Channel Reification: a reflective approach to fault-tolerant software development

M. Ancona W. Cazzola G. Dodero V. Gianuzzi

DISI-University of Genoa, Via Dodecaneso 35, 16146 Genova, Italy
E-mail: {ancona|cazzola|dodero|gianuzzi}@disi.unige.it

October 9, 1995

Abstract

Reflective systems can be used to ease the implementation of fault tolerance mechanisms in distributed applications as show in [Anc95, Fab94]. In this paper we introduce a new model for reflective computations, and we show how it can be used for building up fault tolerant applications.

Keyword: Object-Orientation, Reflection and Fault Tolerance.

1 Introduction and Background

1.1 Fault Tolerance

Software mechanisms used to support fault tolerant applications include checkpointing facilities and replicated servers, for a survey see [Anc90], such fault tolerant behaviors can be implemented either by error processing protocols in the underlying runtime systems, or using pre-defined library functions and primitives, or using object-oriented methodologies, so making non-functional[1] characteristics inheritable. All these approaches have advantages and drawbacks: if the error processing mechanisms are provided by the underlying system, then transparency and separation of concerns[2] can be achieved, but this lacks of flexibility. If fault tolerance behavior is supported by predefined libraries then programmers can write their own error processing mechanisms but transparency and separation of concerns cannot be achieved. In object oriented systems, when using inheritance separation of concerns is achieved but transparency is not totally covered, because some programming conventions are required.

1.2 Reflection

Computational reflection is defined as the activity performed by an agent when doing computations about its own computation [Mae87]. Thus, a reflective system incorporates data structures representing itself in order to support actions on itself. A reflective object-oriented system may:

- monitor the behavior of its components and computations;
- dynamically acquire methods from other objects;
- make addition|deletion or changes to the set of its own methods.

In a reflective system there is a reflective tower of computational domains D_i. A reflective architecture thus defines a layered system, where the base level represents the application domain, and the other levels, the meta levels, represent the system domain.
Reflection makes it possible to open up a system implementation without revealing unnecessary implementation details, thus appearing to be an adequate mean for adding properties to a system. The use of meta level programming permits transparent separation of functional components from non-functional components in a system. Three basic approaches to computational reflection in class based languages have been pointed out in [Fer89, Mae87]:

- The metaclass model, where the class of an object plays the role of the metaobject, controlling the

[1] The non-functional characteristics are the features not needed to solve the problems but used for adding properties like fault tolerance, to the applications

[2] By separation of concerns, also called application transparency or reflection transparency, we mean separation of entities that perform application tasks from entities performing checkpoints and error recovery tasks.

execution of all its instance objects. This approach is considered "the canonical one" for its structural computational model, but it lacks of flexibility, in that all objects in a given class are controlled by the same metaclass.

- The metaobject model, where each object may be controlled by a corresponding metaobject. This approach is more flexible since it specializes metaobject behavior in accordance with the individual object, thus providing good encapsulation.

- The metacommunication model, which is based on the reification of messages sent to objects running at the base level. Instead of sending a message `M` to object `O`, `M` is reified into an object $\mathtt{M_O}$ and the special message "`handle`" is sent to it. This model is the most flexible one, due to its finer granularity of reflection with respect to the other models. On the other hand, it cannot preserve information between meta computations (called *lack of continuity*), and causes an extremely large number of object creations (an object is created for each method call).

Poor flexibility and lack of continuity respectively make the metaclass model and the metacommunication model not well suited for building up fault tolerant applications. In the following a new model called channel reification is proposed, and an example of a fault tolerant application built with the metaobject model, and with the channel reification model, is shown.

2 Channel Reification Model

This model extends the meta-communication approach with the aim of solving some of its drawbacks, while keeping its advantages. Channel reification is based on the following idea: a method call is considered as a message sent through a logical channel established between an object (or a group of objects[3]) requiring a service, and an object (or a group of objects), providing such service (both objects are involved in the call, in a client|server relationship as in figure 1). Then a logical channel is reified into an object called channel. A channel is characterized by the objects|groups it connects and by the kind of the computation performed. For kind of a channel we mean the reflective behavior provided by the channel. In a typed object oriented language the kind is also the type of the channel class. The kind is used to distinguish the reflective activity to be performed.

[3] An object group is an abstraction for a set of objects performing collective jobs, like a distributed execution of requests or management of replicated data [Yon87]

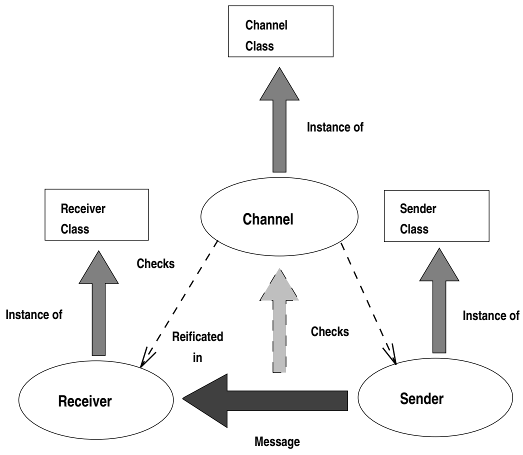


Figure 1: **Channel Reification Model Scheme**

The lack of continuity of the metacommunication model is eliminated by keeping channels after completion of each meta computation. Such a channel is reused when a meta computation of the same kind is performed. In this way, objects operating at meta level are created only once (when they are activated for the first time), and reused until required.
The channel is persistent i.e. it continues to exist after the method activation that originated it.
The features of the model are:

- ⋆ Finer granularity than other reflective approaches.
- ⋆ Continuity of information between meta computations. As we will show channel reification model is well suited for supporting fault tolerant applications where maintaining all links to replicas|versions is fundamental.
- ⋆ Possibility to connect more channels to the same object.
- ⋆ Different method calls can have different channels handling them, thus it is easy to specialize the reflective computations for each method.
- ⋆ Possibility to structure channels in an inheritance hierarchy. In fact, a channel is an object, i.e. an instance of a class that is can be structured in an inheritance hierarchy.

Applications of this model, as fault tolerance, are in distributed environments. In this case, an efficient

channel implementation is mandatory. In order to implement a channel linking two objects running on different sites, we create a meta object at each site, called stub, which is a representative of the channel: a pair of stubs implement a channel connecting two objects, residing on different sites. For groups, the channel connects the group coordinators at meta-level.
With the stubs|channel concept, distributed objects are low level components accessible to remote clients by means of channel supervision. Namely, a stub encapsulates all low level encoding|decoding between different processors and environments, besides that each stub takes care of performing its own meta computation.
For more details see [Caz95].
In the rest of this paper we will show how the reflective paradigm can be merged with fault tolerance. This is done by comparing different approaches to error recovery, developed using two reflective models: the metaobject model and the channel reification model.
The implementation requires that a method call is trapped and handled by the channel operating at meta level. We assume that method calls are implemented by `send`, which transfers control to the meta level and returns at the base level after execution of the meta computation (the shift up). The opposite change of layer (shift down), is made by `apply`. Meta computation is performed by `handle`.
This message passing mechanism is a transparent interface between non functional characteristics (meta level) and functional ones (base level).
In such a way fault tolerance can be added to each critical object organizing it within the meta level, without interfering with the structure of objects belonging to the application domain.

3 Passive Replication: An Example of Reflective System Fault Tolerance

Passive replication is based on replication of critical objects building up the application. Only one of the replicas (the primary replica) processes input messages and delivers output messages (in the absence of failures), while the remaining replicas (the backup replicas) only update their internal status when a checkpoint of the primary replica is performed.
In the first example, we assume the application to be composed by a client, a primary server and a single backup replica server. Client requests are processed by the primary server, and upon completion of status update operation, the server sends its new status to the backup replica. If the primary server crashes, the backup replica may provide continuous service to the client, and a new backup replica is created.

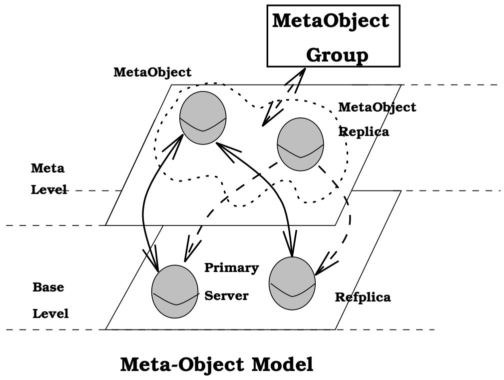


Figure 2: **Metaobject Passive Replication**

3.1 Passive Replication using the Metaobject Model

The reflective tower can be organized as in [Anc95], where a recovery level monitors the execution of the application programs and implements the recovery algorithms, accessing system primitives for fault tolerance support. The application level is only required to specify the fault tolerance technique to be used, to indicate the critical regions, and to supply the redundant code. The structure, of this model is shown in figure 2):

- At the base level there are the client, the primary and replicated server. These entities fulfill their own tasks without taking care of fault tolerance mechanisms.
- At the meta level there is a meta object (actually group) that manages all fault tolerant aspects of the application (i.e. replica creation, updating and recovery from failures).

The primary server is the only object processing input data, and its method calls are achieved by `send`. Then, the control passes to the metaobject, which handles the method call by `apply` execution. After the call has been serviced, if the object status is changed, the metaobject updates the status of the replica. If the primary object crashes, the metaobject authorizes the

replica to become the new primary server and creates a new replica.
To be able to overcome a metaobject failure, we should introduce an additional meta level with a meta-metaobject in charge of fault tolerance with respect to metaobjects of the underlying level. To avoid an infinite regression of metalevels|metaobjects, we close the reflective tower at meta level, by implementing a metaobject group that implements a fault tolerant metaobject.

3.2 Passive Replication using the Channel Reification Model

The metaobject model associates a fault tolerant scheme to application objects. On the other hand, channel reification provides an abstraction of object communication, encapsulating all fault tolerant operations.
In a distributed environment, objects can interact with different levels of fault criticality, as for example:

① "Normal" interaction between a client and a server. If the server fails, an error flag is returned.

② Interaction of a client object with a server group, composed by a set of server replicas. (see example in section 3.1).

③ Interaction between a client group, composed by a set of client replicas, and a server group.

Channel reification easily supports all the above cases: case ① by means of a "normal channel" which simply provides an interface between the client and the primary server. Case ② by means of an "updating channel" operating among clients, primary and replica servers as shown in figure 3 (for additional details see section 3.3 below). Finally, case ③ by means of a transaction-like interaction among all participants. Thus we achieve the separation of concerns.
This is possible since in the channel reification model each object can be connected at meta-level to more than one object. In the metaobject model, on the contrary, each object is connected with only one meta object, and all object interactions mechanisms have to be included in such a metaobject, increasing its complexity and reducing its flexibility.
Channel reification is better suited to fault detection and recovery than other models for reflection. Recovery from site crashes is supported by channel stubs connecting the client to the server. When one site crashes the stub on the other site establishes a temporary connection with a replica of the failed object. A replica

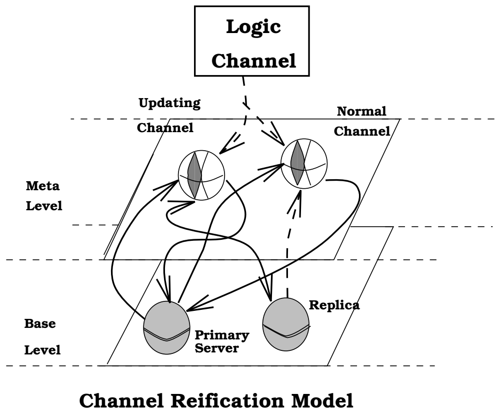


Figure 3: Channel Reification Passive Replication

of the failed stub is then reified on the corresponding site, and a new channel connecting the replica with the non-failed stub is established. Finally execution is restarted from the last stable state.

3.3 Implementing Different Kinds of Channels

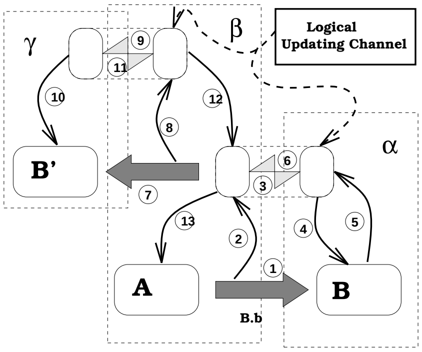
Assuming that we wish to implement cases ① and ② of the previous section, we will show in more details how two kinds of channel are implementable.
Assume that we have an application consisting of two objects **A** and **B**, respectively at site α and β. Object **B** is critical, so we keep its replica $\mathbf{B}'$ at site γ. **B** has two methods **b** and **c**; method **b** modifies the status of **B** while **c** does not. Changes performed by **b** must be applied also to $\mathbf{B}'$. In channel reification execution of these methods is handled by two channels of different kind, an updating channel for **b** that replicates all changes to replica $\mathbf{B}'$, and a normal channel for **c** which does not perform modifications. Both channels show a common behavior consisting of failure detection and recovery.
figure 4 shows how a request **B**.**b** from **A** is handled:

① At base level **A** asks to **B** to execute **b**.

② Request at point ① is trapped by the channel stub at the meta level.

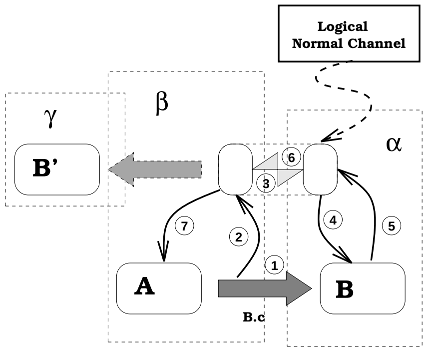
③ The stub communicates to the other stub, data and operation relative to the trapped call.

④ The second stub activates method **b**, by executing meta method "apply".



Updating Channel

Figure 4: **UpDating Channel Approach**



Normal Channel

Figure 5: **Normal Channel Approach**

⑤ At the end of the execution of **b**, "apply" returns the results of the execution of **B**.**b** to the stub.

⑥ The stub forwards data computed in step ⑤ to the other stub.

⑦ Since this logical channel is an updating one, the stub requires to update the status of **B**′.

⑧ The request (performed in ⑦) is trapped by the stub of the meta channel connecting the channel with the replica **B**′.

⑨ The stub communicates to the other stub the data and status of the trapped call.

⑩ The stub updates the status of the replica **B**′ (structural reflection possibilities).

⑪ After the update, a commit message is passed to the other stub.

⑫ The stub passes the commit message to the client.

⑬ The client stub, returns (to **A**) the result of activation of **B**.**b**.

Figure 5 represents how a request from **A** of **B**.**c** is handled.

①-⑥ As in the example of the updating channel (figure 4).

⑦ Method **c** does not perform status changes, then the status of **B**′ has not to be updated: the channel used is of "normal" kind and the stub returns to **A** the results of **B**.**c**.

Now let us assume that site α crashes. In both cases (i.e. updating or normal channel use), a fault detection and recovery action must be performed. Such behavior is the same for both channels (it could be inherited from some common superclass in the channel hierarchy).
The non-failed stub at β detects the failure of the stub at site α.
After fault detection, the stub at β authorizes the replica **B**′ to become the primary server. A new replica **B**″ is created and execution is restarted.
¿From now on existing channels are senseless, because they were established between entities that do not exist any more. So these channels are destroyed and replaced by other channels with the same behavior, which connect the new object to **A**.
Figure 6 shows how a request **B**.**c** from **A** is handled, after the original object **B** has failed.

① At base level, **A** asks a service to B.

② Request **B**.**c** is trapped by the channel stub at meta level (the one shown shaded in figure 6).

③ The shaded stub tries to communicate to the other stub; detecting its down status.

④ The shaded stub promotes the secondary replica **B**′ to become the primary one.

⑤ The shaded stub creates another secondary replica **B**″.

⑥ A new "normal" channel is established between **A** and **B**′ that starts the computation of **c**, the shaded stub provides to it information needed for completing the execution of **c**.

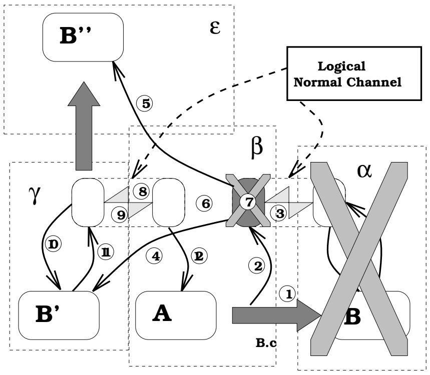


Figure 6: **Handling α crash**

⑦ The shaded stub is destroyed.

⑧-⑫ As in actions ③-⑦ of figure 5.

A request of **B**.**b** would be treated in a similar way.

4 Conclusion and Future Work

In this paper, a new reflective model, *channel reification*, has been introduced and examples have been provided, which should highlight the potentiality of this model in fault tolerant systems development. Channels are reifications of messages to be exchanged between two objects, which can survive and be reused by subsequent messages. The reflective behavior is identified by the *kind* of the channel: examples have illustrated how different requirements for fault tolerance are mapped to channels of different kind. Channel reification has been compared to other models for reflection, the metaobject and metacommunication models: we have shown how it can solve the drawbacks of both.
We have considered how to implement channel reification in order to experience on this new approach to reflection. Channel reification could be implemented on top of a distributed object system (e.g. CORBA) or else on top of a distributed system, like ISIS or PVM. Due to its wider diffusion, we have selected the latter as a basis for our implementation, taking into account the fact that its message passing features already implement many of the stubs functionalities.
We have first implemented a non distributed prototype of channel reification in SmallTalk, then, we have designed a C++ implementation on top of PVM. In all cases, the implementation requires only two procedures/methods, (`send`, and `apply`), and a *channel* class hierarchy. Implementation of channel reification in distributed object systems (like CORBA [Cor93]) is under analysis.

References

[Anc90] M. Ancona, A. Clematis, G. Dodero, E.B. Fernandez, V. Gianuzzi, "A System Architecture for Fault Tolerance in Concurrent Software", *IEEE Computer*, 23(10), pp.23-32, 1990.

[Anc95] M. Ancona, G. Dodero, V. Gianuzzi, A. Clematis, L. Lisboa, "Reflective architectures for reusable fault-tolerant software", *proc. 1st Ibero American Microelectronics Conf.*, Canela, Brazil, july 31, Aug.4, 1995.

[Caz95] W. Cazzola, "The Role of Reflective Architectures in the Development of Fault Tolerant Software.", DISI-University of Genoa, Master Thesis (in Italian) 1995.

[Chi93] S. Chiba, T. Masuda, " Design an Extendible Distributed Language with a Meta-Level Architecture", In Proceeding of 7th European Conference on Object-Oriented Programming (ECOOP 93), pages 482-501, Kaiserslautern Germany, July 1993.

[Cor93] OMG TC, "The Common Object Request Broker: Architecture and specification", Document Number 93.12.1, Revision 1.2, December 1993.

[Fab94] J. Fabre, V. Nicomette, T. Pérennou, Z. Wu, "Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming", PDCS2, 2nd year report, Predictably Dependable Computing Systems, Newcastle upon tyne, England, September 1994, pp 291-313.

[Fer89] J. Ferber, "Computational Reflection in Class Based Object Oriented Languages", *Proc. OOPSLA'89*, Sigplan Notices, pp. 317-326.

[Mae87] P. Maes, "Concepts and Experiments in Computational Reflection", *Proc. OOPSLA '87*.

[Yon87] A. Yonezawa et al., "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1", Object-Oriented Concurrent Programming, The MIT Press 1987.