mChaRM: a Reflective Middleware for Communication-Based Reflection

Walter Cazzola[1,2] and Massimo Ancona[1]

[1] DISI - Department of Informatics and Computer Science,
University of Genova, Genova, Italy
{ancona,cazzola}@disi.unige.it
[2] DISCO - Department of Informatics, Systems, and Communication,
University of Milano - Bicocca, Milano, Italy
cazzola@disco.unimib.it

Abstract There are several classes of features which cannot be modeled (or are hard to be modeled) within the currently available reflective models. One problem is represented by the lack of *global view* inherited from the object-oriented paradigm. This fact limits the potentialities of reflection, and reduces the application domain of the reflective paradigm especially when dealing with distributed programming. In the latter case, the more complex (often very hard to be treated) problems derive from communication mechanisms. In this paper we briefly examine the problems deriving when some enriching of the communication behavior is attempted via reflection. We also present a new reflective model, called the *multi-channel reification* approach, which extends some precedent proposal of the authors, and overcomes most of the limits of the actual reflective approaches. This model is especially designed both for designing and developing from scratch complex communication mechanisms or for extending the behavior of the communication mechanisms provided by the underlying system. To point out the soundness of our approach we also present mChaRM a reflective middleware, composed of a Java extension, an API, and a preprocessor, which realizes the approach described. Some applications realized with mChaRM are also presented.

Keywords: Reflection, Behavioral Reflection, Reification, Reflective Model, Reflective Programming, Communication Features, Middleware

1 Introduction

With the growing interest in distributed environments and systems, the specific requirements of those systems also grow:

- *naming and routing*, messages[1] exchanged between two objects[2] have to be routed from the source to the right destination;

[1] We use the term message following the SmallTalk parlance, representing both (remote) method calls and data-exchanges.
[2] In the following, we consider an object-oriented environment, so we use the specific term object instead of the more general term process.

- *security authentication*, only some objects can freely exchange messages, several times the communication among objects depends on the authentication of their rights;
- *communication reliability*, in most cases, messages exchange have to guarantee the satisfaction of some requirements, as delivery, delivery in the right order, delivery in transaction-like mode, and so on;
- *multi-communications*, some messages have to be dispatched to several destinations (broadcasting, multi-casting, and so on), or repeatedly to the same destination (polling, CRC checksum, and so on).

All these requirements (and many others) are related to communications, and, at the moment, their management is demanded either to the operating system (like in Unix, and in WindowsNT), or to some distributed programming environment running above it (the most known examples are CORBA [17], and ISIS [5]). Of course, this approach can satisfy in most situations, but there are applications requiring a higher degree of flexibility and reusability. In fact, as remarked in [22,27], distributed systems either include coordination code which is tightly *hardwired* into the application code or completely separated from it, and operating in a different execution space which is out of designer's control, thus hindering its reuse.

A reflective approach, as stated in [7], is the glue sticking together distributed and object-oriented programming and filling the gaps in their integration. Reflection improves flexibility, allows developers to provide their own solutions of communication problems, and keeps communication code separated from the application code, and completely encapsulated into the meta-level.

However, traditional reflective approaches (based on the meta-object paradigm) have been designed for handling different requirements (i.e., reflecting on the base-level objects and on ingoing messages), hence they result inadequate to fulfilling all nonfunctional requirements that are communications oriented, as those mentioned above. In order to render the impact of reflection on object-oriented distributed framework effective, and to obtain a complete separation of concern, we need new models and frameworks especially designed for communication-oriented reflection, i.e., a sort of reflective middleware suitable for enriching, manipulating and replacing the distributed communications and their semantics. That is, we need to encapsulate message exchanging into a single logical meta-object instead of scattering any relevant information related to it among several meta-objects and mimicking the real communication with one defined by the meta-programmer[3] among such meta-objects as it is done using traditional approaches.

In this paper, we face the reflective distributed computing area, determining the weak points of the existing approaches, and proposing a new model specially thought to overcome weakness of the traditional ones. In particular, in section 2 we briefly describe how reflective middleware for distributed computing works and their weak points we found; in section 3, we describe our solution, it consists in a new reflective model communication-oriented (§3.1), in an API for meta-programming with the proposed

[3] With the term meta-programmer we mean the person who designs and programs the program — also termed as meta-program — running into the meta-level.

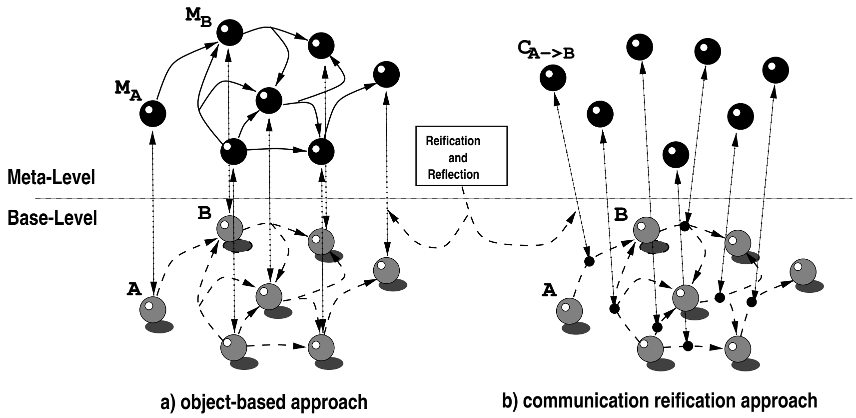

Figure 1. Base- and Meta-Level Communication Graphs.

model (§3.2), and in a preprocessor for an extension of Java supporting the approach (§3.3); in section 4 we show few simple, but not trivial, applications using the proposed approach. In section 5 we analyze the advantages and the properties of the proposed approach, underlying why to use it. In section 6 we compare our work with other reflective middlewares; and finally in section 7 we present some new ideas and the future work.

2 Reflection for Distributed Computing

Since distributed computing includes several aspects as well as application logic, a number of researchers stated that reflection enables separation of concerns [19,33] so that it can simplify software development for distributed computing (see [7]). Several reflective middlewares have been proposed, e.g., GARF [15], CodA [29], and recently OpenCORBA [23]. A crucial issue of designing reflective systems consists in choosing a good meta-model in their application domains. Most of the meta-models that have been presented so far are object-based models (such as the meta-object model [24,20]). In these models, every object is associated to a meta-object, which traps the messages sent to the object (method calls) and implements the behavior of that invocation. Most of the existing reflective middleware for distributed computing are based on these concepts.

However, these approaches are not appropriate to handle all the aspects of distributed computing. In particular adopting an object-based model to monitor distributed communications, the meta-programmer often has to duplicate the base-level communication graph into the meta-level (as depicted in Fig. 1.a) augmenting the meta-program complexity. By example, in order to write a meta-program which encrypts — using an algorithm based on public and private key idea, such as RSA [31] — each message exchanged between `A` and `B`, the meta-programmer has to write two distinct meta-objects $\mathtt{M_A}$ and $\mathtt{M_B}$ which they respectively have to encrypt, to decrypt and, of course, to exchange the message and the public key[4]. Hence, it is obvious that the encryption and decryption code — the meta-objects functional code — is intertwined with code needed to synchronize the meta-objects, to retrieve the public key from the sender and to exchange the encrypted message — the meta-objects nonfunctional code. Thus, object-based approaches to reflection on communications move the well-known problem [19], which the reflection try to remedy to, of nonfunctional code intertwined to functional one from the base- to the meta-level. Simulating a base-level communication into the meta-level, as advocated in [29,28], allows to perform meta-computations either related sending or receiving action, but not related to the whole communication or which involve information owned both by the sender and by the receiver without dirty tricks. This trouble goes under the name of *global view* lacking. The global view concept is defined in [2,8] as the situation in which the meta-computation can involve all base-entities and aspects involved in the computation which it reifies and as advocated in [13], reflection inherits the trouble of the global view lacking from the object-oriented methodology which encapsulates the computation orthogonally to the communication.

Besides, it is hard to filter the messages before they are delivered to the base-object since each ingoing message usually is delivered to the base-object then it is trapped by the meta-object. But several nonfunctional features need to be applied to the message after the message leaves the sender and before getting to the receiver in order to fulfill their commitments. Some of the nonfunctional features with such requirements consist of validating the authorization rights of each request forwarded to a server, as advocated in [3], balancing the computational load among several servers, and routing messages in a network through the fastest via. Using an object-based approach, in a situation as the one depicted in Fig. 1.a, a request forwarded by `A` to `B` could be dispatched and delivered through the standard communication layer, thus the request gets to the base-object `B` before that its meta-object $\mathtt{M_B}$ traps and validates it — in this way a new weak point is introduced in the security of the system — or the standard communication layer could be gone around putting directly in touch the two meta-objects $\mathtt{M_A}$ and $\mathtt{M_B}$ and only after the validation $\mathtt{M_B}$ will really carry the request out — in this wat we fall again in the case of nonfunctional code intertwined with functional one.

To avoid these problems, we argue that a meta-model based on communication is needed to design a reflective system for distributed computing. In fact, there have been a few researchers proposing similar ideas. For example, Ferber [14] proposed the message reification model. In that approach, instead of reifying an object, the meta-objects — termed also as meta-messages — reifies the message, as shown in Fig. 1.b. A similar approach easily permits to extend the semantics of the communication mechanism, but it has the drawback of saying nothing about the objects involved in the communication. Notwithstanding the message reification model results more suitable for enriching the semantics of communication than the object-based approaches, we still think that it is not the most appropriate for distributed computing, because each meta-message exists until the message, which it reifies, exists. Such a behavior hinders the meta-program to perform meta-computations which involve information elaborated in previous meta-computations. We call this phenomenon *lacks of information continuity*. Besides it can-

[4] We cannot hardwire the public key into a meta-object without losing in generality.

not permit to interact with the object involved in the communication, and it is basically designed to reify only point to point communications.

3 Multi-Channel Reification Approach: Model, API and Implementation

A lot of work has already been done about reification of communications [25,38]. ABCL-R, ACT/R, and their extensions are based on the well-known actors model [1]. Their approach reassembles the message reification approach with actors which incarnate messages exchanged and generate new actors for each new communication. Their approach is based on the concept of message and messages are the main entities carrying out the computation. By the way it results a good idea for extending systems with nonfunctional features communication oriented (see [26,27,32]). We extend their idea introducing the concept of *communication channel* in order to avoid the lacks of continuity and global view which characterize some of the previous works.

3.1 Multi-Channel Reification Model

What a Multi-Channel is. The *multi-channel reification model* is based on the idea of considering a method call as a message sent through a logical channel established among a set of objects requiring a service, and a set of objects providing such a service. We propose to reify this logical channel into a logical object called multi-channel, which monitors message exchange and enriches the underlying communication semantics with new features used for the specific communication performed. Each multi-channel can be viewed as an interface established among the requiring objects (in the sequel termed as *senders*) and the objects satisfying such a request (in the sequel termed as *receivers*)[5]. Each method call is trapped by the multi-channel when originated, then it is imbued with the new semantics, and finally it is delivered to the destinations.

The Kind Feature. Each multi-channel is devoted to enrich the behavior of the communication channel it embodies, such behavior is termed *kind*. Messages exchanged among the same groups of senders and receivers may be different and with different needs. For this reason, several multi-channels can be established among the same groups of senders and receivers, each of them implementing a different behavior and monitoring a different set of messages exchanged by the same referents. An example of this situation is a critical system to which we want to add a checkpointing mechanism of exchanged messages. In this specific case it is convenient to save information only about methods whose execution causes state changes, i.e. to partition messages into two sets: non-modifying and modifying messages, and to established two kinds of multi-channels between the same group of objects: one managing non-modifying

[5] In order to simply the exposition, we treat senders and receivers as separated entities, but this choice is not restrictive, a sender and a receiver can be identified without problems. In this case, the multi-channel will look like a meta-object able to reflect both on outgoing, and on ingoing messages.

messages, and the other handling the modifying messages with an enriched semantics defining the expected checkpointing mechanism.

How Multi-Channels are Looked-Up. Each multi-channel is characterized by its behavior, i.e., the communication semantics it defines (the kind), and by the set of referents playing the role of receivers. Because of the intrinsic dynamicity of the communication semantics[6], only receivers are relevant to characterize a multi-channel, senders hook themselves on different moment and requests. Thus each multi-channel is represented by:

$$\textbf{multi-channel} \equiv \textbf{(kind, receiver}_i\textbf{, } \ldots \textbf{, receiver}_n\textbf{)}$$

The logical identification of a multi-channel with the pair composed by its kind (i.e., behavior) and the set of its referents on receiver side, allows, when starts the communication, to determine univocally towards which multi-channel the message has to be hijacked.

When the Execution Flow Passes from the Base- to the Meta-Level and Vice Versa. Each reification is related to the communication channel established among the interacting objects, and it will take place each times a new channel is used, i.e., when a message, requiring a channel never used, is sent in the system. Each instantiated multi-channel is compliant with the requested features (its kind will satisfy the requested behavior and its referents will be the message receivers, and so on). Each message requiring an already used channel, is handled by the multi-channel which embodies that communication channel. When the multi-channel terminates its computation the execution control returns to the sender originating the meta-computation. On the basis of the implemented behavior, the multi-channel may also transfer the execution flow to one (or more) of its referents in order to execute some base-computations and achieve the expected result. Thus, when an object issues a request to another object, the exchanged message is transparently trapped and hijacked towards the multi-channel which reifies the communication channel it has to use, then the multi-channel delivers the message, compliant to its behavior, to the designed receivers, and returns their results.

Communication Loci in Which Carry Some Meta-Computations Out. As outlined in Fig. 2, a multi-point communication has a wide area of influence — the gray cone in

[6] All existing communication statements are asymmetric, in each statement we express explicitly only the receivers, e.g., in Java [4] we use the dot notation `o1.methodName()` where o_1 is the receiver of the message and the sender is implicitly represented by `this`, analogously in C++PVM [16] we create a stream (an instance of `cppvmSendStream` class) connected to the receivers and each message reaches any receiver connected to the stream used. In each of these statements we omit to specify the senders, this is compliant to the fact which they foresee only a sender at time, and that it can be determined by the context. However in different times a communication can involve more than one sender, e.g., in synchronization, but also in these cases the sender is determined from time to time by the context. Due to this behavior we state that the sender side changes dynamically.

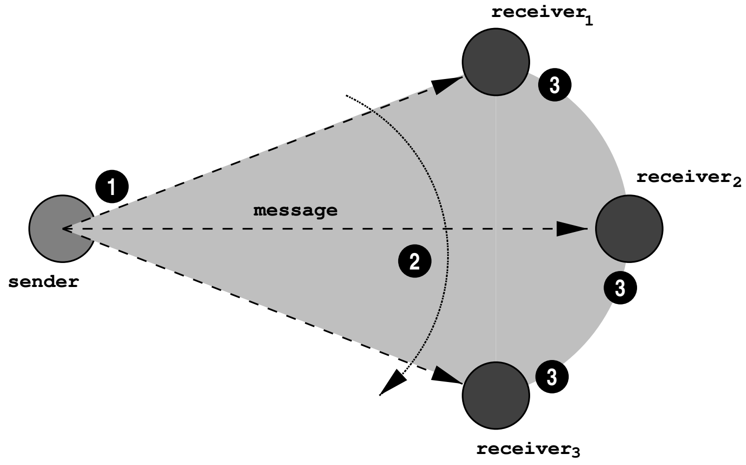

Figure 2. Aspects and loci involved in a multi-communication

the picture — and involves several aspects of the system. We define as *communication locus* a point of the way a message covers in order to get to the designed receivers, which realizes an important step of the communication mechanism. The most relevant loci (the numbers respect the situation depicted in Fig. 2) are:

❶ the sender and the beginning of the communication, termed *source locus*,
❷ the dispatching and the delivering of the message to the designed receivers, termed *abstract locus*, and
❸ the receivers and the carrying the computation really out, termed *target loci*.

These loci represent the whole communication and the entities (hidden or not) which work in each of them are put in charge of managing the basic aspects which allow to a distributed communication to be correctly carried out, e.g., in the source locus messages are marshaled. Each multi-channel incarnates these loci freeing the base-level from the communication mechanism moving it into the meta-level under the control of the meta-program. To allow the meta-programs to customize the semantics of each communication, the multi-channel is able to perform a different meta-computation for each locus substituting the standard behavior. Meta-computations performed on source and target loci guarantee that they are carried out on the site involved in the loci, e.g., on the sender site if the meta-program works on the source locus. This distinction derives from the requirements of performance, reliability and availability. Working on the source and target loci is convenient for reducing communications, while working on the abstract locus allows to decentralize meta-computations and to improve the reliability and availability of the provided services.

3.2 mChaRM and the API used by Multi-Channels.

We have slightly extended the Java language to a system called mChaRM — an acronym for multi-Channel Reification Model —, supporting the homonym model. In this section we describe the set of extensions needed by Java to support the model presented, and a simple API for meta-programming, designed to make the model suitable for implementing useful and reusable communication abstractions. We have chosen Java as language from which deriving our prototype, but the concepts described here can be easily used to extend any other object-oriented language.

Base-Level Language Enrichment. As said above, each message communication statement has to provide every information related to both the communication channel (the receivers of the message) and the behavior it needs to be fulfilled (the kind of the communication channel they intend to use), in order to determine the right multi-channel to be used. To provide this information we have extended the base-level language with constructs which create a link among each method-call, the receivers of such a call and the behavior that will be used for carrying such a call out. The introduction of such as information is coordinated by the new statement **kinds** .

Kinds. The **kinds** statement is used by the meta-programmer to associate a *kind* and a set of receivers, in agreement with the characterization of multi-channel given in section 3.1, to a given method activation. The information provided by the **kinds** statement will be manipulated by the compiler|interpreter and translated into statements to reify the related communication channel, to decouple the base-entities from the communication loci, and to route the method calls towards the right multi-channel. Of course, this translation is tied to the multi-channel architecture adopted, more on this will be explained in section 3.3. The **kinds** statement is described by the following syntax:

```
Kinds            ::= kinds KindsList
KindsList        ::= kind <kind-name> with ReceiverNameList to MethodNameList KindsList |
                     kind <kind-name> with ReceiverNameList to MethodNameList
ReceiverNameList ::= <receiver-name>, ReceiverNameList | <receiver-name>
MethodNameList   ::= <method-name>, MethodNameList | <method-name>
```

where keywords are written in non-proportional font, and *<kind-name>*, *<receiver-name>*, and *<method-name>* are identifiers representing, respectively, kinds, receivers, and methods identity.

The **kinds** statement has to be inserted in the class definition of each potential reflective sender. It identifies a section devoted to describe the interface of each instance towards the meta-level. A reflective sender, written in mChaRM, looks as:

```
class dummy
  kinds
    kind verbose with A, B, Z to dummy1
    kind normal with A, B, Z to dummy2, dummy3 {
      // usual class description
}
```

In the next we will indicate the kind of a multi-channel with the symbols ≪≫, e.g., in the stub of code above we associate a multi-channel of kind ≪`verbose`≫ with the calls of the method $\mathtt{dummy}_1$ carried out by instances of the class `dummy` to the receivers `A`, `B` and `Z`.

Multi-Channels' API. We propose a simple API, sufficient for supporting all multi-channel uses. Our API does not claim to be exhaustive, we kept it simple for presentation purposes. Despite its simplicity our API is enough powerful to be used for implementing sophisticated communication mechanisms and for showing the capabilities of the multi-channel approach. The prototype implements a more extended API that will be widened in the next future. Methods are classifiable, thanks to their purpose, into three categories: introspection, intercession, and meta-behavior.

We postpone the description of the classes containing the methods of the API, which depend on the multi-channel architecture adopted, to section 3.3 where the multi-channel structure and implementation are described.

Generic Multi-Channel Introspection
`senderStubInterface senderStub(String name)` *accessor to a representative of the specified sender.*
`receiverStubInterface receiverStub(String name)` *accessor to a representative of the specified receiver.*
`Object retrieveReceiverFieldValue(String receiverName, String fieldName)` *retrieves the content of a field of the specified receiver.*
`Object retrieveSenderFieldValue(String senderName, String fieldName)` *retrieves the content of a field of the specified sender.*
Senders' and Receivers' Side Introspection and Intercession
`Object referent()` *accessor to the referent of the stub.*
`Object retrieveField(String fieldName)` *queries for the contents of a specified field of the referent.*
`Object invoke(String methodName, Object[] args)` *invokes a specified method of the referent with the specified arguments.*

Table1. A selection of the API provided by mChaRM for carrying introspection and intercession out on senders and receivers.

Introspection and Intercession. The model proposed is designed to support the enrichment of communication semantics, not for managing the base-object structure or semantics; for this reason the part of the API devoted to intercession on multi-channel's referents is kept simple and consists of few methods, as shown in table 1. Besides, as explained in subsection 5.3, haphazard intercession might lead to an inconsistent state of the referents involved. Using the current API a multi-channel can only look into the state of its referents and invoke a method of one of its referents playing the role of receiver.

More exhaustive is the part of the API devoted to carry intercession and introspection out on messages dispatched through a multi-channel. These methods, table 2, together with the methods reported in table 3, represent the core of the whole mechanism allowing to alter the messages which pass through the multi-channel. The API supplies to the meta-programmer methods for looking into the contents of the actual parameters, for modifying their contents, for enriching with new parameters the method call we are handling, and so on. In section 4 we show how to use these methods to implement communication protocols.

Message Introspection and Intercession
`String getMethodName()` *retrieves the name of the method called.*
`String setMethodName(String methodName)` *through the name changes the method which will be really activated.*
`Object inspectArgument(int position)` *retrieves the value of a specified actual argument.*
`Object modifyArgument(int position, Object newValue)` *changes the values of a specified actual argument, and returns the old value.*
`void insertArgument(int position, Object value)` *inserts a new argument in the call.*
`Object removeArgument(int position)` *removes from the call a specified argument.*

Table2. A selection of the API provided by mChaRM for carrying introspection and intercession out on messages.

Meta-Behavior API
`Object coreMetaBehavior(mChaRMMethodCall msg)` *this method embodies the reflective behavior realized by the multi-channel.*
`void senderSideMetaBehavior(mChaRMMethodCall msg)` *embodies the meta-behavior performed by the multi-channel on the sender site.*
`void receiverSideMetaBehavior(mChaRMMethodCall msg)` *embodies the meta-behavior performed by the multi-channel on the receiver site.*

Table3. APIs provided by mChaRM for carrying meta-computation out.

Meta-Behavior. Methods belonging to this category implement the kind of multi-channel defining how multi-channels have to behave. The meta-programmer must override these methods in order to build new kinds of multi-channels which carry the meta-programmer's purposes out. As explained in section 3.1 there are three points of the communication flow, called loci, suitable for meta-level manipulation: source, target, and abstract loci.

Methods in table 3 define the meta-computation carried out by the multi-channel on the messages when they transit through each of the previously exposed loci. Their arguments represent the exchanged message and its components, i.e., the name of the method called, the value of the arguments passed to the method during the call, and the designed receivers of such a message.

The methods `senderSideMetaBehavior`, and `receiverSideMetaBehavior` are respectively performed on the source locus and on the target loci. They do not return values, however these methods allow the multi-channel to modify the exchanged message via side-effects performed on their actual arguments in agreement with the multi-channel behavior. Their default behavior is empty (i.e., they do nothing).

The method `coreMetaBehavior` implements the core of the multi-channel computation. It coordinates the result of the computations performed on the target loci and decides how to handle the received messages and to who demand their computation. As default behavior it forwards the messages to the specified receivers and returns the last value received to the caller.

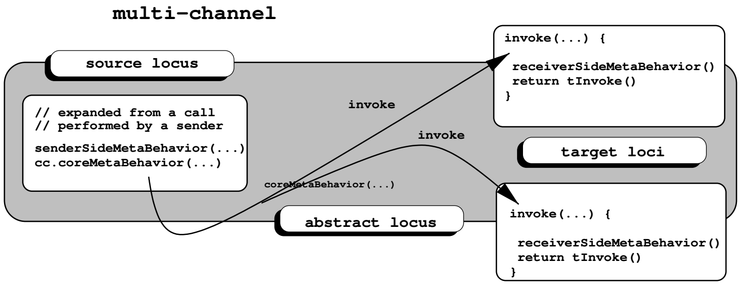

Figure 3. Following a method call through the meta-level.

The meta-program is basically composed of these three methods. In the redefinition of each of these methods, the meta-programmers can use whatever method of the described API. These methods are implicitly activated by meta-computation when messages flow through the meta-level. As depicted in Fig. 3, `senderSideMetaBehavior` is called when the message is trapped and hijacked to the meta-level in the source locus, followed by a `coreMetaBehavior` call[7]. The `receiverSideMetaBehavior` is executed on the receiver site, and it is activated each time that the meta-program calls the method `invoke`.

Note that the names of the receivers have to be explicitly passed to the meta-methods, instead of using the information hardwired into the multi-channel, because they offer to the meta-programmer the flexibility of specifying the receivers that have to be involved by the meta-computation instead of involving each receiver attached to the multi-channel.

The following example shows how the behavior of a multi-channel is described: the stub of code below realizes a multi-channel of kind ≪`verbose`≫. This kind of multi-channel is designed to output some notes while the message travels through the meta-level loci, i.e., they notify when a message leaves the sender, when it goes through the multi-channel computation and when it gets to the designed receivers.

```
public Object coreMetaBehavior(mChaRMMethodCall msg) {
   System.out.print("*** ");
   msg.printmChaRMMethodCall();
   return super.coreMetaBehavior(msg);
}

public void senderSideMetaBehavior(mChaRMMethodCall msg) {
   System.out.println("*** I'm inside the senderSideMetaBehavior ["+retrieveField("whoAmI")+"],");
   System.out.print("*** I'm working for "+msg.getMethodName()+"( ");
   if (msg.hasArgs())
      for(int i=0;i<(msg.actualArguments()).length;i++)
         System.out.print((msg.actualArguments())[i]+" ");
   System.out.print(")\n*** when I exit I'll forward the call to: "+(msg.receivers())[0]);
   for(int i=1;i<(msg.receivers()).length;i++) System.out.print(", "+(msg.receivers())[i]);
   System.out.println(".");
```

[7] We do not make supposition on where the `coreMetaBehavior` is performed, it should containing only statements whose execution is location-independent. For statements location-dependent the meta-programmer has to override either the `senderSideMetaBehavior` or the `receiverSideMetaBehavior` routines (or both).

```
}

public void receiverSideMetaBehavior(mChaRMMethodCall msg) {
   System.out.println("*** I'm in the stub of the receiver ["+retrieveField("whoAmI")+"],");
   System.out.print("*** the call trapped is: "+msg.getMethodName()+"( ");
   if (msg.hasArgs())
      for(int i=0;i<(msg.actualArguments()).length;i++) System.out.print((msg.actualArguments())[i]+" ");
   System.out.println(").");
}
```

3.3 mChaRM: architecture and implementation.

The multi-channel approach and the language extension described have been realized by the mChaRM tool. Such a tool is developed in Java and consists of three aspects:

❶ a preprocessor which handles the extensions to the Java syntax,
❷ a Java package (`mChaRM.multichannel`) containing the basic classes needed to develop multi-channels, and
❸ a collection of multi-channels (package `mChaRM.mChaRMCollection`) implementing some examples.

The preprocessor has been realized using OpenJava [11,36]. We have written a meta-object which leads the OpenJava compiler (`ojc`) during the translation of the mChaRM code into Java. It translates the **kinds** adapting the generated code to support the reflective approach described. The compiling procedure of mChaRM source code looks as in Fig. 4.

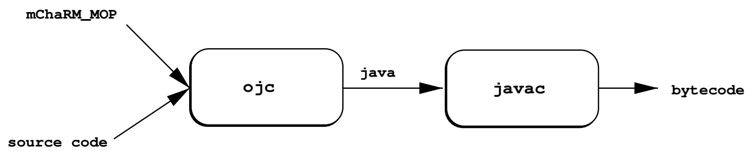

Figure 4. Compiling process.

As already explained the multi-channel approach enriches multi-point communications with new features. Java doesn't support multi-point communications, such as broadcast, multicast, or multi-rmi. Hence mChaRM tool adopts its own multi-point communication mechanism. The current version uses a naive realization of multi-point RMI based on Java RMI [35], which broadcasts a specified method call to a set of servers and then gathers the results and returns the first value received to the caller. A multi-point communication can be raised using the method `multiRMI`:

```
Object multiRMI(String methodName, String[] recsName, Object[] args)
```

At the moment this method is introduced in the client code by the mChaRM preprocessor and hides the reification|reflection mechanism provided by mChaRM, however

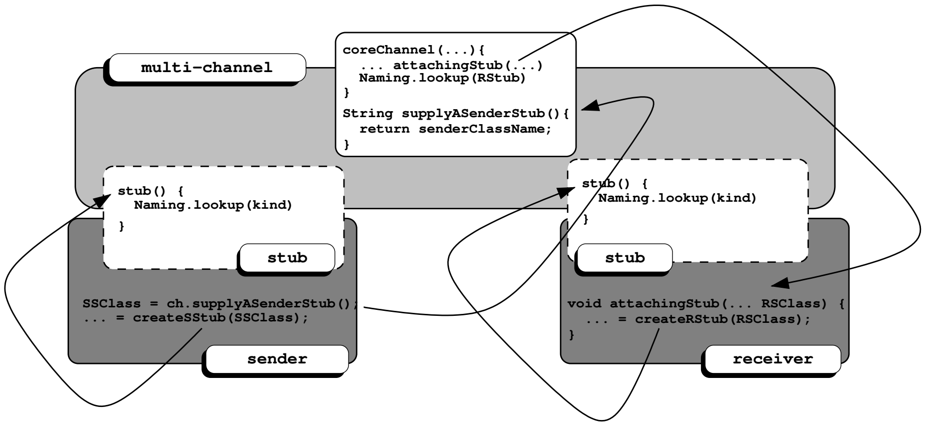

Figure 5. Building a multi-channel up.

we are already working on decoupling it from the mChaRM tool improving the transparency of the reflective mechanism. We have also kept its implementation simple because the main purpose of mChaRM consists of rendering the multi-communication protocol customizable by the (meta-)programmer. For exposition reasons we use the `multiRMI` also for the normal point-to-point RMI.

In the sequel of this section we describe the architecture that we have chosen for the multi-channels and how it has been realized. We also describe OpenJava, the meta-object we have developed in order to translate the mChaRM code into Java, and how the translation takes place.

Multi-Channel Structure. Both to achieve a good ratio among performance, transparency, reliability, and availability and to comply with the model requirements, we have chosen to design and implement the multi-channel as a distributed entity, composed of a central kernel (termed *core*) and as many stubs as its referents. Each stub is an object associated to a base-level object running on the same site and in the same addressing space. It is designed to interface that object to the multi-channel core, and vice versa. A sender stub transparently traps each call appointed to the multi-channel, of which it is part. Sender's stubs reify the source locus, receivers' stubs reify the target loci, while the core encapsulates the abstract locus functionality. Each receiver stub has the task to apply the `invoke`s issued by the multi-channel towards its own receiver. Many stubs can be attached to each base-level object, a stub for each multi-channel connected to that object. Stubs also execute the meta-behavior, developed by the meta-programmer, for source and target loci. The related stub classes define the corresponding API methods (see table 3), i.e., `senderSideMetaBehavior`, and `receiverSideMetaBehavior`.

Both senders' and receivers' stubs are attached to the core of the multi-channel dynamically. Figure 5 sketches the algorithm for building up a multi-channel and the connections among its components. Following its characterization (see the model definition in section 3.1) the multi-channel knows which of its referents are playing the role of receiver. During its initialization it notifies them through their method `attachingStub` of creating a stub that will be part of the multi-channel. The keyword **kinds** provides to the preprocessor all information about which multi-channels the sender could use during its lifecycle. The preprocessor expands the keyword **kinds** to the code for asking a stub to the core of the multi-channel (a remote method invocation of the core method `supplyASenderStub`) and for really creating it. In order to minimize the amount of exchanged data the core of the multi-channel only supplies the name of the stubs' classes to senders and receivers. Then they use the methods offered by the Java core reflection library [34] and such a piece of information to really create a stub.

Each stub can be viewed as an extension of its own referent, it communicates with its referent via a local method call, avoiding problems of communication reliability, and network unreachability. It looks as if sending and receiving action was moved to the meta-level layer where they are played with a new semantics, the one described by the multi-channel. In this way every communication algorithm can be developed without bothering the system functional behavior.

The multi-channel core does not reside in a specific location with respect to its referents, it is independent from its execution site which usually depends on the implemented behavior (e.g., if it implements a fault tolerant behavior it will be located on a faultless machine or replicated on several machines). It communicates with the stubs via remote method calls. All messages exchanged among multi-channel components are encapsulated in the API supplied to the meta-programmer.

This architecture guarantees a complete encapsulation of each aspect and locus involved by the communication. Thus, every communication protocol can be replaced by a new one encoded into a multi-channel. Each message communication is handled as follows[8]:

❶ with respect to the kind specified in the call, all data about the call are passed to the related sender's stub;
❷ the stub performs the `senderStubMetaBehavior`, then
❸ it calls the `coreMetaBehavior` of its multi-channel (this call is a remote call);
❹ after the core computation, which can involve zero or more `invokes`, it returns the result to the sender stub
❺ the sender stub returns the result to its referent, as the required call should have done.

Note that `invoke` calls the `receiverStubMetaBehavior`, before really applying the passed message.

Supporting Framework. The framework we implemented supplies some basic classes (`senderStub`, `receiverStub`, and `channelCore`) contained into the package `mChaRM.multichannel` from which it is possible to derive other stubs' and core's classes and therefore new kinds of multi-channels. In the sequel we examine their implementation.

[8] This Algorithm fills the gaps left in Fig. 3 and in the description given in the previous section explaining how the multi-channel hooks to the base-level.

Stubs. Stubs are the basic components of a multi-channel. Senders' and receivers' stubs inherit by a common class `stub` which keeps information about multi-channel identity (i.e., the core identity, the stub name, the name of its referent, and so on), and methods for handling this information. Most of the task is performed by the constructors of the stubs. In both cases, stubs communicate with the core through the Java RMI mechanism. They play the server role because they will receive messages by the core (e.g., for retrieving the value of its referent fields, or for activating the `invoke`). Senders stubs also play the client role because their main behavior consists in asking the core of the multi-channel to perform its part of the meta-computation. As shown by the following code both kinds of stubs register themselves as servers and server stubs also require an interface in order to communicate with the core of the multi-channel.

```
public senderStub(Object myReferent, String myKind, String myReferentName) {
   setReferent(myReferent); setWhoIsMyCore(myKind);
   Naming.rebind("__"+myReferentName+myKind, this);
   WhoIsMyCore().senderStubHasBeenCreated(myReferentName);
}
```

```
public receiverStub(Object myReferent, String myKind, String myReferentName) {
   setReferent(myReferent); setWhoIsMyCore(myKind);
   Naming.rebind(myKind+"__"+myReferentName, this);
}
```

Of course, the structure and the methods defined for each kind of stub change accordingly to the application they are designed for. The main job performed by a sender stub consists in forwarding the (reworking of the) original call to the core of the multi-channel. The caller implicitly invokes the `stubBehavior` method of the sender stub of the multi-channel it is using. `stubBehavior` performs a call to `senderSideMetaBehavior` and then it passes the reworked information about the original call, to its multi-channel core.

```
final public Object stubBehavior(mChaRMMethodCall msg) {
   senderSideMetaBehavior(msg);
   return WhoIsMyCore().coreMetaBehavior(msg);
}
```

Both `stubBehavior` and `senderSideMetaBehavior` cannot be called directly from the meta-program. The `senderSideMetaBehavior` can be overridden in descendant classes of class `senderStub` in order to build new kind of multi-channels.

Instead the main service offered by each receiver stub is the method `invoke`. Its behavior consists in forcing the receiver to activate the message filtered by the multi-channel. Implicitly the `invoke` performs the `receiverSideMetaBehavior`, and then it calls `tInvoke` which really activates the reworked message.

```
final public Object invoke(mChaRMMethodCall msg) {
   receiverSideMetaBehavior(msg);
   return tInvoke(msg);
}
```

The method `tInvoke` uses the Java core reflection library [34]. The algorithm looks into the receiver method table for a method matching the received message and, if found, invokes it with the received arguments and returns to the caller the achieved result. Method `tInvoke` cannot be overridden nor directly activated from the code of the meta-program.

```
final private Class[] extractArgsClasses(Object[] args) {
   if (args == null) return null;
   Class[] c = new Class[args.length];
```

```
    for (int i=0; i<args.length; i++) c[i] = args[i].getClass();
    return c;
  }

  final protected Object tInvoke(mChaRMMethodCall msg) {
    Class[ ] c = extractArgsClasses(msg.actualArguments());
    Method method = (referent().getClass()).getDeclaredMethod(msg.getMethodName(), c);
    return method.invoke(referent(), msg.actualArguments());
  }
```

Core. The core of the multi-channel is described by a class named: `channelCore`. The class constructor requires the kind of the multi-channel, a string identifying the behavior that its instances realize, and the identifiers of the known referents of the multi-channel it has to create. The constructor hooks the just created instance with the stubs of the specified referents, as well as to initialize the multi-channel structures.

```
  public channelCore(String kind, String[ ] RsName, String SSClassName, String RSClassName) {
    receivers = new Hashtable(); senders = new Hashtable();
    setReceiverStubClassName(RSClassName); setSenderStubClassName(SSClassName);
    setRsName(RsName); setKind(kind); sendersName = new Vector(10, 1);
    serverName = "__"+kind()+"{ ";
    for (int i=0; i<howManyReceivers(); i++) serverName += receiverName(i) + " ";
    serverName += "}";
    Naming.rebind(serverName, this);
    reflectiveReceiverInterface RSI;
    for (int i=0; i<howManyReceivers(); i++) {
      RSI = (reflectiveReceiverInterface)Naming.lookup(receiverName(i));
      RSI.attachingStub(serverName, receiverStubClassName(), receiverName(i));
      setReceiverStub(receiverName(i), (receiverStubInterface)Naming.lookup(serverName+"__"+receiverName(i)));
    }
  }
```

It is possible to define new kinds of multi-channels by extending the class `channelCore`. The behavior of the multi-channel is determined by the method `coreMetaBehavior`. A new kind of multi-channels can be created by writing a subclass of `channelCore` which overrides such a method. The behavior supplied by `coreMetaBehavior` in the framework, simply, consists in forwarding the trapped message to the specified receiver stubs for its activation, and in dispatching the result of the computation of the last receiver back to the caller. The multi-channel provided as default by the framework realizes a context switch between base- and meta-level and vice versa, and a multi-point RMI as really expected by using the `multiRMI` without enhancing its behavior. By the way it is a good starting point for derive new communication behaviors (as explained).

```
  public Object coreMetaBehavior(mChaRMMethodCall msg) {
    Object[ ] result = new Object[(msg.receivers()).length];
    for(int i=0; i<(msg.receivers()).length; i++)
      result[i] = ((receiverStubInterface)receiverStub((msg.receivers())[i])).invoke(msg);
    return result[0];
  }
```

OpenJava. OpenJava [11,36] is a compile-time MOP for Java that inherits most of the design philosophy of its direct ancestor OpenC++ Version 2 [10]. It can be seen as an "advanced macro processor" [11] that performs a source-to-source translation of a set of classes written in a possibly extended version of Java into a set of classes written in standard Java.

The translation to be applied to a base class is described in a metaclass associated, via the **instantiates** keyword, with the base class. The metaclass is written in standard Java using a class library that extends the Java reflection API with classes reflecting language constructs such as assignments, conditional expressions, field accesses, method calls, variables, type castings, and so on. As a result, writing a translation is quite easy and natural because of the object-oriented design of the library, which contrasts with the approach taken in OpenC++ where the sole abstraction available to the meta-level programmer is bare abstract syntax trees.

The mChaRM Preprocessor: `mChaRM_MOP`. As shown in Fig. 4 the code written in mChaRM is translated into standard Java through a preprocessing phase lead by an OpenJava meta-object instance of the class: `mChaRM_MOP`. This meta-object is charged both of expanding the new keyword, and of adding all the needed stuff for supporting the approach (the method `multiRMI`, the hooks for he stubs and so on). As shown by the code below, this meta-object, during the parsing of the source code, will add information for keeping trace of the object stubs, and the code used for creating and hooking the stubs. It has also to expand the **kinds** to the code for routing the messages through the right multi-channel.

```
public void translateDefinition() throws MOPException, CannotAlterException {
  ParseTree suffix = this.getSuffix("kinds");
  if (suffix != null) {
    addField(createAFieldForAnHashTableOfStubs("HashTableForSStub"));
    addCodeForSendersStuff(suffix);
    addMethodMultiRMI();
  }
  addField(createAFieldForAnHashTableOfStubs("HashTableForRStub"));
  addMethodAttachingStub();
  addInterface(OJClass.forName("mChaRM.multichannel.reflectiveReceiverInterface"));
}
```

When the meta-object detects the keyword **kinds** it knows that the instances of such a class will play also the role of sender, hence it renders available at run-time the mapping among multi-channel's kinds, receivers and methods through a static method, `initialize`, which fills an hashtable (`__kinds`), indexed on the messages and its potential receivers, with the kind of the multi-channels which have to be used for handling such messages. The information kept by such a hashtable are used by the method `multiRMI` for picking the right multi-channel up. The new method is executed during the object initialization phase.

```
private void addCodeForSendersStuff(ParseTree pt) {
  String initializeCode ="";
  OJMethod initialize = new OJMethod(this, OJModifier.forModifier(PRIVATE+STATIC),
                              OJClass.forName("void"), "__initialize", null, null, null, null);
    // add a static hashtable for handling the dynamic binding with channels
  TypeName tn = new TypeName("java.util.Hashtable");
  VariableInitializer vi = new AllocationExpression(tn, new ExpressionList());
  FieldDeclaration fd = new FieldDeclaration(new ModifierList(PRIVATE+STATIC), tn, "__kinds", vi);
  addField(new OJField(getEnvironment(), this, fd));

  Enumeration e = ((openjava.ptree.List)pt).elements();
  while (e.hasMoreElements()) {
    // parsing of each row of the section introduced by kinds
```

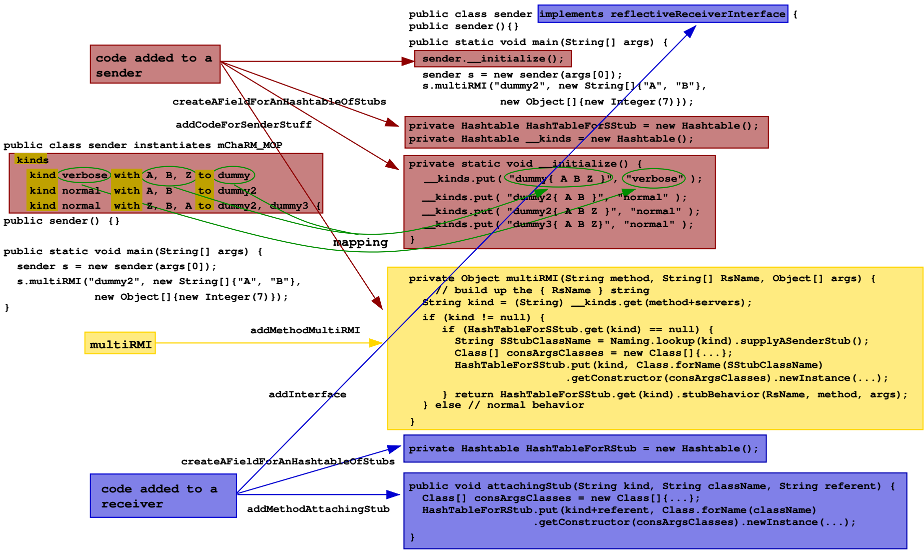

Figure 6. A trivial example of translation carried out by the preprocessor

```
      // building a mapping receivers-method and multi-channel's kind
      // used for routing a message through the designed multi-channel.
                      ...
    }
    initialize.setBody(makeStatementList(initialize.getEnvironment(), initializeCode));
    addMethod(initialize);                // initialize represents the map built above
                      ...
  }
```

In this case it also adds the method `multiRMI` to such a class. The method `multiRMI` has to deliver the message it takes to the designed receivers. It starts verifying in the maps `__kinds`, introduced by the preprocessor, if a multi-channel has been foreseen to reify such a message; If yes it demands the delivery of the message to the multi-channel and it activates the related stub which will begin the meta-computation.

```
private void addMethodMultiRMI() throws CannotAlterException, MOPException {
  OJMethod MRMI = new OJMethod(this, OJModifier.forModifier(PRIVATE), OJClass.forName("Object"),
                    "multiRMI", new OJClass[]{OJClass.forName("String"), OJClass.forName("String[]"),
                    OJClass.forName("Object[]")}, new String[]{"methodName", "RsName", "args"}, null, null);
  String MRMIcode = "String servers = \"{ \";";
      // builds the body of the method multiRMI and stores it into MRMIcode
                      ...
  MRMIcode += "return null;}";
  MRMI.setBody(makeStatementList(MRMI.getEnvironment(), MRMIcode));
  addMethod(MRMI);
}
```

Moreover the preprocessor will add all the stuff for handling the case in which the instances of such a class will play the role of receiver. In this case it basically adds a method which when triggered will hook a receiver stub instantiated from a designed class to the current object. As outlined in Fig. 5 this method will be triggered by the core of the multi-channel.

```
private void addMethodAttachingStub() throws MOPException, CannotAlterException {
  String ASCode; OJMethod AS;
  OJClass stringClass = OJClass.forName("java.lang.String");
  AS = new OJMethod(this, OJModifier.forModifier(PUBLIC), OJClass.forName("void"), "attachingStub",
         new OJClass[]{stringClass, stringClass, stringClass},
         new String[]{"kind", "className", "referentName"}, null, null);
    // puts in ASCode the body of a method which when called will attach
    // a receiver stub to the current object
  AS.setBody(makeStatementList(AS.getEnvironment(), ASCode));
  addMethod(AS);
}
```

In Fig. 6 it is shown an example of the translation that the preprocessor carries out. It is only a little stub of code, but it illustrates well how a class written using mChaRM (the code on the left side of the picture) is transformed in a class written in standard Java (the code on the right side of the picture).

4 Multi-Channel Applications

In this section we describe some multi-channel applications. All the examples show stub of code written using the API and the syntax previously described. Application overview of this section does not claim to be exhaustive, it only tries to explain some details of the multi-channel reification approach by simple examples.

Authentication Policies. One of the possible communication extensions consists in validating if a message can be delivered or not to a server. We can consider each message exchanged between a sender and a receiver as a service request forwarded from the sender to such a receiver. Thus a multi-channel established between two communicating objects, plays the role of judge, verifying if the sender has the permission for requesting or less such a service. As stated in [3] associating the verification phase to the communication whereas to the receiver, hinders malicious attacks. The code reassembles the ATM example presented in [3], with customers and ATMs; customers can deposit in and withdraw from ATMs. Basically a multi-channel established between a customer and an ATM has to verify that only the owner can withdraw from an account. The class `ATM` simply defines the methods `withdraw`, and `deposit`.

```
public class ATM instantiates mChaRM_MOP
  public void deposit(String id, String accNum, int sum) {
    // code for depositing the <sum> on the <id>'s account.
  }
  public void withdraw(String id, String accNum, int sum) {
    // code for withdrawing the <sum> from the <id>'s account.
  }
}
```

The class `customer` results more interesting because it shows how it is possible to use the kind mechanism to monitor only a subset of all the provided services.

```
public class customer instantiates mChaRM_MOP
  kinds
```

```
    kind validation with ATM1 to withdraw {

  public static void main(String[] args) {
    customer c = new client("Walter");
    c.multiRMI("deposit", new String[]{"ATM1"}, new Object[]{"Walter", "#1237", new Integer(1000)});
    c.multiRMI("withdraw", new String[]{"ATM1"}, new Object[]{"Walter", "#1237", new Integer(2000)});
  }
}
```

In our example we stated that a deposit can be done by everyone, whereas a withdrawal can be performed only by the owner of the account. Hence a multi-channel of kind ≪`validation`≫, which only filters the method `withdraw` validating the related constraints, is enough for achieving the desired behavior.

```
class validationChannelCore extends channelCore {
  private Hashtable right;

  public validationChannelCore(String[] RsName, String permissionFileName) {
      // initialize the right table
         ...
    super("validation", RsName, "senderStub", "receiverStub");
  }

  public Object coreMetaBehavior(mChaRMMethodCall msg) {
    bool permission = right.get(key(client, receiversName, methodName));
    if (!permission) thrown IllegallRequestForAService;
    return super.coreMetaBehavior(msg);
  }
}
```

The constructor of the core of the multi-channel has to glue together all the information about the logic multi-channel calling the constructor of its ancestor (the class `channelCore`). It specifies which are the classes of the multi-channel's components (the third and the forth arguments represent respectively the name of the classes of the senders' and receivers' stubs), and the multi-channel's kind: ≪`validation`≫ (through the first argument of the constructor of the class `channelCore`). The `coreMetaBehavior` simply carries the validation phase out.

Reliable Multicast Protocol. As explained, one of the main jobs of the multi-channel approach consists in realizing complex communication behavior transparently to the application. As an example we give an outline of the code of a multi-channel realizing the *reliable multicast protocol* [30].

Reliable Multicast Protocol (RMP) is a reliable multicast transport protocol offering a variety of services of different quality. Each sender can sequence their packets separately, or RMP can sequence all the packets sent to group. RMP offers also a feature called total ordering, which means that RMP makes sure that every member of the group has received the packets before passing them to application.

RMP does not require a special multicast server or group manager. It is based on a token rotating scheme and a mix of negative and positive acknowledgements. The token rotates in the whole recipient group and the token owner acknowledges every packet it has received while it holds the token. The token passes and acknowledges are sent usually as multicasts. When other group members discover that they are missing some packets, they can send negative acknowledges to the original sender.

Using a multi-channel, the RMP algorithm is shattered among the multi-channel components. There are two relevant typology of components involved in the RMP algorithm: the message sender and its receivers. The receivers stubs will play the receivers role, acknowledging each other the receipt of a messages, and waiting before really carrying the message out for the acknowledges from every of its siblings. When a stub receives an acknowledge for a message it has never received it has to ask the core of the multi-channel for it again. The class `RMPReceiverStub` which implements the RMP protocol on the target loci looks as follows:

```
class RMPReceiverStub extends receiverStub {
  private Hashtable acks = new Hashtable();

  public void pACK(int n) {
    if (acks.get(n)==null) channelCore.nACK(n, whoAmI);
    else acks.put(msgNumber, new Integer(((Integer)acks.get(msgNumber)).intValue()+1));
  }

  private void ackTheMsg(int n) {
    for(int i=1; i<howManyReceiverStubs(); i++)
      getTheNextReceiverStub().pACK(n);
  }

  public void receiverSideMetaBehavior(mChaRMMethodCall msg) {
    int msgNumber = ((Integer)msg.removeArgument(0)).intValue();
    acks.put(msgNumber, new Integer(1));
    ackTheMsg(msgNumber);
    while(((Integer)acks.get(n)).intValue()<howManyReceiverStubs()) ;
  }
}
```

Whereas, the channel core replaces the real sender in that role, it ignites the message propagation and when a stub loses a message the core provides to sending the missing message again, only to the interested stub. The class describing the core of the multi-channel looks like[9]:

```
class RMPChannelCore extends channelCore {
  private int msgNumber = 0;
  private Hashtable messages = new Hashtable();

  public RMPChannelCore(String[] RsName) {
    super("RMP", RsName, "senderStub", "RMPReceiverStub");
  }

  public void nACK(int n, String name) {
    ((receiverStubInterface)(stubReceivers.get(name))).invoke((mChaRMMethodCall)messages.get(n));
  }

  public Object coreMetaBehavior(mChaRMMethodCall msg) {
    messages.put(++msgNumber, msg));
    msg.insertArgument(0, new Integer(msgNumber));
    return super.coreMetaBehavior(msg);
  }
}
```

Hence we have implemented in a simple way a tight collaboration among core and receivers stubs, and we synchronize the final message execution. This example

[9] We do not explicitly discuss that, because irrelevant for the explanation, but in order to work as expected both the core and the receivers stub of the multi-channel implementing the reliable multicast protocol, have to be multi-threaded.

shows several potentiality of the proposed approach, e.g., how manipulate the messages — each message is piggybacked with a number in order to distinguish them —, how the multi-channel components cooperate, how to reflect on ingoing messages, and so on. This example well characterizes many problematic tied to reflect on ingoing messages, and that are hard to be handled by the traditional reflective approaches.

5 Multi-Channel Perspectives

This section is devoted to illustrate benefits, purposes, and drawbacks of the multi-channel approach. The multi-channel approach maintains all the typical software development advantages (i.e., separation of concerns, code reuse improvement, transparent extendibility, and so on) of any other reflective approach, filling some of the gaps that the standard approaches leave.

5.1 Multi-Channel Purposes

The multi-channel reification model is oriented to communications: its design principle is to support encapsulation and abstraction of communication mechanisms, and to permit a modular enrichment of the communication behavior. Communication can be easily extended to include new features like tracing mechanisms, reliability control, fault tolerant behavior and so on. It is also a valid framework for testing and simulating new communication protocols.

Once communication mechanisms have been encapsulated into multi-channels, the meta-programmer is able to concentrate his efforts on the enrichment of the behavior of the encapsulated mechanisms, by disregarding implementation details. He does not need to be aware of problems like: how to pass the information from a multi-channel component to another, when and how to synchronize messages coming from several senders, and so on. In this way, the development is simplified, and new behaviors can be prototyped by means of simple changes of either the multi-channel used or of some of its components.

As a consequence, multi-channels can be also used to introduce multi-communications in languages which do not support them. Multi-channels have some similarities with the concept of *object groups*: they can be seen as a *group demon* which dispatches messages coming from outside to the members of the group. Thus, a multi-channel is a special kind of group, in which the members are fixed a priori (i.e., statically), and members can receive messages coming only from outside. Moreover, messages follow a prefixed routing mechanism hardwired into the multi-channel behavior. For this reason, it becomes relatively simple to realize complex group communication abstractions by means of the multi-channel mechanism abstraction.

5.2 Multi-Channel Properties

Global View. As advocated in [13] the object-oriented methodologies (also including the object-oriented reflective methodologies), are inadequate to handle distributed communications, because they do not maintain a *global view* of the program state, i.e., the computation is encapsulated orthogonally to the communication. The information about communication are shattered among and encapsulated into several objects, hiding the design of communication mechanisms (e.g., the protocol, the code, and so on), and hindering their management, reuse, and extension, features requiring a global representation of the communication state. Moreover, the reflective approach requires that each communication be manipulated and monitored as a single logical entity. As advocated in [2], the meta-object based approaches fail on this, because they reify in a meta-entity only some aspects of the communication. These aspects can be collected together only through meta-entities cooperation, and the meta-programmer has to deal with all coordination problems such as replication management, control of information consistency, and so on.

The multi-channel approach shifts every communication aspects to the meta-level, by reifying them into a unique logical entity and the complete communication code is hardwired into the multi-channel structure, and any phase of a communication mechanism can be manipulated through the supplied API. That is, the multi-channel approach maintains, at any time, a global view of the ingoing communication.

Communications Become Open|Close Systems. The reflective mechanism proposed, shifts every communication aspects to the meta-level. Each object involved in the communication is composed by two parts, one is running in the base-level and is developed by the application programmer, the other is running in the meta-level — the sender or the receiver stub —, and is developed by the meta-programmer. The meta-level part abstracts all mechanisms for handling messages sent|received by|from each object, and exposes them to manipulation and redefinition by the meta-program. Multi-channels gather all information about messages passing through them, and monitor each communication step. Hence, each multi-channel transforms the logical channel it reifies and messages passed through it, in a *open|close* system with few or none interactions with the base-level entities or with meta-entities external to the multi-channel. Such a complete separation improves the reusability and changeability of each communication mechanism. A multi-channel is open because it can be refined via inheritance in each of its components, and close because it is usable as-it-is when instantiated from each of its available class hierarchy. Furthermore, as shown in [3], the abstraction supplied by the multi-channel mechanism prevents to deliver every unwanted messages to the designed receivers, instead of filtering them after their actual reception, and this increases abstraction and encapsulation of communication, still enforcing a global view of the communication state. Using multi-channels the meta-objects composition problems can be avoided, because each multi-channel is a close system, and encapsulates the communication, besides, the kind feature permits to differentiate meta-behaviors in a way orthogonal to the behaviors composition. Thus the meta-programmer has less needs to combine multiple meta-entities.

Granularity. We define as *granularity of reflection* [8] the smallest aspect (e.g., objects, methods, method calls and so on) of the base-system, that can be reified into two different meta-entities. The multi-channel approach permits to reify each communication channel used in the base-level with a different multi-channel. In this case the meta-entity both spans between several objects, reifying the channel they use for communicating, and filtering only specific messages patterns. Multi-channels have a twofold granularity. From the point of view of a single object the granularity is at message level, because different messages sent to the same object can be reified by different multi-channels. On the other end, by considering every object involved in a communication the granularity is at communication channel level, because multi-channels with different kinds can reify the same logic channel established among the same objects, imposing two different behaviors for managing the messages passing through them. A fine granularity improves the model flexibility at the cost of meta-entities proliferation.

5.3 Multi-Channel Troubles

The proposed approach permits to hook several multi-channels to the same referent (both to senders and to receivers), hence, as explained above we achieve a finer granularity than using an object-based approach, but we have also to deal with consistency troubles. In fact some of the multi-channels hooked to a base-object could decide to carry out intercession on the state of their referent, and without a careful synchronization of the accesses the state might become inconsistent or wrong (due to the well known *race-condition problem*). Due to this reason we have limited multi-channel's intercession on its referents, but in the future we are going to introduce a mechanism to implicitly lock and unlock the referents state and to improve the intercession mechanism of the approach.

Another potential trouble of this approach, or better of each reflective approach for realizing middlewares, is represented by performance issues. As explained each reflective mechanism is based on trapping a call and hijacking it to the meta-level, of course a similar approach introduces an extra overhead to each communication. Our approach doesn't trap every call but only those specified by the **kinds** section and the hook among method call and multi-channel is statically realized, limiting the extra overhead. At the moment we are quantifying the extent of such an overhead and the really improvement with respect to the other reflective approaches.

6 Related Work

At the moment, several research directions about reflective middlewares for supporting distributed communications are under development (see for example [21,23,12] the section about reflective middleware of [9] and the online proceedings of the workshop on reflective middleware [6]). At the moment the topic is quite hot and result quite impossible to consider all the proposed approaches. However most of the approaches are based on the meta-object model, and so, as stated before, they lacks both of a global view and of a strong communication abstraction. We will only briefly consider two well known approaches in this category: GARF, and CodA.

GARF [15,18] is a middleware which lays between the system layer (which supplies the distribution mechanisms) and the application layer. Its basic purpose consists in wrapping the distribution primitives, supplying a uniform and abstract interface to them. Hence, GARF does not implement the distribution environment but it masks each access to it and offers to the programmer a tool for changing and enriching the basic behavior, as our model does. The GARF model is derived by the meta-object one with some adaptation in order to handle communications. It links a meta-object, called *encapsulator*, to the referents, and two encapsulators communicate via another meta-object, called *mailer*. This architecture reassembles the one we adopted for the multi-channels, and supplies a valid communication customizability via a built-in library containing several mailers and encapsulators. As the authors state mailers can only control out-going messages, thus an object has to receive the message in order to decide of taking care of it or not.

CodA [29] is a reflective middleware introducing a mechanism for distributed programming. Distribution is entrusted to the meta-level, each object is reified by seven meta-objects; each of them manages a specific distribution aspect (e.g., messages queue, sending, and so on). This meta-level factorization simplifies the definition of new behaviors for specific distribution aspects, unfortunately it also jeopardizes the mechanism flexibility by selecting which aspects can be manipulated. The CodA architecture reassembles the multi-channels architecture, but meta-objects reify and reflect about how an object must manage a message, and not about the specific semantics of message management and related aspects. Hence, the CodA framework lacks of global view, the author suggest to achieve it by communications among meta-entities and their cooperation. Although it is flexible, CodA has a complicated meta-level with several and different meta-objects, and might bring unnecessary additional complexity for coding the meta-program distributed among as many meta-objects, and for composing the behavior of so many meta-objects.

The Harmony/2 language [37] is not a true reflective language, or better it is not presented as a reflective language, but it has several similarities with our work. It introduces the method continuation concept in the actor model [1]. Each exchanged message is reified into an object which takes care of achieving the delivery of the request and, if necessary, of enriching the predefined behavior. The Harmony/2 model can be considered as derived by the message reification model [14]. Each meta-object is created, manages the message which embodies and is destroyed, thus Harmony/2 suffers from a lack of *information continuity*, i.e., it is not possible — without using a mechanism for handling the meta-object in a persistent way — to perform meta-computations which need information computed in previous meta-computations.

Our previous work about reflective middleware — simply, termed *channel reification model* [2] —, proposes an approach derived by the message reification model [14]. It establishes several channels among a sender and a receiver, with the purpose of trapping and manipulating each message sent by that sender to that receiver. The approach overcomes the cited lacks of global view, and the lacks of information continuity of the message reification model. Unfortunately, it was designed for enriching point-to-point communication, so it is inadequate for managing multi-point communications, or better it can be used for handling multi-point communications but in that case several channels are involved in each communication, and the meta-programmer is charged of coding the communication among channels.

7 Conclusions and Future Works

The role of reflection, in monitoring object communication, in complex distributed applications, is a crucial issue. We presented a reflective framework — called multi-channel reification approach —, which abstracts and encapsulates inter-object communications and enables the meta-programmer to enrich and|or replace the predefined communication semantics. The multi-channel approach fills most of the gaps of the traditional reflective approaches. Its main features are a reflective mechanism operating on ingoing messages before the recipient really receives them, and an improved global view of communication without compromising encapsulation. The approach also offers a finer reification|reflection granularity than previous approaches, and a simplified approach to the development of communication-oriented software. The model proposed is realized by a test bed framework, called mChaRM, written in Java by means of the language extension capabilities of the OpenJava [36] reflective tool. The reusability of the approach is shown by prototyping some non-trivial applications [2,3]. A simple API for supporting a simple meta-program development is also provided. A beta version of the mChaRM, can be downloaded from `http://www.disi.unige.it/person/CazzolaW/mChaRM_webpage.html`.

Besides the API and implementation improvements, we are also studying a method for improving the transparency of the multi-channel architecture. In the current version, the meta-programmer knows that a multi-channel is divided in several components, and programs its behavior on the basis of this knowledge. The idea is to unify the multi-channel logical structure, allowing the meta-programmer to develop only a class, and entrusting the redistribution of the code in the correspondent multi-channel components to the preprocessor|compiler. Another improvement will consist in creating new kinds of multi-channels by means of components composition, e.g., by composing a future-based sender stub with a reliable multicast protocol, thus achieving a furthermore enhancement in the reusability and in the readability of the meta-level code.

Acknowledgements

The authors wish to thank Prof. Shigeru Chiba from University of Tsukuba both for his interest in this work and for his valuable contributions to the ideas presented in this paper.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, 1986.
2. Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: A Reflective Model for Distributed Computation. In *Proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, 98CH36191, pages 32–36, Phoenix, Arizona, USA, on 16th-18th February 1998. IEEE.
3. Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective Authorization Systems: Possibilities, Benefits and Drawbacks. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Lecture Notes in Computer Science 1603, pages 35–51. Springer-Verlag, July 1999.

4. Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachussetts, second edition, December 1997.
5. Kenneth P. Birman, Robert Cooper, Thomas A. Joseph, Keith Marzullo, Mesaac Makpoungu, Kenneth Kane, Frank Schmuck, and Mark D. Wood. The ISIS System Manual V2.0. Technical report, Cornell University's Computer Science Department, May 1990.
6. Gordon Blair and Roy Campbell. Proceedings of the Workshop on Reflective Middleware. Available on-line at `http://www.comp.lancs.ac.uk/computing/RM2000/`, April 2000.
7. Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.
8. Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.
9. Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors. *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, June 2000.
10. Shigeru Chiba. A Meta-Object Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notices*, pages 285–299, Austin, Texas, USA, October 1995. ACM.
11. Shigeru Chiba, Michiaki Tatsubori, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A Class-based Macro System for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 119–135. Springer-Verlag, Heidelberg, Germany, June 2000.
12. Jim Dowling, Tilman Schäfer, Vinny Cahill, Peter Haraszti, and Barry Redmond. Using Reflection to Support Dynamic Adaptation of System Software: A Case Study Driven Evaluation. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 171–190. Springer-Verlag, Heidelberg, Germany, June 2000.
13. Mohamed E. Fayad and Rachid Guerraoui. OO Distributed Programming Is Not Distributed OO Programming. *Communications of the ACM*, 42(4):101–104, April 1999.
14. Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of 4th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.
15. Benoît Garbinato, Rachid Guerraoui, and Karim R. Mazouni. Distributed Programming in GARF. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveil, editors, *Object-Based Distributed Programming*, LNCS 901, pages 1–32. Springer-Verlag, 1994.
16. Steffen Görzig. C++ Interface to PVM. Technical report, University of Stüttgart, Germany, 1999.
17. Object Management Group. The Common Object Request Broker: Architecture and Specification. formal-document 98-07-01 Revision 2.2, OMG, February 1998.
18. Rachid Guerraoui, Benoît Garbinato, and Karim Mazouni. GARF: A Tool for Programming Reliable Distributed Applications. *IEEE Concurrency*, 5(4), October-December 1997.
19. Walter Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.
20. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

21. Fabio Kon, Roy Campbell, and Manual Roman. Design and Implementation of Runtime Reflection in Communication Middleware: the DynamicTAO Case. In *Proceedings of ICDCS'99 Workshop on Middleware*, 1999.
22. Shinji Kono and Mario Tokoro. Parallel Reflection. Technical memo SCSL-TM-90-011, Sony CSL, June 1991.
23. Thomas Ledoux. OpenCorba: A Reflective Open Broker. In Pierre Cointe, editor, *Proceedings of the 2nd International Conference on Reflection'99*, LNCS 1616, pages 197–214, Saint-Malo, France, July 1999. Springer-Verlag.
24. Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.
25. Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-Oriented Concurrent Reflective Languages Can Be Implemented Efficiently. In Andreas Pæpcke, editor, *Proceedings of 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, volume 27(10) of *Sigplan Notices*, pages 127–144, Vancouver, British Columbia, Canada, October 1992. ACM.
26. Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. Implementing Parallel Language Constructs Using a Reflective Object-Oriented Language. In *Reflection Symposium'96)*, pages 79–91, April 1996.
27. Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In Pierre America, editor, *Proceedings of ECOOP'91*, pages 231–250, Switzerland, July 1991. Springer-Verlag.
28. Jeff McAffer. Meta-level Architecture Support for Distributed Objects. In *Proceedings of IWOOS'95*, 1995.
29. Jeff McAffer. Meta-Level Programming with CodA. In Walter Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, LNCS 952, pages 190–214. Springer-Verlag, 1995.
30. Todd Montgomery. RMP - Reliable Multicast Protocol version 2. Document Available from `http://research.ivv.nasa.gov/projects/RMP/index.html`, October 1996.
31. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
32. Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Proceedings of COORDINATION'99*, LNCS 1594, pages 211–226, Amsterdam, The Netherlands, April 1999. Springer-Verlag.
33. Robert J. Stroud and Zhixue Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter 3, pages 31–52. CRC Press, Inc., 2000 Corporate Blvd.,N.W., Boca Raton, Florida 33431, 1996.
34. SUN Microsystems. Java™ Core Reflection API and Specification. Technical report, SUN Microsystems, February 1997.
35. SUN Microsystems. Java™ Remote Method Invocation - Distributed Computing for Java. White paper, SUN Microsystems, 1998. Internet Publication - `http://www.sun.com`.
36. Michiaki Tatsubori. An Extension Mechanism for the Java Language. Master of engineering dissertation, Graduate School of Engineering, University of Tsukuba, University of Tsukuba, Ibaraki, Japan, February 1999.
37. Ken Wakita. First Class Continuation Facilities in Concurrent Programming Language Harmony/2. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Paral-*

lel Programming: Proceedings of the International Workshop TPPP'94, LNCS 907, pages 300–319. Springer-Verlag, Berlin, Heidelberg, November 1995.

38. Takuo Watanabe and Akinori Yonezawa. An Actor-Based Metalevel Architecture for Group-Wider Reflection. In J. W. de Bakker, W. P. de Rover, and G. Rozenberg, editors, *Fondations of Object-Oriented Languages*, pages 405–425. Springer-Verlag, 1990.