vita dei canali consente di limitare le entità da dover monitorare per rilevare un fallimento, rispetto all'approccio basato sul modello a meta-oggetto. È, infatti, sufficiente monitorare il canale e non i suoi referenti per determinare un fallimento di una qualsiasi delle tre entità.
I canali tengono anche traccia dell'evoluzione della computazione; questo consente di recuperare la computazione dal punto in cui è stata interrotta.

Pool di Repliche e di Canali

Il modello a materializzazione del canale fornisce una granularità più fine rispetto al modello a meta-oggetto. La granularità più fine si manifesta con la possibilità di avere canali diversi che replicano interazioni diverse, nonostante che queste interazioni coinvolgano un oggetto comune.
I diversi canali devono, comunque, operare in modo che vengano rispettate le seguenti regole:

- Lo stato delle repliche, affinché esse siano effettivamente tali, deve subire tutte le modifiche che subisce lo stato degli oggetti di cui sono repliche.
- Un oggetto fallito deve essere rimpiazzato da un solo sostituto.

Quindi ogni replica è controllata dallo stesso numero e genere di canali (repliche) dell'oggetto critico di cui è replica. A loro volta ognuno di questi canali è la replica di uno dei canali instaurati con l'oggetto critico. In seguito ad un fallimento, un canale, prima di creare una nuova replica, deve verificare se l'oggetto non è già stato sostituito da un altro canale.
Allo scopo si introduce il concetto di *pool di repliche*. L'insieme delle repliche di un oggetto critico costituisce il pool associato a tale oggetto critico. Ogni replica appartenente al pool è identificabile all'interno del pool, a prescindere dalla propria identità di oggetto, tramite degli interi progressivi. Associando ai canali un intero progressivo si stabilisce una relazione biunivoca tra i canali e gli elementi dei pool di repliche. In questo modo anche i canali formano un pool (l'anello di canali descritto in precedenza) ed ogni canale del pool di canali è instaurato tra le corrispondenti, in base al numero progressivo, repliche dei pool di repliche. In questo modo ogni replica è monitorata dallo stesso numero di canali che monitorizzano l'oggetto critico replicato. In caso di fallimento la nuova replica prende il posto nel pool della replica fallita e quindi il numero progressivo ad essa assegnato. I canali verificano se la replica fallita è stata sostituita, controllando se il posto nel pool è ancora vacante. Nel caso in cui il posto sia vacante creano la nuova replica ed instaurano un canale tra essa ed il referente non fallito, altrimenti si limitano ad agganciare il nuovo canale alle due repliche già esistenti. In entrambi i casi il canale creato prende il posto del canale rimosso nel pool di canali.

5.5.4 Implementazione in PVM della Replicazione dell'Interazione

La modellazione descritta è ottenuta utilizzando dei canali istanze della classe repl_channel che si descrive di seguito.

Al momento della prima interazione tra due oggetti critici viene instaurato tra di essi un canale istanza della classe repl_channel.
Il canale instaurato tra gli oggetti critici è detto canale primario ed ha il compito di:

- Creare le repliche dei propri referenti (se non sono già state create da un altro canale primario) e di organizzarle in pool.
- Creare le proprie repliche, chiamate canali secondari, e di agganciarle alle repliche dei referenti.
- Gestire l'evoluzione delle computazioni che avvengono tra i propri referenti.

I canali secondari, invece, in seguito a modifiche dello stato degli oggetti monitorati dal canale primario, hanno il compito di:

- Aggiornare lo stato dei propri referenti consistentemente con le modifiche effettuate dal canale primario ai propri referenti.

Sia il canale primario che i canali secondari hanno il compito di rilevare e quindi recuperare un eventuale fallimento del canale successivo nell'ordinamento (per il ciclo di vita del modello a materializzazione del canale equivale a rilevare e recuperare il fallimento di un oggetto critico).

Sia a livello di modello che di implementazione, per semplicità, si suppone che gli oggetti interagenti abbiano lo stesso grado di criticità. È comunque possibile gestire interazioni in cui il client ed il server hanno grado di criticità diverso. Per gestire un'interazione di due oggetti critici con grado di criticità, rispettivamente, α e β con $\beta > \alpha$ si progetta la classe dei canali che implementano la replicazione dell'interazione in modo che il canale primario crei β canali secondari. I primi α canali secondari sono instaurati tra le α repliche del referente meno critico e le prime α repliche del referente più critico, mentre i rimanenti $\beta - \alpha$ canali secondari sono instaurati tra le rimanenti $\beta - \alpha$ repliche del referente più critico ed uno pseudo-oggetto (ad esempio i target descritti nella sezione [2.4]). Le azioni di checkpoint sono gestite come descritto dai primi α canali. I rimanenti $\beta - \alpha$ canali gestiscono normalmente i checkpoint dei referenti effettivi, mentre i checkpoint destinati ai referenti fittizi vanno persi. Le operazioni di rilevamento e recupero sono gestite analogamente, prestando attenzione ai canali di posizione α e $\alpha + 1$, in quanto il canale $\alpha + 1$ sovraintende il recupero del canale α però non possiede le informazioni per il ripristino di tutte le entità coinvolte (può comunque ottenere le informazioni mancanti dagli altri canali, ad esempio dal canale primario). L'implementazione che si presenta considera solamente interazioni tra oggetti con lo stesso grado di criticità.

I pool, descritti in precedenza, possono essere implementati in più modi, ad esempio utilizzando delle strutture dati condivise dai diversi canali, che mantengono i riferimenti e la posizione nel pool delle diverse entità.
In PVM è possibile utilizzare i gruppi di processi per implementare i pool. Gli oggetti attivi appartenenti ad uno stesso pool appartengono ad uno stesso gruppo e la loro posizione nel pool è rappresentata dal numero di istanza che PVM associa

ad ogni membro di un gruppo. Un numero di istanza associato ad un processo che per scelta od in seguito ad un fallimento lascia il gruppo viene liberato (cioè non è più associato a nessun processo|oggetto) e viene assegnato automaticamente alla prima entità che si associa a quel gruppo. Quindi per verificare il fallimento di un oggetto attivo è sufficiente controllare se esiste nel gruppo un processo associato al suo numero di istanza. In fase di recupero quando l'oggetto sostituto viene arruolato nel gruppo gli viene assegnato il numero di istanza dell'oggetto che ha sostituito.

Nella modellazione presentata i checkpoint vengono passati da un canale all'altro, seguendo l'ordinamento imposto dalla posizione dei canali nel pool. Utilizzando PVM si può sfruttare la comunicazione in broadcast comunicando il checkpoint in una sola volta a tutti i canali. Questa è una possibile soluzione, semplice da realizzare, che presenta, però, il rischio di deadlock. Il canale primario, inviato il checkpoint, attende che tutti i canali secondari abbiano gestito l'aggiornamento del proprio referente e gli invino un segnale di acknowledge (situazione gestita dalla primitiva *pvm_barrier*). Il deadlock si manifesta se un canale secondario non invia il proprio acknowledge in seguito ad un fallimento. Il deadlock può essere evitato utilizzando implementazioni diverse, basate ad esempio su attese con timeout o su sincronizzazioni tra canali adiacenti nell'ordinamento (*token passing*). Questa implementazione, comunque, è adatta alla descrizione che si vuole fare ed è più semplice da implementare.

repl_channel: i canali per la replicazione dell'interazione

Tutti i canali necessari alla replicazione dell'interazione sono istanza della stessa classe (repl_channel), sia che siano canali primari che secondari. Di seguito è riportata l'interfaccia della classe repl_channel.

```
class repl_channel public: channel {
  private:
    int channel_pool_number, replica_number, progress_counter = 0, memo_n;
      // eseguibili del sender, receiver e del canale
    char *sender_filename, *receiver_filename, channel_filename[] = ''repl_channel'';
      // pool (implementati come gruppi) di sender, receiver e canali
    char *sender_pool, *receiver_pool, *channel_pool;
    void **memo_args, *(*memo_f)(ReflObj, int, void **);
  public:
    // costruttore
      repl_channel( int s, int r, char *t, int cpn = 0, int rn = 2 );
    // distruttore
      ~repl_channel( void ) { pvm_lvgroup( channel_pool ) };
    // metodi per la gestione della riflessione
      void HandleMessage( void *(*f)(ReflObj, int, void **), int n, void **args );
    // metodi per la creazione delle repliche
      void create_channel_replica( int i );
      void create_sender_replica( int i );
      void create_receiver_replica( int i );
    // metodi per la gestione del rilevamento e recupero dai fallimenti
      void wait_event( void );
      void fault_recovery( void );
```

```
    void recovery( void );
  // metodi per la gestione degli aggiornamenti
    void update_sender( void );
    void update_receiver( void );
    void update( void );
    void make_checkpoint_sender( void );
    void make_checkpoint_receiver( void );
    void make_checkpoint( void );
}
```

Il costruttore dei canali stabilisce il proprio comportamento in base al tipo di canale che si sta creando. Se si sta creando un canale secondario, questo una volta creato, si pone in un ciclo infinito, in attesa di un evento (**wait_event**) che può essere la ricezione di un checkpoint od il rilevamento della rimozione del canale precedente nel pool di canali. Se si sta creando un canale primario per la prima volta (riconoscibile in quanto è il primo ad arruolarsi al pool di canali ed ha numero di istanza pari a 1), esso, una volta creato, provvede a creare le repliche del sender e del receiver (sempre che non siano già state create da un altro canale primario) ed ad organizzarle nei corrispondenti pool (cioè arruola tali repliche al rispettivo gruppo) quindi si occupa di creare le proprie repliche e di costituire il pool di canali. Se non è la prima volta che viene creato il canale primario allora vuol dire che si sta recuperando il fallimento della terna primaria e si procede di conseguenza attivando il metodo **recovery**. Di seguito è riportato il codice del costruttore.

```
repl_channel::repl_channel( int s, int r, char *t, int cpn = 0, int rn = 2) :
        channel( s, r, t ), channel_pool_number(cpn), replica_number(rn), reply(LOCKING_BEHAVIOUR) {
                      ⋮
  // determinazione dell'eseguibile dei sender e dei receiver e del nome dei pool
                      ⋮

  pvm_joingroup( channel_pool );                     // arruolamento del canale nel pool di canali
  if ( channel_pool_number != 0 ) wait_event();      // è stato creato un canale replica
    // è stato creato il canale primario
  if ( pvm_gsize( sender_pool ) == 1 )               // verifica se esistono le repliche del sender
    for( int i=1; i<replica_number; i++ )            // se no le si crea
      create_sender_replica( i );
  if ( pvm_gsize( receiver_pool ) == 1 )             // verifica se esistono le repliche del receiver
    for( int i=1; i<replica_number; i++ )            // se no le si crea
      create_receiver_replica( i );
  if ( pvm_gsize( channel_pool ) == 1 )              // verifica se esistono le repliche del canale
    for( int i=1; i<replica_number; i++ )            // se no le si crea
      create_channel_replica( i );
  else recovery();                                   // recupero del canale primario
}
```

La meta-computazione svolta dai canali istanza dell classe **repl_channel** è di tipo bloccante (comportamento stabilito dall'attributo **reply**). Questo comportamento garantisce la modifica di un oggetto critico da parte di un solo canale alla volta e quindi il ripristino può essere effettuato tranquillamente senza preoccuparsi di computazioni in progresso svolte da canali diversi.

Tralasciando il costruttore, i metodi della classe repl_channel possono essere distinti in tre categorie: *metodi per la gestione della riflessione, metodi per la gestione del rilevamento e del recupero dei fallimenti* e *metodi per la gestione delle repliche*.

Metodi per la gestione della riflessione

Il comportamento riflessivo è definito dal metodo HandleMessage. Il metodo HandleMessage viene attivato solo dal canale primario in seguito alla richiesta di un servizio inoltrata dal proprio sender al proprio receiver. Il metodo HandleMessage verifica se il canale precedente nell'ordinamento è stato rimosso (siccome il canale che attiva il metodo è il primo dell'ordinamento quello che viene controllato è il canale con numero di istanza più grande, cioè l'ultimo dell'ordinamento (ordinamento circolare)). In caso di rimozione del canale controllato si iniziano le operazioni di recupero attivando il metodo fault_recovery.
Il metodo HandleMessage supervisiona la computazione dei due referenti del canale e provvede ad inviare ai canali secondari il checkpoint dello stato dei referenti, quando la computazione effettua delle modifiche allo stato dei referenti.
L'elaborazione svolta dal metodo HandleMessage può essere suddivisa in tre tappe utili per il recupero della computazione che si sta elaborando.

- La prima tappa è successiva alla notifica ai canali secondari della computazione riflessiva che si sta elaborando.
- La seconda tappa è successiva al completamento della computazione svolta dal receiver (apply) ed all'invio del checkpoint del receiver ai canali secondari.
- La terza tappa è successiva al completamento della meta-computazione e quindi al ritorno dei risultati della computazione al sender ed alla notifica del checkpoint del sender agli altri canali.

La definizione di queste tappe stabilisce dei punti da cui è possibile continuare la computazione consistentemente con l'ultimo checkpoint effettuato e quindi senza ripetere parti della computazione già effettuate (e soprattutto già notificate alle repliche).
Per evitare, in seguito alla ripetizione di parte della computazione svolta, di fornire|richiedere più volte agli oggetti non falliti lo stesso servizio, i canali che ne monitorizzano la richiesta mantengono le code dei servizi richiesti e dei risultati forniti, come nell'algoritmo presentato da Powell in [Powell 91], e quando viene nuovamente richiesto un servizio già fornito il canale se ne avvede consultando la relativa coda e grazie alla coda dei risultati forniti restituisce al client il risultato corretto senza ricalcolarlo (utile quando si recupera una computazione interrotta).
I servizi sono identificati unicamente tramite, ad esempio, un identificatore composto dal "nome del client" e dal valore che ha il "program counter" del client al

momento in cui inoltra la richiesta (in caso di fallimento anche il program counter viene ripristinato al valore che aveva al momento dell'ultimo checkpoint).
Un servizio richiesto due volte viene gestito entrambe le volte dallo stesso canale o da un suo sostituto perché le due richieste hanno gli stessi requisiti (sender, receiver e genere). Le code vengono replicate in ogni membro del pool di canali.

Di seguito è riportato il codice del metodo HandleMessage.

```
void repl_channel::HandleMessage( void *(*f)(ReflObj, int, void **), intn, void *args ) {
  if ( pvm_gettid( channel_pool, pvm_gsize(channel_pool) ) == PvmNoInst ) fault_recovery();
  if ( !deja_requested() ) {
    if ( progress_counter == 0 ) {
      pvm_initsend(PvmDataRaw);
      pvm_pkmeth(f); memo_f = f;
      pvm_pkint( &n, 1, 1 ); memo_n = n;
      pvm_pkvoidp( args, n, 1 ); memo_args = args;
      pvm_bcast( channel_pool, UPDATE );
      pvm_barrier( channel_pool, pvm_gsize( channel_pool ) );
      progress_counter = 1; ____________________ prima tappa ____________________
    }
    if ( progress_counter == 1 ) {
      int rbufid = pvm_recv( receiver, APPLY_RESPONSE );
      make_checkpoint_receiver();
      pvm_bcast( channel_pool, UPDATE_RECEIVER );
      pvm_barrier( channel_pool, pvm_gsize( channel_pool ) );
      progress_counter = 2; ____________________ seconda tappa ____________________
    }
    if ( progress_counter == 2 ) {
      if (info == 0) {
        pvm_setsbuf( rbufid );
        pvm_send( sender, APPLY_RESPONSE );
        make_checkpoint_sender();
        pvm_bcast( channel_pool, UPDATE_SENDER );
        pvm_barrier( channel_pool, pvm_gsize( channel_pool ) );
      } else
        if ( reply ) pvm_send( sender, ACK_REQUEST );
      progress_counter = 0; ____________________ terza tappa ____________________
      set_deja_requested();
    }
  } else recovery_reply();
}
```

I metodi deja_requested, set_deja_requested e recovery_reply si limitano a visitare ed a modificare i dati relativi alla gestione dei servizi già forniti, perciò non si dettaglia ulteriormente il loro codice.

Metodi per la Gestione del Rilevamento e del Recupero dei Fallimenti

Il rilevamento si basa sul ciclo di vita del canale definito dal modello a materializzazione del canale. Un canale rileva un fallimento quando rileva la rimozione del canale che controlla. Il rilevamento è svolto dal metodo wait_event mentre il recupero è lasciato ai metodi fault_recovery e recovery.

wait_event: L'attivazione di questo metodo pone il canale secondario in attesa di un evento. Gli eventi consistono nel rilevamento di un fallimento o nella ricezione di un checkpoint inviato dal canale primario.
Il canale secondario controlla in successione se il canale precedente nell'ordinamento è stato rimosso e se ha ricevuto un checkpoint da parte del canale primario. Se rileva la rimozione del canale, vuol dire che si è verificato un fallimento ed inizia le operazioni di recupero (fault_recovery). Se, invece, riceve un checkpoint, in base a quale oggetto si riferisce tale checkpoint, provvede ad aggiornare lo stato della replica corrispondente (update, update_sender e update_receiver). Inoltre provvede ad aggiornare l'evoluzione della meta-computazione. Di seguito è riportato il codice del metodo wait_event.

```
void repl_channel::wait_event( void ) {
  int rbufid, msg_tag;
  int check_channel_number = (channel_pool_number -1) % replica_number;
  while( TRUE ) {
    if ( pvm_gettid( channel_pool, check_channel_number ) ) == PvmNoInst ) fault_recovery();
    if ( rbufid = pvm_nrecv( any_proc, any_tag ) > 0 ) {
      pvm_bufinfo( rbufid, (int *)0, &msg_tag, (int *)0 );
      switch( msg_tag ) {
        case RECOVERY_SENDER:
        case UPDATE_SENDER: update_sender();
              process_counter = 0;
              break;
        case RECOVERY_RECEIVER:
        case UPDATE: update();
              process_counter = 1;
              break;
        case RECOVERY:
        case UPDATE_RECEIVER: update_receiver();
              process_counter = 2;
              break;
      }
      if ( msg_tag == UPDATE_SENDER || msg_tag == UPDATE_RECEIVER || msg_tag == UPDATE )
        pvm_barrier( channel_pool, pvm_gsize( channel_pool ) );
    }
  }
}
```

fault_recovery: La fase di recupero del fallimento consiste nel verificare quale referente sia fallito e, quindi, nel creare un sostituto dell'oggetto fallito e del canale rimosso e nell'aggiornare lo stato di entrambe le nuove entità. Il fallimento viene rilevato quando si rileva che il numero di istanza del pool di canali non è assegnato ad alcun canale.
Il problema principale del recupero del fallimento di un oggetto di posto i è dovuto al fatto che ci possono essere più canali di posto i+1 (appartenenti a pool diversi) che provvedono alla sostituzione generando sovrapposizioni. Il problema è evitabile progettando l'operazione di rilevamento e quella di sostituzione come un'unica azione atomica (*test and set*). Tralasciando il codice relativo al rendere atomiche

un'insieme di istruzioni (per il quale si rimanda alla consultazione di [Ben-Ari 90] e [Silberschatz-Peterson-Galvin 91]) il codice dell'operazione di **test_and_set** è:

```
int repl_channel::test_and_set( char *pool, char *filename, int position ) {
  int result = FALSE;
  // inizio regione atomica
  if (result = (pvm_gettid(pool, position) == PvmNoInst))
    create_replica( pool, filename ); // crea la nuova replica (filename) che viene arruolata a pool
  // fine regione atomica
  return result;
}
```

Di seguito è riportato il codice del metodo **fault_recovery** che svolge il recupero della computazione e che viene attivato ogni volta che un canale rileva che il precedente nell'ordinamento è stato rimosso.

```
void repl_channel::fault_recovery( void ) {
  int faulty_element = (channel_pool_number - 1) % replica_number;
  test_and_set( sender_pool, sender_filename, faulty_element );
  test_and_set( receiver_pool, receiver_filename, faulty_element );
  create_channel_replica( faulty_element );
  int ch = pvm_gettid( channel_pool, faulty_element );
  make_checkpoint_sender();
  pvm_send(ch, RECOVERY_SENDER);
  make_checkpoint_receiver();
  pvm_send(ch, RECOVERY_RECEIVER);
  make_checkpoint();
  pvm_send( ch, RECOVERY );
}
```

Nel fallimento di un oggetto vengono coinvolti diversi canali (tutti i canali instaurati con l'oggetto fallito). Per ogni canale rimosso esiste un canale nel rispettivo pool di canali che provvede al suo ripristino. Il ripristino di un canale consiste, innanzitutto, nel ripristino dei suoi referenti, cioè nella creazione dei referenti (se sono falliti) e nel ripristino del loro stato all'ultimo stato corretto.
I referenti sono di due tipi primari e, quindi, partecipanti in modo attivo alla computazione e repliche e che, quindi, subiscono la computazione. Dal punto di vista del ripristino dello stato dei referenti, la situazione è più delicata quando sono falliti degli oggetti primari. I referenti primari non falliti, al momento del recupero possono trovarsi in una delle seguenti tre situazioni:

- Ozianti (cioè non sono coinvolti in alcuna interazione).
- Coinvolti in un'interazione con l'oggetto fallito.
- Coinvolti in un'interazione con un altro oggetto non fallito.

Nei primi due casi si effettua il ripristino dello stato ad un punto antecedente a quello che hanno al momento del fallimento. Se il referente sta oziando (primo caso) non ha computazioni in corso, quindi il rollback dello stato non crea problemi. Nel secondo caso il ripristino all'ultimo stato sicuro è necessario per riprendere la computazione

interrotta.
Nel terzo caso l'oggetto è impegnato in un'interazione con un altro oggetto non fallito. La loro interazione è supervisionata da un'istanza della classe **repl_channel** che non viene rimossa (nessuno dei suoi referenti è fallito). Siccome la meta-computazione è bloccante, il nuovo canale, ricevuto il checkpoint, si accorge (operazione effettuata dai metodi di aggiornamento) della computazione in corso e non effettua l'aggiornamento, che, comunque, in questo caso è inutile.

recovery: Il metodo recovery viene attivato dal costruttore quando si procede al recupero della rimozione del canale primario e permette di ricevere, dal canale che sovraintende il recupero, i checkpoint degli oggetti critici per aggiornare i suoi referenti e permette anche di riprendere un eventuale meta-computazione interrotta dal fallimento. Di seguito è riportato il codice del metodo recovery.

```
void repl_channel::recovery( void ) {
  int msg_tag;
  while( TRUE ) {
    int rbufid = pvm_recv( any_proc, any_tag );
    pvm_bufinfo( rbufid, (int *)0, &msg_tag, (int *)0 );
    if ( msg_tag == UPDATE_SENDER ) update_sender();
    else if ( msg_tag == UPDATE_RECEIVER ) update_receiver();
      else { update(); break; }
  }
  if ( progress_counter != 0 )
    HandleMessage( memo_f, memo_n, memo_args );
}
```

Metodi per la Gestione delle Repliche

I metodi per la gestione delle repliche possono, a loro volta, essere suddivisi in tre categorie: metodi per la creazione delle repliche, metodi per la costruzione del checkpoint dello stato delle repliche e metodi per l'aggiornamento dello stato delle repliche.

Per ogni categoria la classe repl_channel definisce i metodi necessari per ogni tipo di replica del sistema (sender, receiver e canale).
I metodi per la costruzione del checkpoint (make_checkpoint, make_checkpoint_sender e make_checkpoint_receiver) ed i metodi per l'aggiornamento dello stato delle repliche (update, update_sender e update_receiver) sono collegati. I primi costruiscono il checkpoint che i secondi utilizzano per aggiornare lo stato delle repliche.
Per quanto riguarda sender e receiver, sia i metodi per la costruzione del checkpoint che quelli per l'aggiornamento dello stato si avvalgono della riflessione strutturale. Per quanto riguarda i canali, l'azione di checkpoint è ottenuta inserendo (durante la costruzione) nel checkpoint il contenuto dello stato e togliendolo (durante l'aggiornamento) dal checkpoint per farlo diventare lo stato del canale che effettua l'aggiornamento.
I metodi per la costruzione del checkpoint e per l'aggiornamento dello stato delle

repliche non introducono nulla di rilevante per l'integrazione delle riflessioni con la fault tolerance, pertanto non vengono ulteriormente dettagliati.

I metodi per la creazione delle repliche sono due create_channel_replica e create_replica. Il metodo create_channel_replica instaura un canale tra un sender ed un receiver che risiede sull'host del sender (determinato dalla primitiva PVM *pvm_tidtohost*). Siccome il canale non viene instaurato normalmente, come visto nel capitolo [3], si deve provvedere manualmente al suo inserimento nel dizionario residente sull'host del sender. Il dizionario in questione è recuperato dalla primitiva dictionary_of_host che effettua un'intersezione tra gli oggetti attivi residenti sull'host del sender e quelli che sono membri del gruppo di dizionari, DICTIONARY_GROUP. Il risultato di questa elaborazione è il tid del dizionario cercato perché esiste un solo dizionario per ogni host. Di seguito è riportato il codice del metodo create_channel_replica.

```
void repl_channel::create_channel_replica( int i ) {
  int s, r, c, h;
  s = pvm_gettid( sender_pool, i );
  r = pvm_gettid( receiver_pool, i );
  h = pvm_tidtohost( s );
    // crea l'i-sima replica del canale sull'host dell'i-sima replica del sender
  pvm_spawn( channel_filename, {(char *)s, (char *)r, t, (char *)i, NULL }, PvmTaskHost, h, 1, &c );
    // inserimento ''manuale'' del nuovo canale nel dizionario
  pvm_initsend( PvmDataRaw );
  pvm_pkint( &s, 1, 1 );
  pvm_pkint( &r, 1, 1 );
  pvm_pkstr( t );
  pvm_pkint( &c, 1, 1 );
  pvm_send( dictionary_of_host( h ), INSERT );
}
```

Il metodo create_replica si limita a creare la nuova replica ed ad inserirla nel corrispondente pool, pertanto non viene ulteriormente dettagliato.

5.5.5 Comportamento di un'Applicazione con Architettura a Replicazione dell'Interazione

La descrizione della tolleranza ai guasti gestita da canali che implementano la replicazione dell'interazione è fatta tramite un esempio.

Il primo passo consiste nell'esaminare la classe degli oggetti critici.
La modellazione che si intende implementare presenta la peculiarità di duplicare l'interazione tra due oggetti critici, quindi, per ogni oggetto critico vengono instaurati tanti canali quanti sono gli oggetti critici con cui interagisce.
La replicazione dell'oggetto critico è svolta al momento della prima interazione da uno dei canali che lo controllano (non ci sono problemi su fallimenti precedenti alla prima interazione in quanto è sufficiente ricreare l'oggetto con lo stato di partenza). Come visto nella sezione [2.5] grazie all'ereditarietà multipla è possibile costruire una

nuova classe con comportamento fault tolerant senza modificare quella di partenza. Ad esempio considerando la classe X riportata di seguito.

```
class X {
  public:
   // costruttore
    X( int init_value = 0 ) : x_state( init_value ) {};
   // distruttore
    ~X( void ) {};
   // metodi
    int set_x_state( int new_value ) {return x_state += new_value; };
  private:
    int x_state;
}
```

Si crea una classe Refl_X definita come

class Refl_X public: X, ReflObj {};

le cui istanze hanno la capacità di essere controllate da dei canali il cui genere è specificato al momento della computazione (**send**). Il comportamento fault tolerant viene, quindi, specificato al momento della richiesta della computazione richiedendo una meta-computazione di genere **repl_channel** (e quindi svolta da un canale istanza della classe **repl_channel**).

Per semplicità si considera un'applicazione composta da soli due oggetti attivi (critici) istanze della classe Refl_X, un client $\mathcal{A}_{Client}$ ed un server $\mathcal{A}_{Server}$. Entrambi gli oggetti sono critici ed hanno grado di criticità pari ad uno (si sceglie per semplicità che abbiano lo stesso grado di criticità, anche se si è visto che è possibile avere gradi di criticità differenti). $\mathcal{A}_{Client}$ e $\mathcal{A}_{Server}$ sono replicati rispettivamente da $\mathcal{A}'_{Client}$ e $\mathcal{A}'_{Server}$ e le due coppie ($\mathcal{A}_{Client}$, $\mathcal{A}_{Server}$ e $\mathcal{A}'_{Client}$, $\mathcal{A}'_{Server}$) sono controllate, rispettivamente, dai canali $\mathcal{RC}h_{Primario}$ e $\mathcal{RC}h_{Secondario}$ istanze della classe **repl_channel**.

Nella descrizione del comportamento dell'applicazione si tralascia la fase di creazione delle repliche e dei canali.

5.5.5.1 Interazione senza Fallimenti

Si supponga che $\mathcal{A}_{Client}$ richieda ad $\mathcal{A}_{Server}$ l'esecuzione del servizio **set_x_state** sul valore 7 e che il valore ottenuto dal servizio sia assegnato all'attributo **x_state** di $\mathcal{A}_{Client}$. Si supponga, inoltre, che inizialmente lo stato degli oggetti interagenti ($\mathcal{A}_{Client}$ e $\mathcal{A}_{Server}$) e delle loro repliche ($\mathcal{A}'_{Client}$ e $\mathcal{A}'_{Server}$) siano corretti (ad esempio $\mathcal{A}_{Client}$.x_state = $\mathcal{A}_{Server}$.x_state = $\mathcal{A}'_{Client}$.x_state = $\mathcal{A}'_{Server}$.x_state = 0).
Si esaminano i passi effettuati per elaborare il servizio richiesto nel caso in cui non si incorre nel fallimento di alcuna entità.

① $\mathcal{A}_{Client}$ inoltra ad $\mathcal{A}_{Server}$ la richiesta per il servizio set_x_state ($\mathcal{A}_{Client}$.x_state = $\mathcal{A}_{Server}$.x_state = $\mathcal{A}'_{Client}$.x_state = $\mathcal{A}'_{Server}$.x_state = 0).

② La richiesta per il servizio set_x_state viene intercettata dal canale $\mathcal{RC}h_{Primario}$ instaurato tra gli oggetti $\mathcal{A}_{Client}$ e $\mathcal{A}_{Server}$. $\mathcal{RC}h_{Primario}$ verifica se il canale precedente nel pool è fallito, quindi, informa anche le sue repliche della richiesta.

③ Il canale $\mathcal{RC}h_{Primario}$ richiede ad $\mathcal{A}_{Server}$ l'esecuzione del servizio set_x_state (apply). Lo stato di $\mathcal{A}_{Server}$ viene modificato ($\mathcal{A}_{Server}$.x_state = 7).

④ Il canale $\mathcal{RC}h_{Primario}$ costruisce il checkpoint di $\mathcal{A}_{Server}$ e lo invia agli altri canali del pool (nell'esempio solo $\mathcal{RC}h_{Secondario}$), quindi, attende la sincronizzazione con gli altri canali.

⑤ Il canale $\mathcal{RC}h_{Secondario}$ ricevuto il checkpoint di $\mathcal{A}_{Server}$, aggiorna lo stato della replica di $\mathcal{A}_{Server}$ che controlla $\mathcal{A}'_{Server}$. Quindi si sincronizza con gli altri canali. Lo stato di $\mathcal{A}'_{Server}$ viene modificato ($\mathcal{A}'_{Server}$.x_state = 7).

⑥ Il canale $\mathcal{RC}h_{Primario}$ invia il risultato del servizio set_x_state all'oggetto $\mathcal{A}_{Client}$. Lo stato di $\mathcal{A}_{Client}$ viene modificato ($\mathcal{A}_{Client}$.x_state = 7).

⑦ Il canale $\mathcal{RC}h_{Primario}$ costruisce il checkpoint di $\mathcal{A}_{Client}$ e lo invia agli altri canali del pool (nell'esempio solo $\mathcal{RC}h_{Secondario}$), quindi, si sincronizza con gli altri canali.

⑧ Il canale $\mathcal{RC}h_{Secondario}$ ricevuto il checkpoint di $\mathcal{A}_{Client}$, aggiorna lo stato della replica di $\mathcal{A}_{Client}$ che controlla $\mathcal{A}'_{Client}$. Quindi si sincronizza con gli altri canali. Lo stato di $\mathcal{A}'_{Client}$ viene modificato ($\mathcal{A}'_{Client}$.x_state = 7).

⑨ La meta-computazione ha termine.

Terminata la computazione il contenuto dell'attributo x_state degli oggetti $\mathcal{A}_{Client}$ e $\mathcal{A}_{Server}$ è pari a 7 e consistentemente a tale modifica anche l'attributo x_state delle loro repliche $\mathcal{A}'_{Client}$ e $\mathcal{A}'_{Server}$ è pari a 7.

5.5.5.2 Fallimento di una delle Repliche ($\mathcal{A}'_{Client}$, $\mathcal{A}'_{Server}$ o $\mathcal{RC}h_{Secondario}$)

Il caso del fallimento della replica del client, del server o del canale che li controlla vengono descritti assieme per le seguenti ragioni:

- Il canale viene eliminato sia quando fallisce il sender che quando fallisce il receiver, quindi, il caso del recupero del canale è compreso nel caso del recupero di uno dei due referenti.

- Il canale per scelta risiede sull'host del sender (il client) e data la natura dei fallimenti che si vuole gestire (fallimenti hardware), il fallimento del client coinvolge anche il canale.

- Il recupero della replica del receiver è del tutto analogo al recupero della replica del sender.

Si supponga che $\mathcal{A}_{Client}$ richieda ad $\mathcal{A}_{Server}$ l'esecuzione del servizio **set_x_state** sul valore 7 e che il valore ottenuto dal servizio sia assegnato all'attributo **x_state** di $\mathcal{A}_{Client}$. Si supponga, inoltre, che inizialmente lo stato degli oggetti interagenti ($\mathcal{A}_{Client}$ e $\mathcal{A}_{Server}$) e delle loro repliche ($\mathcal{A}'_{Client}$ e $\mathcal{A}'_{Server}$) siano corretti (ad esempio $\mathcal{A}_{Client}.\mathsf{x\_state} = \mathcal{A}_{Server}.\mathsf{x\_state} = \mathcal{A}'_{Client}.\mathsf{x\_state} = \mathcal{A}'_{Server}.\mathsf{x\_state} = 0$).
Si esaminano i passi effettuati per elaborare il servizio richiesto nel caso si verifichi il fallimento della replica del server $\mathcal{A}'_{Server}$. Data la caratteristica passiva della computazione svolta dalle repliche è indifferente il punto della computazione in cui $\mathcal{A}'_{Server}$ fallisce, nella descrizione si suppone che il fallimento sia occorso all'inizio dell'interazione.

① $\mathcal{A}_{Client}$ inoltra ad $\mathcal{A}_{Server}$ la richiesta per il servizio **set_x_state** ($\mathcal{A}_{Client}.\mathsf{x\_state} = \mathcal{A}_{Server}.\mathsf{x\_state} = \mathcal{A}'_{Client}.\mathsf{x\_state} = \mathcal{A}'_{Server}.\mathsf{x\_state} = 0$).

② La richiesta per il servizio **set_x_state** viene intercettata dal canale $\mathcal{RC}h_{Primario}$ instaurato tra gli oggetti $\mathcal{A}_{Client}$ e $\mathcal{A}_{Server}$. Il canale $\mathcal{RC}h_{Primario}$ rileva che il canale precedente nel pool (il canale $\mathcal{RC}h_{Secondario}$) è stato rimosso. Quindi procede al recupero (fault_recovery).

③ Il canale $\mathcal{RC}h_{Primario}$ determina che la rimozione del canale $\mathcal{RC}h_{Secondario}$ è dovuta al fallimento del suo referente $\mathcal{A}'_{Server}$. Quindi crea il sostituto ($\mathcal{A}''_{Server}$) dell'oggetto fallito.

④ Il canale $\mathcal{RC}h_{Primario}$ crea un nuovo canale $\mathcal{RC}h'_{Secondario}$ e lo aggancia ad $\mathcal{A}'_{Client}$ ed ad $\mathcal{A}''_{Server}$. A questo punto costruisce il checkpoint di $\mathcal{A}_{Server}$ e lo invia al nuovo canale $\mathcal{RC}h'_{Secondario}$ che provvede ad aggiornare, consistentemente con il checkpoint, lo stato della nuova replica. Analogamente per lo stato del canale.

⑤ Il canale $\mathcal{RC}h_{Primario}$ informa, quindi, tutti i canali del pool della richiesta intercettata.

⑥ Il canale $\mathcal{RC}h_{Primario}$ richiede ad $\mathcal{A}_{Server}$ l'esecuzione del servizio **set_x_state** (apply). Lo stato di $\mathcal{A}_{Server}$ viene modificato ($\mathcal{A}_{Server}.\mathsf{x\_state} = 7$).

⑦ Il canale $\mathcal{RC}h_{Primario}$ costruisce il checkpoint di $\mathcal{A}_{Server}$ e lo invia agli altri canali del pool (nell'esempio solo $\mathcal{RC}h'_{Secondario}$), quindi, attende la sincronizzazione con gli altri canali.

⑧ Il canale $\mathcal{RC}h'_{\mathsf{Secondario}}$ ricevuto il checkpoint di $\mathcal{A}_{\mathsf{Server}}$ aggiorna lo stato della replica di $\mathcal{A}_{\mathsf{Server}}$ che controlla $\mathcal{A}''_{\mathsf{Server}}$. Quindi si sincronizza con gli altri canali. Lo stato di $\mathcal{A}''_{\mathsf{Server}}$ viene modificato ($\mathcal{A}''_{\mathsf{Server}}$.x_state = 7).

⑨ Il canale $\mathcal{RC}h_{\mathsf{Primario}}$ invia il risultato del servizio **set_x_state** all'oggetto $\mathcal{A}_{\mathsf{Client}}$. Lo stato di $\mathcal{A}_{\mathsf{Client}}$ viene modificato ($\mathcal{A}_{\mathsf{Client}}$.x_state = 7).

⑩ Il canale $\mathcal{RC}h_{\mathsf{Primario}}$ costruisce il checkpoint di $\mathcal{A}_{\mathsf{Client}}$ e lo invia agli altri canali del pool (nell'esempio solo $\mathcal{RC}h'_{\mathsf{Secondario}}$), quindi, si sincronizza con gli altri canali.

⑫ Il canale $\mathcal{RC}h'_{\mathsf{Secondario}}$ ricevuto il checkpoint di $\mathcal{A}_{\mathsf{Client}}$ aggiorna lo stato della replica di $\mathcal{A}_{\mathsf{Client}}$ che controlla $\mathcal{A}'_{\mathsf{Client}}$. Quindi si sincronizza con gli altri canali. Lo stato di $\mathcal{A}'_{\mathsf{Client}}$ viene modificato ($\mathcal{A}'_{\mathsf{Client}}$.x_state = 7).

⑬ La meta-computazione ha termine.

Nonostante sia occorso il fallimento di una replica la computazione giunge a termine con successo, è stata creata una nuova replica $\mathcal{A}''_{\mathsf{Server}}$ ed un nuovo canale $\mathcal{RC}h'_{\mathsf{Secondario}}$ con il compito di gestirla e lo stato di tutte le entità coinvolte ($\mathcal{A}_{\mathsf{Client}}$ e $\mathcal{A}_{\mathsf{Server}}$) è stato modificato nel modo corretto e di conseguenza è stato modificato anche lo stato delle repliche ($\mathcal{A}'_{\mathsf{Client}}$ e $\mathcal{A}''_{\mathsf{Server}}$).

5.5.5.3 Fallimento delle entità primarie ($\mathcal{A}_{\mathsf{Client}}$, $\mathcal{A}_{\mathsf{Server}}$ o $\mathcal{RC}h_{\mathsf{Primario}}$)

Le osservazioni fatte sul caso del fallimento delle repliche sono applicabili anche al caso del recupero del fallimento delle copie primarie. L'attenzione va posta, invece, sul fatto che le entità primarie ricoprono un ruolo attivo nella computazione, come tale è essenziale che la fase di recupero permetta di riprendere la computazione in modo che le azioni siano consistenti con lo stato iniziale delle entità primarie recuperate.
I punti salienti, in cui può occorrere il fallimento, sono sostanzialmente riconducibili a tre regioni temporali in cui le azioni di recupero sono analoghe:

- Prima che il cliente richieda un servizio al server e dopo che i risultati del servizio richiesto sono stati inviati al client.

- Dopo che il client ha richiesto un servizio al server e prima che il server abbia soddisfatto tale richiesta.

- Dopo che il server ha soddisfatto il servizio richiestogli e prima che i risultati del servizio richiesto siano stati inviati al client.

Ognuna di queste regioni ha una frontiera definita dalle azioni di checkpoint, cioè "prima" deve essere interpretato come "prima che venga effettuata la relativa azione di checkpoint" e "dopo" deve essere interpretato come "dopo che è stata effettuata la relativa azione di checkpoint".

L'ultimo canale del pool di canali ha il compito di sovraintendere il rilevamento e quindi il recupero del fallimento del canale primario. I checkpoint sono propagati dal canale primario alle repliche in un'unica soluzione. Alla ricezione di un checkpoint i canali secondari provvedono ad aggiornare un contatore (progress_counter) che scandisce l'evolvere della computazione in base alle tre regioni temporali descritte. Il contatore permette il recupero della computazione da un punto sicuro in cui è possibile ripetere l'elaborazione senza avere problemi.

Il recupero è effettuato rilevando quale referente è fallito, quindi, si crea una replica per sostituirlo aggiornandone lo stato consistentemente con lo stato delle proprie repliche. Il canale che effettua il recupero, valuta se effettuare o meno il rollback del referente non fallito a seconda dell'elaborazione che sta effettuando. Se il referente non fallito stava interagendo con l'oggetto fallito o non stava effettuando alcuna computazione allora viene effettuato il rollback, se invece era intento ad interagire con un altro oggetto non fallito non è necessario effettuare il rollback. Infine, viene instaurato un nuovo canale, tra la nuova replica ed il referente non fallito, che riprende la computazione dell'azione interrotta dal fallimento.

Svolgere il recupero in modo che lo stato dei referenti ozianti o che interagivano con l'oggetto fallito sia quello che avevano quando è stata effettuata l'ultima azione di checkpoint consente di non perdere alcuna richiesta (le richieste, inoltrate da un client non fallito, antecedenti all'ultima azione di checkpoint sono note al canale che le continua a svolgere, quelle successive all'ultima azione di checkpoint sono automaticamente ripetute in seguito al rollback del client).

5.5.6 Vantaggi Ottenuti dall'Uso del Modello a Materializzazione del Canale

I vantaggi ottenuti dall'impiego della riflessione nel caso del modello a meta-oggetto sono ottenuti anche nel caso del modello a materializzazione del canale.

La <u>trasparenza</u> è una proprietà tipica delle riflessioni e quindi presente anche nel modello a materializzazione del canale. I canali possono controllare qualsiasi oggetto critico ed il loro operato è trasparente alla computazione svolta dagli oggetti che monitorizzano.

I canali sono degli oggetti, quindi, grazie all'ereditarietà è possibile ottenere comportamenti fault tolerant differenti ridefinendo alcuni metodi. Il meccanismo risulta, quindi, essere molto flessibile.

La computazione può essere distinta in due aspetti: funzionale all'applicazione e non funzionale all'applicazione. Il primo aspetto è svolto dagli oggetti del livello base mentre il secondo è svolto dai canali. Quindi si ha la separazione dei due aspetti.

A questi vantaggi si aggiunge la possibilità di diversificare la computazione riflessiva. È possibile, quindi, definire un comportamento fault tolerant che varia in base alla criticità dell'interazione che monitorizza. Questa possibilità è stata esaminata in [Ancona-Cazzola-Dodero-Gianuzzi 95].

Il particolare ciclo di vita del modello a materializzazione del canale permette di rilevare un fallimento verificando esclusivamente se uno dei canali è stato rimosso, riducendo, quindi, il numero dei controlli.

Il modello a materializzazione del canale presenta lo svantaggio di avere più meta-entità (canali) che gestiscono un unico oggetto critico, pertanto è necessario coordinare|sincronizzare questi canali per evitare che gli oggetti falliti siano recuperati più volte (con il rischio di eliminare risultati parziali ancora in uso).

5.6 Canali Versus Meta-Oggetti nella Realizzazione della Replicazione

Il confronto tra i due approcci può essere effettuato su due livelli:

- Il rispetto dei requisiti imposti.
- Il numero di meta-entità necessarie per rispettare i requisiti imposti.

I requisiti che il sistema fault tolerant deve rispettare affinché si possa garantire il corretto operato dei suoi oggetti critici possono essere riassunti nei seguenti punti:

- Un oggetto critico deve vedere soddisfatta (ed in modo corretto) ogni sua richiesta inoltrata ad ogni altro oggetto.
- Un oggetto critico deve soddisfare (in modo corretto) ogni servizio che gli viene richiesto.

L'approccio basato sul modello a meta-oggetto prevede un meta-oggetto separato per ogni entità critica del sistema e per ogni replica degli oggetti critici. L'approccio garantisce il recupero delle computazioni, la cui interruzione è dovuta al fallimento del server che le stava calcolando. Però non viene garantita la computazione di

servizi la cui richiesta è inoltrata ad un server fallito (pervenuti durante il recupero di quest'ultimo). Il problema persiste anche se l'entità che richiede il servizio dispone di un meta-oggetto, in quanto è peculiarità del modello a meta-oggetto che i meta-oggetti siano consci solo ed esclusivamente dei propri referenti e quindi non possono rilevare il fallimento di un oggetto differente dal proprio referente (previo il considerare la possibilità di controllare tutti gli oggetti con cui un oggetto può collaborare, azione complessa e di difficile realizzazione perché a priori non si è in grado di determinare gli oggetti con cui un oggetto può comunicare).
L'approccio a replicazione dell'interazione invece prevede un canale associato ad ogni richiesta, in questo modo si riesce a monitorare sia il server che il client della computazione garantendo, quindi, il soddisfacimento anche dei servizi richiesti dal client durante il recupero del server.

L'approccio a meta-oggetti prevede un meta-oggetto per ogni oggetto critico ed uno per ogni replica degli oggetti critici. L'approccio a replicazione dell'interazione invece prevede un canale per ogni interazione tra client e server ed un canale tra ogni replica del client e del server che interagiscono.
Quindi considerando l'interazione di due oggetti critici con grado di criticità pari a uno, per garantirne il corretto espletamento, l'approccio a meta-oggetto necessita di un meta-oggetto per ogni oggetto critico ed un meta-oggetto per ogni replica per un totale di quattro meta-oggetti, mentre con l'approccio a replicazione dell'interazione si ha un canale instaurato tra i due oggetti critici ed uno tra le repliche dei due oggetti critici, per un totale di due canali.

Quindi l'approccio a replicazione dell'interazione, a differenza dell'approccio a meta-oggetto, presenta il vantaggio di soddisfare entrambi i requisiti richiesti e di utilizzare un numero minore di meta-entità al costo di una complicazione delle operazioni di recupero, in quanto per ogni oggetto fallito ci sono più canali instaurati che provvedono al suo recupero; sono quindi necessarie delle politiche di sincronizzazione tra canali per attuare il recupero.

5.7 Conclusioni

Il problema riscontrato nella modellazione della replicazione utilizzando il modello a meta-oggetto è la perdita delle richieste inoltrate ad un oggetto critico quando questo è fallito. Il problema è superabile introducendo per ogni oggetto (critico e non) un meta-oggetto, che ha il compito di sincronizzare l'invio di una richiesta ad un oggetto critico con il recupero di un eventuale fallimento. In questo modo un meta-oggetto deve poter monitorare tutti gli oggetti con cui il proprio referente deve comunicare.

5.7. Conclusioni

Nella modellazione della replicazione basata sulla replicazione dell'interazione ogni canale è legato ad un client ed ad un server, le richieste inoltrate dal client sono intercettate dal canale che le smista alle proprie repliche ed al server che le soddisfa. In caso di fallimento i canali replica provvedono al recupero. Al momento del fallimento (e del conseguente recupero), gli oggetti attivi che hanno la possibilità di interagire con l'oggetto fallito possono essere ozianti, legati ad una computazione con l'oggetto fallito o legati con una computazione ad uno degli altri oggetti del sistema. Gli unici oggetti che possono inoltrare una richiesta all'oggetto fallito sono gli oggetti ozianti. La loro richiesta comunque non viene persa perché i canali al momento del recupero provvedono a riavvolgere gli oggetti ozianti e quindi le richieste inoltrate durante il recupero vengono ripetute.

Associando le meta-entità alle computazioni (come nella modellazione a replicazione dell'interazione) piuttosto che agli oggetti interagenti (come nella modellazione basata sul modello a meta-oggetto) si ha il vantaggio di ridurre le meta-entità e la coordinazione delle stesse.

La replicazione dell'interazione, quindi, garantisce che <u>tutti</u> i servizi richiesti vengano soddisfatti correttamente.

Conclusioni

In questa tesi si sono analizzati i pregi ed i difetti dei diversi modelli di riflessione esistenti e l'applicazione della riflessione nella costruzione di applicazioni tolleranti ai guasti.

Il lavoro svolto può essere riassunto nei seguenti tre risultati.

Modello a Materializzazione del Canale

La materializzazione del canale è un nuovo modello di riflessione derivato dal modello materializzazione del messaggio. Presenta la caratteristica di preservare i vantaggi del modello da cui è derivato (granularità fine) e di correggerne le limitazioni (ciclo di vita del messaggio, mancanza di continuità e mancanza di referenti manipolabili). Concettualmente il canale viene instaurato tra due oggetti (i suoi referenti) che si scambiano dei messaggi (cioè tra due oggetti legati da una relazione di clientship) ed effettua una meta-computazione sulla computazione richiesta da un oggetto client (o sender) ad un oggetto server (o receiver) e come effetto collaterale ha la possibilità di controllare|modificare i <u>due</u> referenti oltre che la computazione stessa. I canali instaurati tra due oggetti possono essere diversi e sono distinti in base al *genere* che definisce il comportamento riflessivo che hanno le computazioni supervisionate da quel canale.
I canali grazie alla granularità, al ciclo di vita del canale ed al genere del canale possono emulare ogni altro modello di riflessione tra quelli presentati (modello a meta-classe, modello a meta-oggetto e modello a materializzazione del messaggio) sottolineando la duttilità del nuovo modello.

send-HandleMessage-apply

L'analisi dei diversi modelli di riflessioni esistenti (compreso il modello a materializzazione del canale), all'interno di diversi linguaggi di programmazione orientati agli oggetti (SmallTalk, C++ e Oberon), sia in ambiente sequenziale che concorrente, ha condotto alla definizione di un meccanismo, composto da tre metodi (send, HandleMessage ed apply), che permette agevolmente di introdurre delle computazioni riflessive all'interno delle proprie applicazioni.

Il meccanismo separa le operazioni di cambiamento di livello, *shift-up* e *shift-down* (send e apply) della meta-computazione vera e propria. Le operazioni di shift-up e shift-down permettono di attivare la meta-computazione (send) e di effettuare la computazione "normale" (apply) ed hanno la caratteristica di essere sempre uguali ed indipendenti dalla meta-computazione richiesta. Quindi i metodi send ed apply possono essere inseriti in una classe da cui ereditano tutti gli oggetti riflessivi ed essere, quindi, scritti una sola volta per ogni modello e non essere più toccati (c'è quindi la possibilità di inserire questa classe in una libreria e riutilizzarla successivamente).
Il metodo HandleMessage è proprio della meta-entità (è un metodo di tutte le meta-entità) e definisce la meta-computazione, quindi, deve essere ridefinito ogniqualvolta sia necessario definire un comportamento riflessivo diverso per una nuova meta-entità. Con questo meccanismo le modifiche da effettuare per ottenere un nuovo comportamento riflessivo sono limitate alla definizione di una nuova classe che ridefinisce un unico metodo (HandleMessage).
Il meccanismo presentato è indipendente dal linguaggio adottato e dall'ambiente di sviluppo.

Replicazione dell'Interazione

Applicando le riflessioni nella costruzione di software fault tolerant si ottengono i seguenti vantaggi:

- Separazione dell'aspetto funzionale dell'applicazione, svolto dalle entitè del livello base, dall'aspetto fault tolerant, svolto dalle meta-entità.
- Trasparenza nell'introduzione e nell'utilizzo delle politiche di fault tolerance nei sistemi. Grazie alla trasparenza, rendere fault tolerant un sistema già esistente è molto agevole e non richiede molte modifiche al sistema originale.
- Possibilità da parte del progettatore di definire le proprie politiche di fault tolerance adattando per i propri scopi politiche predefinite.

Riallacciandosi al lavoro svolto in [Fabre-Nicomette-Pérennou-Stroud-Wu 95] che adotta il modello a meta-oggetto, si è definita una modellazione della replicazione chiamata replicazione dell'interazione, che sfrutta i canali per duplicare l'intera interazione. In questo modo si riesce a dotare un'applicazione, in modo trasparente, di comportamento fault tolerant e, rispetto al lavoro di Fabre et al., il modello a replicazione dell'interazione garantisce il corretto svolgimento di tutti i servizi richiesti ad un oggetto critico. Inoltre il modello a replicazione dell'interazione permette di discriminare facilmente e trasparentemente, grazie al genere dei canali, il tipo di tolleranza ai guasti di una computazione in base alla criticità delle operazioni.

Il lavoro svolto può essere considerato solo il punto di inizio di molti approfondimenti ed idee nuove relativamente all'integrazione delle riflessioni con le tecniche di fault tolerance. Di seguito vengono elencate le idee principali che monopolizzeranno l'attenzione delle analisi successive sull'argomento.

- Nel corso di questo lavoro si è considerata esclusivamente la possibilità di instaurare un canale solo tra due referenti (canale 1:1), in futuro si studierà la possibilità di costruire canali 1:n, m:1 e m:n (multicanali) cioè istaurati tra più di due referenti, esaminandone il ruolo e le possibili applicazioni.

- Analisi delle possibilità di integrazione delle architetture riflessive fault tolerant per CORBA. CORBA è considerato il futuro dei sistemi concorrenti orientati ad oggetti, pertanto la naturale evoluzione del nostro lavoro è rappresentata dalla migrazione in CORBA, delle idee introdotte in questa tesi.

- Analisi dell'integrazione delle riflessioni con tecniche di software fault tolerance (ad esempio la programmazione a n-versioni). L'idea si basa sullo studio delle interfacce degli oggetti versionati in modo da unificarle in un'unica segnatura [Baumgartner-Russo 93]. Tramite la segnatura unificata le meta-entità possono attivare i metodi "equivalenti" delle diverse versioni che controllano.

- Analisi dell'integrazioni delle riflessioni a tecniche di *forward error recovery*. Le tecniche di forward error recovery sono politiche di fault tolerance che, invece di attendere che occorra un fallimento, tentano di prevenirlo esaminando le successive azioni prima che siano effettuate.

Versioni preliminari di questo lavoro sono state presentate a OOPSLA'95 (poster section) ed alla conferenza annuale di TABOO.

Appendici

A SmallTalk

SmallTalk è il primo linguaggio ad oggetti, dopo SIMULA, degno di questo nome. Implementa i concetti di classe e di ereditarietà ed introduce per la prima volta il concetto di scambio di messaggi, che è diventato uno standard di fatto nel mondo della programmazione orientata agli oggetti. Tre principi fondamentali riassumono la filosofia del linguaggio:

- Tutte le entità manipolate sono degli oggetti.
- Ogni oggetto è una istanza di una classe che definisce la sua struttura ed il suo comportamento. In particolare una classe è l'istanza di un'altra classe, chiamata la metaclasse corrispondente.
- Un oggetto è accessibile soltanto attraverso l'invio di messaggi.

A differenza dei linguaggi ad oggetti concepiti come estensione di altri linguaggi, come il LISP (ObjVLisp e CLOS) o il C (C++), l'universo di SmallTalk è totalmente omogeneo: è composto esclusivamente di oggetti e istanze di classi che comunicano tra loro tramite l'invio di messaggi.
Il sistema SmallTalk, è ad un tempo un sistema di sviluppo, un ambiente di programmazione ed un linguaggio tradizionale. La ricchezza della sua libreria di classi predefinita ne fa uno strumento ideale per lo sviluppo rapido di applicazioni. Nel seguito si esamina la sintassi e la semantica del linguaggio.

Il Linguaggio SmallTalk

SmallTalk si distingue dalla maggioranza degli altri linguaggi ad oggetti, per il fatto che propone un modello uniforme di programmazione. Tutte le entità del sistema sono degli oggetti, che siano numeri, stringhe di caratteri, entità più complesse come i processi o nuovi oggetti definiti per una specifica applicazione. Tutte le operazioni sono realizzate tramite un'unica struttura di controllo: l'invio dei messaggi.

Sintassi e Semantica

Esistono quattro tipi di espressioni: le costanti, le variabili, gli invii di messaggi ed i blocchi. Ciascuna espressione definisce un oggetto chiamato valore dell'espressione.

Costanti

Le costanti sono ripartite in cinque categorie:

- I numeri, scritti nel modo classico come nella maggior parte dei linguaggi (es. 3, 8r377 (ottale), e 2r11e6 (binario)).
- I caratteri, scritti preceduti dal segno $, (es. $a, $@, $$).
- Le stringhe di caratteri, scritte racchiuse tra apici semplici (es. 'SmallTalk').
- I simboli sono stringhe di caratteri alfanumerici che servono a dare un nome agli oggetti del sistema. Si scrivono facendo precedere la stringa dal carattere # (es. #Object).
- I vettori, oggetti strutturati ogni elemento è accessibile tramite un indice numerico, il cui primo valore è uno. un vettore si scrive facendo seguire un '#' dai suoi elementi raggruppati tra parentesi. (es. #(1 'uno' $k)).

Variabili

Una variabile permette di indicare un oggetto all'interno di un'espressione. Non è tipata e può prendere successivamente per valore degli oggetti che sono istanze di classi diverse. L'assegnamento, indicato con il simbolo $\leftarrow$[1], è la sola operazione che non è realizzata tramite l'invio di un messaggio. Le variabili sono caratterizzate dalla loro durata di vita e dalla loro visibilità. Possono essere private, proprietà esclusiva di un oggetto, o pubbliche, accessibili da più oggetti. Ne esistono quattro tipi:

- Le variabili di istanza che rappresentano lo stato corrente di un oggetto. La loro durata di vita è la stessa dell'oggetto a cui si riferiscono e sono private: soltanto l'oggetto stesso può accedervi direttamente, senza passare per un invio di messaggi.
- Le variabili temporanee sono le variabili locali ai metodi. La loro durata di vita non è superiore al tempo di attivazione del metodo.

[1] Si scrive := in SmallTalk-V

- Le variabili di classe sono variabili condivise tra l'insieme delle istanze di una classe e delle sottoclassi relative. Sono definite all'atto della creazione e della modifica della classe.

- Le variabili globali sono le variabili condivise dall'insieme degli oggetti del sistema. Sono elencate in un oggetto particolare, che si chiama SmallTalk e che è organizzato come un dizionario.

Esistono inoltre le pseudo-variabili che hanno la proprietà di non poter essere modificate tramite l'assegnazione (es. true, false, nil, self e super).

Invio dei Messaggi

Un messaggio si compone di un ricevente, un selettore ed eventuali argomenti. Quindi nell'espressione:

#(1 2 3) at: 2 put: $i

Il vettore #(1 2 3) è il ricevente del messaggio, at:put: è il selettore del messaggio, mentre 2 e $i sono gli argomenti. Esistono tre categorie di messaggi:

- I messaggi unari che invocano dei metodi senza argomenti, sono chiamati così perché solo il ricevente del messaggio viene effettivamente menzionato (es. 3 IsInteger).

- I messaggi con parole chiave o keyword message che rappresentano la forma più generale dei messaggi. I metodi invocati hanno almeno un argomento che è introdotto da una parola chiave che è seguita dal carattere ':' (es. #(1 2 3) at: 2). Il selettore è formato dalla concatenazione delle parole chiave.

- I messaggi binari permettono di usare degli operatori binari. Contengono un solo argomento ed il loro selettore deve essere composto da uno o due caratteri alfanumerici (es. 3 <= 4, 3 + 4).

Blocchi

I blocchi, che sono chiusi tra parentesi quadre, sono delle sequenze di espressioni la cui valutazione viene differita. Sono degli oggetti a pieno titolo il cui comportamento è precisato dalla classe BlockContext e possono avere dei parametri, introdotti nella dichiarazione dal carattere ':' e separati dal corpo del blocco dal carattere '|' (es. inf ← [:a :b | a<b]). Il blocco viene valutato alla ricezione di un messaggio che riguarda i parametri in esso elencati. I parametri devono essere preceduti dalla parola chiave value: (es. inf value: 3 value: 4). I blocchi sono usati per definire delle strutture di controllo simili ai cicli o alle espressioni condizionali (es. anObject ifTrue: trueBlock ifFalse: falseBlock).

Metodi

Un metodo SmallTalk è analogo ad una procedura di un linguaggio come il Pascal. Da un punto di vista sintattico è composto da tre parti:

- Intestazione: definisce il selettore ed i parametri del metodo. L'oggetto a cui si applica il metodo non viene menzionato; è noto implicitamente e può essere riferito all'interno del metodo stesso, tramite le pseudo-variabili self e super.

- Dichiarazione delle variabili locali: definisce delle variabili temporanee usate dal metodo per contenere dei valori intermedi.

- Corpo del metodo: composto da una sequenza di espressioni separate da punti.

Un invio di messaggi corrisponde alla chiamata di un metodo che ha lo stesso nome del selettore del messaggio, la ricerca del metodo è effettuata a partire dalla classe dell'oggetto ricevente, seguendo le linee di ereditarietà tra le classi (vedere [1.5.1]). Tutti i metodi restituiscono un risultato che per difetto è l'oggetto ricevente. Nel caso un metodo debba restituire un risultato particolare, questo è introdotto dal simbolo "↑".
I metodi, inoltre, possono essere distinti in:

- Metodi d'istanza attivabili dall'oggetto.

- Metodi di classe metodi attivabili dalla classe dell'oggetto. Questi metodi normalmente sono usati per creare le istanze di quella classe o per inizializzare le variabili di classe (ricordo che le classi sono oggetti e come tali possono avere dei metodi).

La Libreria delle Classi

La caratteristica migliore offerta da SmallTalk è la libreria delle classi predefinita.
La dimensione della libreria disponibile non può mancare di sorprendere: l'immagine virtuale specificata in [Goldberg-Robson 83] conta non meno di 500 classi e più di 4000 metodi! Queste classi possono essere raggruppate in tre categorie:

- Le classi dedicate al funzionamento del sistema, come Class, che definisce il comportamento della classe, Object che definisce il comportamento per default degli oggetti, Number, Boolean, BlockContext e così via.

- Le classi che definiscono l'ambiente di programmazione come Model, View, Controller e così via, che servono a creare, fra gli altri, i *browser*, gli *editor* di testi e grafici.

- Le classi che costituiscono la libreria iniziale del sistema, come Collection, Stream, Dictionary e così via, che definiscono la maggior parte dei tipi di dato utilizzati solitamente nella programmazione, oltre alle operazioni che sono loro proprie.

La ricchezza di classi predefinite rende SmallTalk un ottimo strumento per la prototipizzazione rapida di applicazioni.

L'Implementazione del Sistema SmallTalk

La maggior parte del sistema SmallTalk è scritta nel linguaggio SmallTalk stesso. Questa parte viene chiamata l'immagine virtuale e comprende più di 15000 oggetti predefiniti con cui sono costruiti tra le altre cose un compilatore, un debugger, degli editor ed il sistema di gestione dei file. La parte restante che viene chiamata macchina virtuale, è definita in termini di una macchina astratta che possiede un proprio ciclo di istruzioni. Il compilatore traduce il testo sorgente dei metodi in istruzioni di questa macchina astratta e non in istruzioni della macchina fisica che supporta il sistema SmallTalk. In questo modo, soltanto la macchina virtuale deve essere ridefinita quando il sistema deve essere installato su una macchina fisica particolare. L'implementazione della macchina virtuale si basa su un sistema di gestione degli oggetti che gestisce la memoria, su un interprete che esegue il codice prodotto dal compilatore e su delle primitive che sono le procedure scritte direttamente nel linguaggio della macchina ospite.

Il sistema di gestione degli oggetti

Tutte le operazioni di consultazione e manipolazione degli oggetti sono realizzate dal sistema di gestione degli oggetti: accesso alle variabili degli oggetti, accesso alle classi degli oggetti (per ritrovare i metodi e attivarli in seguito alla ricezione di un messaggio), creazione di nuovi oggetti, gestione dello spazio liberato da oggetti non più utilizzati.
Ciascuna entità del sistema SmallTalk, compresi i metodi, è rappresentato da un oggetto identificato da una chiave unica, il *puntatore all'oggetto*. Quest'ultimo permette un accesso indiretto all'oggetto attraverso una tavola che raggruppa gli indirizzi di memoria delle strutture fisiche dell'insieme degli oggetti e le chiavi corrispondenti. Nelle espressioni del linguaggio, un oggetto è definito attraverso la sua chiave. Questa chiave non cambia più per tutta la durata della vita dell'oggetto e permette di fare riferimento ad esso indipendentemente dalla sua localizzazione in memoria. Così, spostare un oggetto in memoria non necessita che di una semplice modifica tra i suoi indirizzi nella tavola degli oggetti. Ciascun oggetto è rappresentato da un blocco di memoria che contiene: la dimensione del blocco, la chiave della classe d'appartenenza dell'oggetto, che permette di accedere al suo comportamento e i

valori dei campi dell'oggetto, che descrivono il suo stato in quel momento. Questi valori sono a loro volta delle chiavi o meglio, per i numeri e le stringhe di caratteri sono più precisamente dei valori numerici direttamente interpretati senza passare per la tavola degli oggetti. A differenza dei linguaggi come il C++, in cui la gestione della memoria è compito del programmatore, non è necessaria alcuna operazione esplicita per l'allocazione o per la liberazione di spazio in memoria. Queste operazioni sono interamente gestite dal sistema di gestione degli oggetti che, per ciascun oggetto, gestisce un *contatore di riferimenti* che totalizza il numero di riferimenti all'oggetto. Quando questo contatore raggiunge lo zero, l'oggetto non è più utilizzato e lo spazio che occupa in memoria può essere riallocato. Il vantaggio principale di questa tecnica è la ripartizione dei tempi di gestione della memoria sull'insieme delle operazioni che la modificano. Ciò necessita però l'esistenza di una procedura di garbage-collection, per recuperare lo spazio occupato dalle strutture circolari.

L'Interprete

Il compilatore traduce il corpo dei metodi in istruzioni della macchina virtuale, chiamate byte-code. Quando un metodo viene attivato, i byte-code corrispondenti vengono eseguiti da un interprete, scritto direttamente per la macchina ospite. La macchina virtuale è una macchina a stack e l'esecuzione delle sue istruzioni è di tipo classico, a parte ciò che riguarda lo scambio dei messaggi. La maggior parte delle istruzioni riguarda l'aggiunta e l'estrazione di elementi dalla pila e operazioni sugli elementi che si trovano sulla cima della pila.

Le Variabili

Le variabili utilizzate in un metodo possono essere ripartite in tre diverse categorie:

- Le variabili temporanee, che sono locali al metodo, sono situate in una zona speciale, creata all'attivazione del metodo. Questa zona contiene anche i parametri del metodo stesso.
- Le variabili d'istanza che sono locali all'oggetto ricevente.
- Le variabili globali, di cui fanno parte le variabili di classe, che si trovano nei dizionari condivisi dalla classe del ricevente. i loro valori si trovano in un campo del metodo, chiamato literal frame, dove sono collegate anche le costanti, che sono usate localmente all'interno del metodo.

Il compilatore traduce un riferimento ad una variabile con un indirizzamento di tipo indiciato all'interno della zona in cui la variabile è memorizzata. L'accesso a ciascuna zona è realizzato da un byte-code particolare in cui gli ultimi bit esprimono il valore dello spiazzamento rispetto alla locazione iniziale della zona di memoria.

Lo scambio dei messaggi

Una volta compilato, uno scambio di messaggi è rappresentato da un byte-code che fa riferimento alla posizione in cui è stato posto il selettore del messaggio interessato all'interno del literal frame del metodo. La ricerca del selettore comincia all'interno del dizionario dei metodi della classe del ricevente e prosegue all'interno di quelle delle super classi successive, fino alla cima. Se il selettore non viene trovato in nessuno dei dizionari, viene inviato al ricevente un messaggio doesNotUnderstand:, che specifica che il messaggio non è stato compreso. Per ragioni di ottimizzazione delle prestazioni, alcune operazioni vengono tradotte in una sequenza di byte-code che non comporta alcuno scambio di messaggi.

I metodi

La macchina virtuale è in grado di eseguire solo metodi. Quando ad esempio, viene sottoposta al sistema un'espressione aritmetica il sistema crea un metodo temporaneo della classe UndefinedObject e lo distrugge subito dopo averlo eseguito. Un metodo è un oggetto come un altro, un'istanza della classe CompiledMethod, in cui i campi rappresentano la literal frame e i byte-code corrispondenti.
Per eseguire un metodo, l'interprete crea una istanza di quella classe e riempie i suoi campi con (in termini di puntatori ad oggetti) il ricevente, la pila di esecuzione creata per l'occasione, il metodo stesso, il prossimo byte-code che va eseguito, la zona temporanea creata appositamente ed il contesto di chiamata. Al termine del metodo, il risultato viene piazzato in cima alla pila del chiamante, in modo da essere accessibile da quest'ultimo.

Le primitive

Le primitive, che sono circa un centinaio, sono procedure scritte nel linguaggio della macchina ospite per interrompere la regressione infinita dello scambio dei messaggi e per ottimizzare le prestazioni del sistema. Sono rappresentate nello stesso modo dei metodi ordinari, con un flag per avvertire l'interprete che contengono un codice, che può essere eseguito direttamente dalla macchina ospite e non dei byte-code.
Le primitive realizzano in particolare l'input e l'output, l'aritmetica intera, certe operazioni grafiche e l'allocazione di oggetti in memoria. Alcune di queste operazioni sono difficili o anche impossibili da scrivere il SmallTalk. Altre sono chiamate così spesso che la loro definizione in SmallTalk penalizzerebbe le prestazioni complessive del sistema.

B Oberon e Affini

Oberon è un linguaggio di programmazione general-purpouse discendente da Modula-2 [Wirth 85]. La sua principale caratteristica è il concetto di estensione di tipo, una forma forte di ereditarietà singola, che permette la costruzione di nuovi tipi di dato a partire da quelli già esistenti.

Rispetto al suo predecessore, Modula-2, Oberon è un linguaggio significativamente più semplice, pur presentando un potere espressivo indubbiamente superiore.
Il meccanismo per l'estendibilità dei tipi ed il supporto del run-time type information sono alcuni dei pochi concetti introdotti nel passaggio da Modula-2 ad Oberon; viceversa molte sono le caratteristiche eliminate da Modula-2 in tale passaggio, come ad esempio i tipi opachi, il costrutto *for*, i tipi sotto-rango ed elencazione.

Procedure e Procedure di Ordine Superiore

Esistono due tipi di procedure, dette procedure vere e proprie e funzioni e si distinguono nell'intestazione per l'assenza o meno del tipo del valore di ritorno e nel corpo per la presenza o meno dell'istruzione RETURN. Le procedure possono essere ricorsive, esportabili (se marcate con *) o definibili successivamente (se marcate con ↑).
I parametri possono essere passati in due modi, per *variabile* o per *valore* ed il modo con cui sono passati viene indicato nella lista dei parametri formali dalla presenza o meno della parola chiave VAR.
I parametri formali possono essere di qualsiasi tipo lecito in Oberon compresi gli array, nel qual caso si dichiarano senza specificare la dimensione e sono detti array aperti. Il tipo di ritorno di una funzione non può essere né un record né un array.
Esistono inoltre delle procedure predefinite come ABS, MIN, CHR, INC.
Oberon, riprende da Modula-2, la possibilità di definire dei tipi procedura e quindi delle variabili di tipo procedura i cui valori sono delle procedure.
Affinché una procedura P possa essere assegnata ad una variabile di tipo procedura TP, occorre che il tipo dei parametri formali di P sia lo stesso di quello dei parametri formali di TP. Questo comprende anche il tipo del risultato.

Naturalmente le variabili di tipo procedura possono essere argomenti di altre procedure, che, in tal caso, sono dette di ordine superiore.

Estendibilità dei Record

La principale caratteristica di Oberon risulta essere il meccanismo del record type extension, che permette la definizione di nuovi tipi sulla base di tipi già esistenti, e stabilisce una relazione di sotto-tipazione tra i nuovi ed i vecchi tipi record.[1]

def. Dati due tipi record T e T′ legati da una dichiarazione della forma

$$T' = \text{RECORD(T) ... END;}$$

T′ è un'estensione di T e T è il tipo base di T′.

La definizione precedente esprime il fatto che T′ ha la struttura di T (cioè tutti i campi definiti in T) più i campi definiti direttamente in T′.
Esempio

```
TYPE
  P1 = PROCEDURE( p : Point; cx, dy : INTEGER );

  PointDesc =
    RECORD
      x, y : INTEGER;
      move : P1;
    END;

  Point = POINTER TO PointDesc;

  PROCEDURE Move_Point(p:Point; dx,dy:INTEGER)
  BEGIN
    (* Codice Move_Point *)
  END Move_Point;

  p : Point;
```

```
TYPE
  CircleDesc =
    RECORD( PointDesc )
      radius : INTEGER;
    END;

  Circle = POINTER TO CircleDesc;

  PROCEDURE Move_Circle(p:Point; dx,dy:INTEGER)
  BEGIN
    (* Codice Move_Circle *)
  END Move_Circle;

  c : Circle;

  p.move := Move_Point;
  c.move := Move_Circle;
```

Il precedente esempio permette inoltre di illustrare il meccanismo che associa una procedura ad una classe. Infatti, in Oberon i tipi procedura, più che le procedure stesse, sono associate agli oggetti nel codice del programma.
Il collegamento delle procedure attuali agli oggetti è rimandato al momento in cui il programma è in esecuzione.
L'associazione di un tipo procedura con un tipo di dato occorre nel momento della dichiarazione dei campi di un record (nell'esempio la dichiarazione del campo move).
L'associazione di un "metodo" con un oggetto occorre nel momento in cui si assegna

[1] Fanno eccezione i parametri passati per variabile di tipo record che si comportano come i puntatori.

uno specifico valore di tipo procedura al campo del record (nell'esempio p.move := Move_Point).

Il fatto di legare tipi procedura a tipi record consente al tipo record di mantenere inalterata la sua struttura e di conservare la natura di tipo puro. In questo modo un tipo record rappresenta sia l'interfaccia che l'implementazione di una classe.
Pur non avendo alcun legame diretto con le classi e quindi con gli oggetti, le procedure ad essi associate sono operazioni virtuali, attivate con binding dinamico.
Si può, infatti, definire una procedura che implementa il comportamento della classe discendente.
Considerando il precedente esempio, associato al seguente stub di codice

```
p.move( p, 1, 1 );
p := c;
p.move( p, 2, 1 );
```

si può osservare che mentre la prima chiamata di p.move invoca la procedura Move_Point in quanto il tipo di p è Point, la seconda, invece, invoca la procedura Move_Circle in quanto il tipo di p è Circle. Questo esempio esprime perfettamente il concetto di binding dinamico presente in Oberon.
Va osservato che il cambiamento dinamico di tipo avviene solo se le variabili coinvolte hanno tipo puntatore a record, e non record. Se le variabili fossero dei record invece che dei puntatori a record, l'assegnazione non comporterebbe un polimorfismo per sotto-tipazione, ma bensì un'operazione di proiezione tra record.
Ad esempio se p fosse di tipo PointDesc e c di tipo CircleDesc allora l'istruzione p := c; avrebbe come risultato l'assegnazione ai campi di p, x, y e move del valore dei corrispondenti campi di c, mentre il valore del campo radius di c non verrebbe copiato in p perché non ha alcun campo corrispondente nel tipo PointDesc.

Disponibilità a Run-Time delle informazioni sui Tipi (RTTI)

Come visto le procedure, in Oberon, possono essere ridefinite, dal momento che è possibile associare ad un campo di tipo procedura di una classe discendente una procedura differente da quella della classe antenata. Tuttavia, la ridefinizione delle procedure comporta, in generale, l'accesso ai campi aggiuntivi definiti da un tipo estensione. Poiché il tipo dei campi non può essere ridefinito, la segnatura delle procedure ridefinite deve essere identica a quella delle procedure relative alla classe antenata. Questa limitazione impedisce la visibilità dei campi definiti dalla classe discendente.
Tale accesso, tuttavia, può essere ottenuto tramite meccanismi di controllo dinamico del tipo, quali il type test ed il type guard.
Il type test (istruzione IS) consente di controllare a run-time il tipo di una variabile, mentre il type guard permette ad una variabile di assumere dinamicamente la struttura di un tipo discendente del tipo dichiarato staticamente.

I meccanismi per il controllo dinamico del tipo degli oggetti sono strumenti molto potenti, che aumentano notevolmente la flessibilità nella programmazione. Comunque, per implementare i metodi tradizionali nessun controllo di tipo dinamico è in realtà necessario; l'utilizzo del type test e del type guard per l'implementazione delle operazioni è legato, solamente, al fatto che essi rappresentano l'unico strumento che consente di accedere a tutti i campi definiti da un record.
In riferimento all'esempio precedente, la procedura Move_Circle per poter accedere al campo radius di p, deve accedere al campo radius definito dal tipo Circle, estensione di Point, mediante l'istruzione p(circle).radius. Una soluzione più elegante è ottenuta sfruttando il costrutto WITH che estende il type guard ad una sequenza di istruzioni. Il costrutto WITH può essere visto come un type guard regionale. La procedura Move_Circle può essere implementata come segue:

```
PROCEDURE Move_Circle( p : Point; dx, dy : INTEGER );
BEGIN
  WITH p : Circle DO
    (* Istruzioni necessarie per traslare il cerchio p di (dx, dy) *)
  END;
END Move_Circle;
```

All'interno del costrutto WITH p : Circle DO ... END la variabile p è considerata come se fosse stata dichiarata di tipo Circle e pertanto può accedere a tutti i campi appartenenti a questo tipo. Se a run-time il tipo dinamico di p non è Circle il programma viene abortito.

Moduli in Oberon

Un modulo è una collezione di dichiarazioni di costanti, tipi, variabili, procedure ed una sequenza di istruzioni atte ad inizializzare variabili. Il modulo normalmente è un file compilabile come un'unità. Tramite la lista di IMPORT è possibile indicare di quali moduli il modulo in questione è cliente. La sequenza di istruzioni di inizializzazione che seguono l'istruzione BEGIN viene eseguita unicamente quando un modulo viene aggiunto ad un programma.
Menzione particolare va al modulo predefinito SYSTEM, che contiene le procedure necessarie per effettuare operazioni di basso livello. Come ad esempio la possibilità di accedere ai dispositivi controllati dal computer e la possibilità di superare la regola di compatibilità tra tipi. Questo esporta in particolare il tipo BYTE, per il quale non sono specificati valori, ma solamente alcune regole di compatibilità:

① BYTE è compatibile con CHAR e SHORTINT

② se un parametro formale è di tipo ARRAY OF BYTE, allora i corrispondenti parametri attuali possono essere di qualsiasi tipo.

Oberon-2

Oberon-2 è un raffinamento del linguaggio Oberon, che introduce i metodi tradizionali chiamati procedure type-bound, la possibilità di esportare i dati in sola lettura, gli array aperti multidimensionali e l'istruzione FOR.
La caratteristica più saliente del linguaggio che non è presente nel suo predecessore, Oberon, sono le procedure legate al tipo.

Procedure Type-Bound

Le procedure dichiarate globalmente possono essere associate con un tipo record dichiarato nello stesso modulo. Tali procedure sono dette legate al tipo record. Il legame è espresso con il tipo del ricevitore nell'intestazione della dichiarazione di procedura. Il ricevitore può essere sia un parametro variabile di tipo record T o un parametro di tipo puntatore a T. La procedura è legata al tipo T ed è considerata locale a T (come i suoi campi).

```
Point =
  RECORD
    x, y : INTEGER;
  END;

PROCEDURE(p : Point) Move( x, y : INTEGER )
BEGIN
  (* Codice di Move Punti *)
END Move;
```

```
Circle =
  RECORD( Point )
    radius : INTEGER;
  END;

PROCEDURE(c : Circle)Move( x, y : INTEGER )
BEGIN
  (* Codice di Move cerchi *)
END Move;
```

Nel precedente esempio i parametri p e c delle procedure Move, legati, rispettivamente, ai tipi Point e Circle sono semanticamente equivalenti al parametro *self* dei linguaggi object-oriented tradizionali.
Se una procedura P (nell'esempio P coincide con Move, T_0 con Point e T_1 con Circle) è legata al tipo T_0 allora è implicitamente legata a T_1 che è una estensione di T_0. Comunque una procedura P', con il nome uguale a quello di P (nell'esempio la seconda procedura Move) può essere esplicitamente legata a T_1 in tal caso P' ridefinisce P. I parametri formali di P e P' devono essere dello stesso tipo e numero.
Se v è un designatore, P è una procedura legata al tipo dinamico di v allora v.P denota quella procedura, mentre v.P↑ denota la procedura P legata al tipo base del tipo di v.

EOS (Extended Oberon System)

EOS è un linguaggio di programmazione orientato agli oggetti derivato da Oberon, ed ispirato da Oberon-2. EOS non è una vera e propria estensione di Oberon in quanto presenta una differente regola di assegnabilità delle procedure, basata su una stretta equivalenza per nome e sulla possibilità di dare un tipo esplicito alle procedure.
Alcune estensioni di EOS sono state suggerite da Oberon-2, come ad esempio l'introduzione del concetto di metodo ed il concetto di procedure legate al tipo. La principale innovazione introdotta da EOS è rappresentata dai campi SHARED, altre innovazioni di rilevanza minore sono ad esempio l'importazione filtrante ed il l'esportazione *entension only*.

Campi Shared

Oltre quanto detto riguardo ai record illustrando Oberon, si ha che un tipo record può eventualmente contenere una parte condivisa, introdotta dalla parola chiave SHARE. I valori dei campi appartenenti alla parte shared sono condivisi da tutte le istanze di un record, a differenza degli altri campi i cui valori sono specifici solo per una singola istanza (esempio RECORD a : INTEGER; SHARE b : INTEGER END).
Oltre quanto detto per i tipi procedura illustrando Oberon, si ha che un tipo procedura può avere un parametro speciale dichiarato tra parentesi quadre. Tale parametro deve essere un parametro valore di tipo puntatore a record. Un tipo procedura avente questo parametro speciale è detto tipo procedura bound ad un record (per brevità record-bound). Sebbene qualsiasi variabile possa essere dichiarata di tipo record-bound, generalmente tale tipo viene attribuito ai campi di un record che rappresentano le operazioni relative ad una classe. Un campo di un record avente tipo record bound viene detto campo self-bound. Un campo self-bound deve necessariamente avere un tipo procedura bound al record contenente il campo stesso.
Nella dichiarazione di un tipo record-bound il parametro tra parentesi quadre può essere seguito da un identificatore di tipo record-bound. In tal caso viene stabilita una relazione di sottotipazione tra questi due tipi, che riflette la relazione esistente tra i record puntati del tipo del parametro speciale.
Esempio

```
Point = POINTER TO PointDesc;
Circle = POINTER TO CircleDesc;
P1 = PROCEDURE [self : Point]( x, y : INTEGER );
C1 = PROCEDURE [self : Circle] : P1;
```

Esportazione per la Sola Estensione

In Oberon un'entità definita in un modulo, marcandola con *, può essere esportata in altri moduli, clienti di quest'ultimo, per poter essere utilizzata ed eventualmente modificata. In EOS questa possibilità associata alla presenza di componenti *condivise*, e quindi modificabili comunque dai moduli importatori, è in contrasto con il concetto di encapsulation presente nei linguaggi ad oggetti. Per venire incontro all'esigenza di proteggere gli oggetti da modifiche improprie, in EOS è stata introdotta la possibilità di esportare i campi di un record in modalità *extension only*, marcandoli con ∼. I campi definiti in questo modo sono accessibili esclusivamente dalle procedure che estendono il record che lo definisce. Questo è un concetto analogo al concetto protected presente in C++.

Importazione Filtrante

L'importazione filtrante è un meccanismo base per la condivisione di codice e comportamento e per la riusbilità delle implementazioni. L'importazione filtrante rende possibile esportare ai clienti entità importate da altri moduli.
L'importazione filtrante è un meccanismo particolarmente efficace per l'implementazione delle interfacce.

EOS implementa l'importazione filtrante tramite i marker "*" e "∼", che consentono di esportare ai clienti le entità definite in un modulo, e la dichiarazione DEFS, le cui caratteristiche sono le seguenti:

- Consente di esportare ai clienti le entità importate da altri moduli. Il nome locale e quello importato fanno riferimento alla stessa entità, cioè se un modulo M_1 definisce il tipo T_1 ed il modulo M_2 importa il modulo M_1 e definisce T_2 come l'importazione filtrante di T_1 (DEFS $\mathsf{T}_2 = \mathsf{M}_1.\mathsf{T}_1$) allora se il modulo M_3 importa sia M_1 che M_2 questo "vede" T_1 e T_2 come se fossero lo stesso tipo.
- Consente di accedere ad entità importate da altri moduli in modo non qualificato, cioè usando l'esempio del caso precedente si utilizza il tipo T_2 in luogo di $\mathsf{M}_1.\mathsf{T}_1$ per definire variabili di tipo T_1.

Per un maggiore approfondimento di questa variante di Oberon si rimanda comunque ad [Ancona-Moreno 94]

C Hashing e Dizionario dei Canali

C.1 Tecniche di Hashing: Nozioni Generali

Le tecniche di hashing, sia statiche che dinamiche sono dei metodi usati per implementare un insieme in modo che il reperimento degli elementi sia ottenibile in modo efficiente. Ogni elemento è rappresentato da due componenti: la chiave e l'informazione vera e propria. Una funzione detta di hash si occupa di codificare, in tempo costante, la chiave in modo da distribuire le informazioni il più omogeneamente possibile all'interno della struttura dati che implementa l'insieme.
I problemi derivano dall'impossibilità di definire una funzione di hash che sia bigettiva se non presupponendo uno spazio equipotente a quello delle chiavi, cioè non si riesce a definire una funzione che date due chiavi distinte produca sempre due valori distinti (iniettività) e che ricopra tutto l'insieme destinazione omogeneamente (surgettività); la non iniettività della funzione di hash da origine ad un problema detto collisione delle chiavi, cioè due chiavi vengono ad avere la stessa codifica e quindi rappresentano lo stesso elemento dell'insieme, inserendo così due valori distinti nello stesso slot della struttura rappresentante l'insieme.
Il problema delle collisioni può essere risolto in diversi modi: se l'hashing è di tipo statico, cioè l'insieme ha una dimensione fissata e non può essere aumentata durante l'esecuzione, si possono sfruttare delle strutture aggiuntive in cui andare a mettere le informazioni in eccesso, oppure calcolare una nuova chiave cercando così un elemento non ancora utilizzato, comunque con entrambe le soluzioni si provoca un degrado della complessità, nel caso peggiore, da costante a lineare; se l'hashing è di tipo dinamico (hashing virtuale, estendibile, a spirale, ...), cioè l'insieme ha una dimensione che può essere ampliata a seconda delle esigenze, il problema è risolto aumentando lo spazio destinato all'insieme ed in alcuni casi sfruttando anche delle strutture ausiliarie ed adeguando la funzione di hash alle modifiche.

C.2 Spazio Occupato dal Dizionario dei Canali

Lo spazio allocato per contenere l'indice, nel caso peggiore, in seguito ad un numero elevato di collisioni può raddoppiare in maniera spropositata. Questo rappresenta una spreco di spazio. Questa situazione può essere scongiurata riformulando la funzione di hash od aumentando il numero di elementi inseribili in ogni slot di index. Lo stesso Fagin in [Fagin 79] ha dimostrato che lo spazio occupato dall'indice nel caso effettivo è $\mathcal{O}(\xi^2)$ dove ξ è il numero di elementi inseriti nella struttura; questa stima è comunque una maggiorazione della complessità effettiva che da prove pratiche risulta essere $\mathcal{O}(\xi)$ e quindi più che accettabile.

C.3 Cancellazione Implicita in Assenza del Garbage-Collector

Nell'analisi del dizionario dei canali presentata nella sezione [2.6] si è supposto di effettuare l'operazione di cancellazione implicita eliminando semplicemente il riferimento al sotto-albero e lasciando poi l'effettiva liberazione della memoria al garbage-collector, questa è una soluzione efficiente e ragionevole, infatti il garbage-collector se non è intrinseco nel supporto a tempo di esecuzione del linguaggio come in Oberon o SmallTalk è comunque presente nell'ambiente di sviluppo, come ad esempio nel Borland C++ o nel sistema operativo.
In questa sezione viene esaminato il costo di un'implementazione dell'operazione di cancellazione implicita in un ambiente privo del garbage-collector, sfruttando il seguente algoritmo:

1. Individuazione dell'oggetto nell'insieme dei sender. Questo è fattibile in tempo costante. $\mathcal{O}(1)$

2. Cancellazione del sotto-albero corrispondente. Il sotto-albero rappresenta tutti i canali che coinvolgono quel sender, il quale sarà collegato con un certo numero r di receiver, e con ciascuno di essi in un certo numero t di modi. $\mathcal{O}(\mathrm{r}\cdot\mathrm{t})$

3. Cancellazione dell'oggetto dall'insieme dei sender. Si tratta di cancellare un elemento da un insieme, operazione eseguibile in tempo costante. $\mathcal{O}(1)$

4. Per ognuno dei sender rimasti nel dizionario: $\mathcal{O}(\mathrm{s})$

    (a) Si determina se nel corrispondente insieme dei receiver è presente l'oggetto in questione. $\mathcal{O}(1)$

    (b) In caso affermativo si cancella il sotto-albero corrispondente. Questo sotto-albero rappresenta tutti i canali instaurati tra un dato sender ed un dato receiver, per un totale di t tipi, tanti quanti sono i canali instaurati

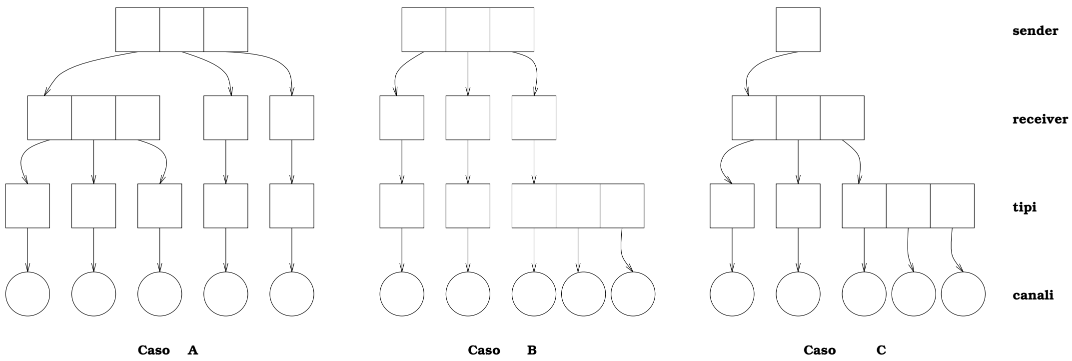


Figura C.1: **Casi impossibili**

tra i due oggetti e per ogni tipo il riferimento al canale vero e proprio. $\mathcal{O}(t)$

(c) Cancellazione del receiver in questione e, se non ci sono altri receiver per quel sender, anche il sender stesso. $\mathcal{O}(1)$

Quindi la complessità dell'operazione di cancellazione implicita è $\mathcal{O}(\max\{r{\cdot}t,\ s{\cdot}t\})$ dove s, r e t sono rispettivamente il numero massimo di sender, receiver e tipi di canali coinvolti;
Prima di effettuare qualsiasi maggiorazione è bene fare alcune osservazioni. Innanzittutto le tre quantità possono tutte assumere valori nell'insieme $\{1, \ldots, n\}$, dove n è il numero dei canali nel dizionario anche se non possono assumere il valore massimo contemporaneamente. Vediamo perché:

- ⋆ Supposto che sia s = n, se r = n vuol dire che uno degli n sender è collegato con n receiver e nell'ipotesi ottimistica che gli altri n - 1 sender siano collegati rispettivamente con solo un receiver e che ogni collegamento sia effettuato tramite un solo tipo di canale, si avrebbero n + n - 1 canali aperti, che è più di quanto è possibile (vedere situazione in figura [C.1.A] con n = 3).

- ⋆ Supposto che s = n e t = n significa che uno degli n sender è collegato con un receiver tramite n canali diversi, e nell'ipotesi ottimistica che gli altri n - 1 sender siano collegati rispettivamente con un solo receiver tramite un solo tipo di canale, nel dizionario sarebbero inseriti n + n - 1 canali, che è assurdo. (vedere situazione in figura [C.1.B] con n = 3).

- ⋆ Supposto che r = n e t = n significa che un sender è collegato con n receiver ed il collegamento con uno di questi è effettuato tramite n tipi di canali diversi, ed anche supponendo che non ci siano altri sender e che i rimanenti n - 1 receiver siano collegati in al più un modo con tale sender, si hanno nel dizionario n + n - 1 canali, che sono un numero maggiore di quanti ne esistono effettivamente (vedere situazione in figura [C.1.C] con n = 3).

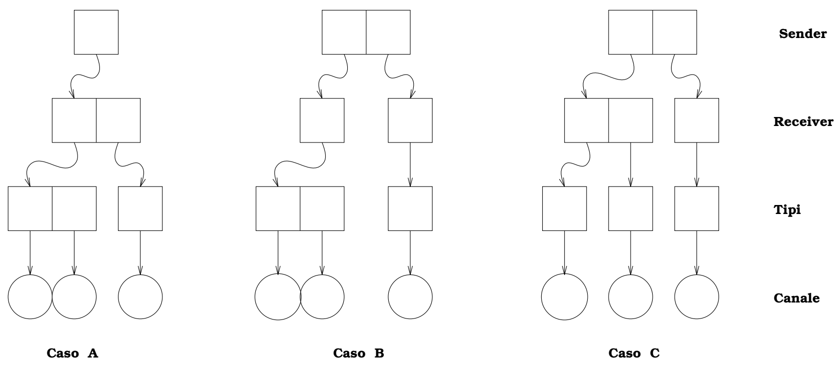


Figura C.2: **Esempi di Upper (A, B) e Lower (C) Bound per la quantità max{r·t, s·t}**

Quindi si può affermare che le tre quantità sono legate dalla relazione

$$\boxed{\mathrm{s} + \mathrm{r} + \mathrm{t} \leq \mathrm{n} + 2}$$

Questa formula segue dalle osservazioni precedenti ed è confermata dalla seguente osservazione: supposto che s + r + t sia maggiore di n + 2 allora una delle tre quantità è aumentata, ma qualunque sia questa quantità, la verifica costruttiva della struttura porta ad avere più di n canali, che è un assurdo.

A questo punto si possono esaminare i casi peggiori per le due quantità (s·t e r·t). Nel caso di r·t, il caso peggiore è da considerare quando il sotto-albero da cancellare coincide con l'intero albero, quindi nel caso in cui s = 1 (come nella figura [C.2.A]). Ovviamente questo caso non consiste anche nel caso peggiore per s·t, nel caso di s = 1 s·t ≡ t, che, essendo r un intero maggiore o uguale a 1, è una quantità minore di r·t, quindi in questa situazione la complessità è $\mathcal{O}$(r·t).
Il caso peggiore per la quantità s·t è rappresentato dal caso in cui ogni sender comunica con un solo receiver (r = 1) attraverso molti canali (come nella figura [C.2.B] per n = 3). Ovviamente anche questo caso non rappresenta anche il caso peggiore per l'altra quantità, infatti nel caso di r = 1, r·t≡t ed essendo s un intero maggiore od al più uguale ad uno, è una quantità minore od al più uguale ad s·t. Quindi in questo caso la complessità è di $\mathcal{O}$(s·t).
Queste due quantità si equivalgono in quanto nel caso s = 1 si ha che r + t ≤ n + 1 e nel caso che r = 1 si ha che s + t ≤ n + 1. Quindi posso considerare i due casi peggiori come analoghi e continuare la trattazione in un solo caso sostituendo ad r e s un generico x.
Quindi si utilizza x + t ≤ n + 1 e $\mathcal{O}$(x·t), mentre il valore massimo in funzione di n per x·t viene determinato quando x e t hanno pressoché lo stesso valore e la loro somma è minore o uguale a n + 1. Siccome interessa il caso peggiore allora si

considera la situazione in cui vale $x + t = n + 1$.
Siccome le quantità che rappresentano x e t non possono essere frazionarie occorre prestare attenzione al fatto che n sia pari o dispari.
Se n è pari $x = \lceil \frac{n+1}{2} \rceil$ e $t = \lfloor \frac{n+1}{2} \rfloor$ o viceversa. Se n è dispari $x = t = \frac{n+1}{2}$
Quindi $x \cdot t \equiv (\frac{n+1}{2})^2$ se n dispari e $x \cdot t = \lceil \frac{n+1}{2} \rceil \cdot \lfloor \frac{n+1}{2} \rfloor$ se n pari; comunque la complessità nel caso peggiore è in $\mathcal{O}(n^2)$.
Però questa è una maggiorazione nel caso peggiore e non medio.
Il caso migliore è rappresentato da una situazione in cui $n>1$ ed i collegamenti tra due oggetti sono fatti in al più un modo $t = 1$ (vedi figura [C.2.C]) e in questo caso la complessità è in $\Omega(n)$.

Bibliografia

FAULT TOLERANCE

[Ancona-Clematis-Dodero-Fernandez-Gianuzzi 90] Massimo Ancona, Andrea Clematis, Gabriella Dodero, Eduardo B. Fernandez e Vittoria Gianuzzi, "*A Systems Architecture for Fault Tolerance in Concurrent Software*", IEEE Computer, pages 23-31, October 1990.

[Ancona-Clematis-Dodero-Gianuzzi-Lisboa 95a] Massimo Ancona, Andrea Clematis, Gabriella Dodero, Vittoria Gianuzzi e Maria Lucia Lisboa, "*An Object Oriented Approach To Fault Tolerance Software*" 3^{rd} Euromicro Workshop on Parallel and Distribuited Processing, IEEE Computer Society, San Remo Italia, January 1995.

[Arjuna 92] The Arjuna Research Group, "*The Arjuna System Programmer's Guide*", Public Release 1.1, Department of Computing Science, University of Newcaste-Upon-Tyne, July 1992.

[Avižienis 85] Algirdas Avižienis, "*The N-Version Approach To Fault Tolerant Software*", IEEE Transaction on Software Engineering, Vol. SE-11 N° 12, pages 1491-1501, December 1985.

[Burlando-Gianetto-Mainini 92] Paola Burlando, Laura Gianetto, Maria Teresa Mainini, "*Functional Diversity*", In [Kersken-Saglietti 92] pages 49-113, New York 1992.

[Chérèque-Powell-Reynier-Richier-Voiron 92] M. Chérèque, David Powell, P. Reynier, J. L. Richier, J. Voiron, "*Active Replication in Delta-4*", In Proceeding of the 22^{nd} IEEE International Symposium on Fault Tolerant Computing (FTCS-22), Computer Society Press,pages 28-37, 1992.

[Clematis-Dodero-Gianuzzi 92] Andrea Clematis, Gabriella Dodero, Vittoria Gianuzzi, "*Process Checkpointing Primitives for Fault Tolerance: Definitions and Examples*", In Microprocessors and Microsystems Vol. 16 N°1, pages 15-23, 1992.

[Di Giandomenico-Stringini 90] Felicita Di Giandomenico, Luigi Stringini, "*Adjudicators for Diverse-Redundant Components*", Dependability Prediction and Evaluation, PDCS Esprit Project Report 1990.

[Ezhilchalvan-Shrivastava 86] Paul D. Ezhilchalvan e Santosh K. Shrivastava, "*A Characterization of Faults in Systems*", Proc. 5th ACM Symposium on principles of Distribuited Software and DataBase Systems, pages 215-222, January 1986.

[Kersken-Saglietti 92] Manfred Kersken, Francesca Saglietti Editors, "*Software Fault Tolerance: Achievement and Assessment Strategies*", Research Reports Esprit Vol. 1, Springer Verlag New York 1992.

[ISIS 90] Kenneth P. Birman, Robert Cooper, Thomas A. Joseph, Keith Marzullo, Mesaac Makpoungu, Kenneth Kane, Frank Schmuck, M. Wood, "*The ISIS System Manual V2.0*", Cornell University's Computer Science Department Technical Report, May 1990.

[Kim 82a] Kane H. Kim, "*Software Fault Tolerance*", Handbook of Software Engineering, pages 437-454, 1982.

[Kim 82b] Kane H. Kim, "*Approaches to Mechanization of the Conversation Scheme Based on Monitors*", IEEE Transactions on Software Engineering, Vol. SE-8, N°3, pages 189-197, May 1982.

[Kim 84] Kane H. Kim, "*Distribuited Execution of Recovery Blocks: on Approach to Uniform Treatment of Hardware and Software faults*", In proceeding of 4th International Conference on Distribuited Computing Systems, pages 526-532, May 1984.

[Powell 91] David Powell, "*Delta-4: A Generic Architecture for Dependable Distribuited Computing*", Research Reports Project 818|2252 Esprit, 484 pages, Springer-Verlag, Berlin, Germany, 1991.

[Randell 75] Brian Randell, "*System Structure for Software Fault Tolerant*", IEEE Transaction on Software Engineering, Vol. SE-1 N°2, pages 220-232, June 1975.

[Schneider 90] Fred B. Schneider, "*Implementing Fault Tolerant Services Using the State Machine Approach: a Tutorial*", In ACM Computing Surveys Vol. 22 N°4, pages 299-319, December 1990.

[Speirs-Barret 89] N. A. Speirs, P. A. Barret, "*Using Passive Replicates in Delta-4 to Provide Dependable Computing*", In Proceeding of the 19th IEEE International Symposium on Fault Tolerant Computing (FTCS19), pages 184-190, 1989.

[Xu-Liskov 89] Andrew Xu, Barbara Liskov, "*A Design for a Fault Tolerant, Distribuited Implementation of Linda*", In Proceeding of the 19th IEEE International Symposium on Fault Tolerant Computing (FTCS19), pages 199-206, 1989.

[Baumgartner-Russo 93] Gerald Baumgartner, Vincent F. Russo, "*Signatures: A C++ Extension for Type Abstaction and Subtype Polymorphism*", Technical Report CSD-TR-93-059 of Department of Computer Sciences Purdue University West Lafayette, IN 47907 USA September 1993.

[Birrel-Nelson 84] Andrew D. Birrel, Bruce Jay Nelson, "*Implementing Remote Procedure Calls*", ACM Transactions on Computer Systems, Vol. 2 N°1, pages 39-59, February 1984.

[Fagin 79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger and Raymond H. Strong, "*Extendible Hashing - A Fast Access Method for Dynamic Files*", ACM Transaction on DataBase Systems, Vol.4, n°3, pages 315-344, September 1979.

[Horowitz-Sahni-Andersen–Freed 93] Ellis Horowitz, Sartaj Sahni and Susan Andersen–Freed, "*Fundamentals of Data Structures in C*", Freeman and Company, New york, 1993.

[Rogers 95] John Rogers, "*Your Own Endian Engine*", In Dr. Dobb's Journal, pages 30-36, November 1995.

[Silberschatz-Peterson-Galvin 91] Abraham Silberschatz, James L. Peterson, Peter Galvin, "*Operating System Concepts 3rd edition*", Addison Wesley 1991.

[Stroustrup-Lenkov 92] Bjarne Stroustrup and Dmitry Lenkov, "*Run-Time Type Identification for C++: A Proposal for New Features to the Language*", Proceedings, 1992 USENIX C++ Conference, pages 32-42, Portland Oregon, March 1992.

[Sun 87] Sun Microsystems, Inc. "*XDR: External Data Representation Standard.*" RFC 1014, Sun Microsystems, Inc., June 1987.

[Vines-Kishimoto 93] Don Vines, Zen Kishimoto, "*SmallTalk's run-time type support for C++: Emulation of SmallTalk typing in C++ facilities CORBA implementation*", C++ Report pages 44-52 January 1993.

[Wirth 76] Niklaus Wirth, "*Algorithms+Data Structures = Programs*", Prantice-Hall, Inc – EngleWood Cliff, New Jersey, 1976.

[Wirth 85] Niklaus Wirth, "*Programming in Modula-2 3^{rd} corrected edition*", Springer-Verlag 1985.

OBJECT ORIENTED

[Ancona-Moreno 94] Massimo Ancona, Luciano Moreno, "*EOS: Una estensione di Oberon verso la programmazione ad oggetti*", Tesi di Laurea, DISI Università di Genova, Febbraio 1994.

[Goldberg-Robson 83] Adele Goldberg and David Robson, "*SmallTalk-80 : The language and its implementation*", Addison-Wesley Reading, Massachusetts 1983.

[Marchesi 89] Michele Marchesi, "*SmallTalk e la Programmazione Object-Oriented*", Gruppo Editoriale Jackson 1989.

[Masini-Napoli-Colnet-Léonard-Tombre 89] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard et Karl Tombre, "*Les Languages à objects*", Inter Editions, Paris 1989.

[Meyer 88] Bertrand Meyer, "*Object-Oriented Software Construction*", Prantice-Hall, Inc – EngleWood Cliff, New Jersey, 1988.

[Mössenböck 91] Hans-Peter Mössenböck, "*Object-Oriented programming in Oberon-2*", Technical Report, E.T.H. Zürich, Institut für Computersysteme, 1991.

[Mössenböck-Wirth 92] Hans-Peter Mössenböck, Niklaus Wirth, "*The Programming Language Oberon-2*", Technical Report, E.T.H. Zürich, Institut für Computersysteme, June 1992.

[Reiser-Wirth 92] Martin Reiser and Niklaus Wirth, "*Programming in Oberon: Steps Beyond Pascal and Modula*", ACM Press 1992.

[Snyder 86] Alan Snyder, "*Encapsulation and Inheritance In Object-Oriented Programming Languages*", In Proceeding of OOPSLA '86, ACM Sigplan Notices, Vol.21 N°11, pages 38-45, Portland Oregon USA, December 1986.

[Stroustrup 91] Bjarne Stroustrup, "*The C++ Programming Language 2nd Edition*", Addison-Wesley, Reading MA, 1991.

[Wegner 87] Peter Wegner, "*Dimensions of Object Based Language Design*", Proceeding of OOPSLA '87, ACM Sigplan Notices, Vol. 22 N°12, pages 168-182, Orlando, USA October 1987.

[Wegner 90] Peter Wegner, "*Concepts and Paradigms of Object-Oriented Programming*", OOPS Messenger Vol.7 N°7 pages 8-87, June 1990.

[Wegner 92] Peter Wegner, "*Dimensions of Object-Oriented Modelling*", In IEEE Computer, pages 12-20, October 1992.

PARALLELISMO

[Agha 86] Gul Agha, "*A Model of Concurrent Computation in Distribuited Systems*", MIT Press 1986.

[Barker-Hewitt 77] H. G. Barker Jr, C. Hewitt, "*The Incremental Garbage Collection of Processes*", In Proceeding of ACM Symposium on Artificial Intelligence and Programming Languages, pages 55-59, ACM, New York, August 1977.

[Ben-Ari 90] M. Ben-Ari, "*Principles of Concurrent and Distribuited Programming*", Prantice-Hall International Ltd, Hemel Hempstead 1990.

[Birman-Joseph 87] Kenneth P. Birman, Thomas A. Joseph, "*Reliable Communication in the Presence of Failures*", ACM Transaction on Computer Systems, Vol.5 N°1, pages 47-76, February 1987.

[Goldman-Gabriel 89] Ron Goldman, Richard P. Gabriel, "*Q-Lisp: Parallel Processing in Lisp*", In IEEE Software, pages 51-59, July 1989.

[Hoare 74] C.A.R. Hoare, "*Monitors: An Operating System Structuring Concepts*", Communications of the ACM, Vol.17, N°10, pages 549-557, 1974.

[Liang-Chanson-Neufeld 90] Luping Liang, Samuel T. Chanson, Gerald W. Neufeld, "*Process Groups and Group Communications: Classifications and Requirements*", IEEE Computer, Vol. 23 N°2, pages 56-66, February 1990.

[Manchek 94] Robert Manchek, "*PVM Design*", Master's Thesis University of Tennesee, June 1994.

[Mullender 89] Sape Mullender Editor, "*Distribuited Systems*", ACM Press, Addison Wesley, Reading, New York 1989.

[Needham 89] R. M. Needham, "*Names*", In [Mullender 89], pages 89-101, 1989.

[PVM3 94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek,Vaidy Sunderam, "*PVM 3 User's Guide and Reference Manual*", Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennesee 37831, May 1994.

[Satyanarayanan 89] M. Satyanarayanan, "*Distribuited File Systems*", In [Mullender 89], pages 149-188, 1989.

[USDoD 80] U.S.Department of Defence, "*Ada Reference Manual*", U.S. Government Printing Office, Washington DC, 1980.

[Zorn-Ho-Larus-Semenzato-Hilfinger 89] Benjamin Zorn, Kinson Ho, James Larus, Luigi Semenzato, Paul Hilfinger, "*Multiprocessing Extensions in Spur Lisp*", In IEEE Software, pages 41-49, July 1989.

PARALLELISMO e OBJECT-ORIENTED

[America 87] Pierre America, "*POOL-T: A Parallel Object-Oriented Language*", In Object-Oriented Concurrent Programming, editors Akinori Yonezawa and Mario Tokoro, pages 198-220, MIT Press 1987.

[Andrew-Hutchinson-Jul-Levy 86] A. Andrew, N. Hutchinson, E. Jul, H. Levy, "*Object Structure in the Emerald System*", In Proceeding of OOPSLA '86, ACM Sigplan Notices, Vol.21 N°11, pages 78-85, December 1986.

[Bennett 87] John K. Bennett, "*The Design and Implementation of Distribuited SmallTalk*", Proceeding of OOPSLA '87, ACM Sigplan Notices, Vol.22 N°12, pages 318-330, Orlando USA, October 1987.

[Briot-Yonezawa 87] Jean-Pierre Briot, Akinori Yonezawa, "*Inheritance and Synchronization in Concurrent OOP*", In Proceedings of ECOOP'87, vol.276 of Lecture Notes in Computer Science, pages 33-40, Springer-Verlag 1987.

[Briot 94] Jean-Pierre Briot, "*Modélisation et Classification de langages de programmation concurrente à objets: l'experience Actalk*", Proceeding of the Colloquium "Langages et Modèles à Objets", Grenoble, France, October 1994.

[Chin-Chanson 91] Roger S. Chin, Samuel T. Chanson, "*Distribuited Object-Based Programming Systems*", ACM Computing Surveys, Vol.23, N°1, pages 91-124, March 1991.

[Corba 93] Object Management Group (OMG), "*The Common Object Request Broker: Architecture and Specification V1.2*", Technical Report N° 93.12.1, December 1993.

[Kafura-Lee 89] Dennis G. Kafura, Keung Hae Lee, "*Inheritance in Actor Based Concurrent Object*", In Proceedings of ECOOP'89, pages 131-145, Cambridge University Press, July 1989.

[Lalis-Sanders 94] Spiros Lalis, Beverly A. Sanders, "*Adding Concurrency to the Oberon System*", Technical Report, E.T.H. Zürich, Institut für Computersysteme, Switzerland 1994.

[Liskov-Scheifler 83] Barbara Liskov, R. Scheifler, "*Guardians and Actions, Linguistic Support for Robust Distribuited Programs*", TOPLAS vol. 5 N°3, pages 381-404, 1983.

[Matsuoka-Yonezawa 92] Satoshi Matsuoka, Akinori Yonezawa, "*Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*", Technical Report Department of Information Science, The University of Tokyo 1992.

[Okamura-Tokoro 90] Hideaki Okamura, Mario Tokoro, "*Concurrent SmallTalk-90*", Technical Report Department of Information Science, The University of Tokyo 1990.

[Vinoski 93] Steven Vinoski, "*Distribuited Object Computing with CORBA*", C++ Report 5(6), pages 33-38, July-August 1993.

[Wyatt-Kavi-Hufnagel 92] Barbara B. Wyatt, Krishna Kavi, Steve Hufnagel, "*Parallelism in Object-Oriented Languages: a Survey*", IEEE Software, pages 56-86, November 1992.

[Yokote-Tokoro 86] Yusuhiko Yokote, Mario Tokoro, "*The Design and Implementation of Concurrent SmallTalk*", Proceeding of OOPSLA '86, ACM Sigplan Notices, Vol.21 N°12, pages 331-340, Portland Oregon USA, October 1986.

[Yokote-Tokoro 87] Yusuhiko Yokote, Mario Tokoro, "*Experience and Evolution of Concurrent SmallTalk*", Proceeding of OOPSLA '87, ACM Sigplan Notices, Vol.22 N°12, pages 406-415, Orlando USA, October 1987.

[Yonezawa 90] Akinori Yonezawa, "*ABCL: an Object-Oriented Concurrent System*", MIT Press 1990.

PARALLELISMO e RIFLESSIONI

[Chiba 93] Shigeru Chiba, "*Open C++ Release 1.2 Programmer's Guide*", Technical Report N°93-3, Department of Information Science, University of Tokyo, 1993.

[Chiba-Masuda 93] Shigeru Chiba, Takashi Masuda, "*Design an Extendible Distribuited Language with a Meta-Level Architecture*", In Proceeding of 7$^{\text{th}}$ European Conference on Object-Oriented Programming (ECOOP 93), pages 482-501, Kaiserslautern Germany, July 1993.

[Kono-Tokoro 91] Shinji Kono, Mario Tokoro, "*Parallel Reflection*", Sony CSL Technical memo SCSL-TM-90-011, June 1991.

[Masuhara-Matsuoka-Watanabe-Yonezawa 92] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, Akinori Yonezawa, "*Object-Oriented Concurrent Reflective languages can be implemented efficiently*", In proceeding of OOPSLA'92, ACM Sigplan Notices Vol.27 N° 12, Vancouver Canada, October 1992.

[Matsuoka-Watanabe-Yonezawa 91] Satoshi Matsuoka, Takuo Watanabe, Akinori Yonezawa, "*Hybrid group reflective architecture for object-oriented concurrent reflective programming*", In proceeding of 5^{th} European Conference on Object-Oriented Programming, Geneve, Switzerland, July 1991.

[Stroud 93] Robert J. Stroud, "*Transparency and Reflection in Distribuited Systems*", ACM Operanting Systems Review, Vol. 22 N°2, pages 99-103, April 1993.

[Watanabe-Yonezawa 88] Takuo Watanabe, Akinori Yonezawa, "*Reflection in an Object-Oriented Concurrent Language*", In Proceeding of OOPSLA'88, ACM Sigplan Notices, Vol. 23 N°12, pages 306-315, September 1988.

[Watanabe-Yonezawa 90] Takuo Watanabe, Akinori Yonezawa, "*An Actor-Based Metalevel Architecture for Group-Wider Reflection*", In J. W. de Bakker, W. P. de Rover, G. Rozenberg editors, Fondations of Object-Oriented Languages, pages 405-425, Springer-Verlag 1990.

[Yokote-Mitsuzawa-Fujinami-Tokoro 91] Yusuhiko Yokote, A. Mitsuzawa, N. Fujinami, Mario Tokoro, "*The Muse Object Architecture: A New Operating System Structuring Concept*", Technical Report SCSL-TR-91-002, Sony Computer Science Laboratory Inc., February 1991.

RIFLESSIONI

[Attardi-Bonini-Boscotrecase-Flagella-Gaspari 89] G. Attardi, C. Bonini, M. R. Boscotrecase, T. Flagella, M. Gaspari, "*Meta-Level Programming in CLOS*", In Proceeding of ECOOP'89, pages 243-256 1989.

[Chiba 95] Shigeru Chiba, "*A MetaObject Protocol for C++*", Proceeding of OOPSLA '95, ACM Sigplan Notices, Vol.30 N°12, Austin, Texas, USA, October 1995.

[Cointe 87] Pierre Cointe, "*MetaClasses are first class objects : the ObjVLisp model*", Proceeding of OOPSLA '87, ACM Sigplan Notice, Vol.22 N°12, Orlando, USA October 1987.

[Duby-Meyers-Reiss 92] Carolyn K. Duby, Scott Meyers, Steven P. Reiss, "*CCEL: A Metalanguage for C++*", Department of Computer Science Brown University Providence, Rhode Island 02912 CS-92-51 October 1992.

[Ferber 89] Jacques Ferber, "*Computational Reflection in Class Based Object Oriented Languages*", Proceeding of OOPSLA '89, ACM Sigplan Notices, Vol. 24 N°12, pages 327-335, December 1989.

[Foote-Johnson 89] Brian Foote e Ralph E. Johnson, "*Reflective Facilities in SmallTalk-80*", OOPSLA '89, Sigplan Notices, Vol. 24 N°12, October 1989.

[Lee-Zachary 94] Arthur H. Lee, Joseph L. Zachary, "*Using Meta Programming to Add Persistence to CLOS*", 1994 International Conference on Computer Languages, Los Alamitos, California, IEEE CS Press 1994.

[Lisboa-Tepedino-Meira 93] Paulo H. C. Lisboa, José Fernando Tepedino, Silvio Lemos Meira, "*Reflexão Computacional em SmallTalk*", Revista Brasileira De Computação, Rio de Janeiro Brazil, Vol.7 N°1 pages 13-24, July 1993.

[Maes 87] Pattie Maes, "*Concepts and Experiments in Computational Reflection*", OOPSLA '87, Sigplan Notices, Vol. 22 N°12, pages. 147-156, October 1987.

[Matsuoka-Yonezawa 90] Satoshi Matsuoka, Akinori Yonezawa, "*Metalevel Solution to Inheritance Anomaly in Concurrent Object-Oriented Languages.*", In Proceedings of ECOOP|OOPSLA 90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, October 1990.

[Pascoe 86] Geoffrey A. Pascoe, "*Encapsulators : A New Software Paradigm in SmallTalk-80*", OOPSLA '86, ACM Sigplan Notices, Vol. 21 N°12, pages 341-346, Portland Oregon USA October 1986.

[Rao 91] Ramana Rao, "*Implementational Reflection in Silica*", In Proceedings of ECOOP'91, Springer-Verlag, July 1991.

RIFLESSIONI e FAULT TOLERANCE

[Agha-Frølund-Panwar-Sturman 93] Gul Agha, Svend Frølund, Rajendra Panwar, Daniel Charles Sturman, "*A linguistic Framework for Dynamic Composition of Dependability Protocols*", In Proceeding of DCCA-3 pages 197-207 September 1993.

[Ancona-Cazzola-Dodero-Gianuzzi 95] Massimo Ancona, Walter Cazzola, Gabriella Dodero, Vittoria Gianuzzi, "*Channel Reification: a reflective approach to fault-tolerant software development*", Poster Section of OOPSLA'95, Austin Texas, October 1995.

[Ancona-Clematis-Dodero-Gianuzzi-Lisboa 95b] Massimo Ancona, Gabriella Dodero, Vittoria Gianuzzi, Andrea Clematis, Maria Lucia Lisboa, "*Reflective architectures for reusable fault-tolerant software*", proc. 1st Ibero American Microelectronics Conference, Canela, Brazil, pages 87-98, July 31 - August 4, 1995.

[Fabre-Nicomette-Pérennou-Wu 94] Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, Zhixue Wu, "*Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming*", In Proceeding of Predictably Dependable Computing Systems 2 (PDCS2) Second Year Report, pages 291-313, September 1994.

[Fabre-Nicomette-Pérennou-Stroud-Wu 95] Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, Robert Stroud, Zhixue Wu, "*Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming*", In Proceeding of the 25th IEEE International Symposium on Fault-Tolerant Computing (FTCS-25 "Silver Jubilee") Pasadena, CA USA, June 1995.

[Sturman 94] Daniel Charles Sturman, "*Fault Adaption for Systems in Unpredictable Environments*", University of Illinois' Master Thesis in Computer Science, 1994.