Distributed Proxy: Design Pattern and Framework

António Rito Silva, Francisco Assis Rosa, Teresa Gonçalves and Miguel Antunes
INESC/IST Technical University of Lisbon, Rua Alves Redol n^o 9, 1000 Lisboa, PORTUGAL
Tel: +351-1-3100287, Fax: +351-1-3145843
Rito.Silva@acm.org, {fjar, tsg, mjsca}@scallabis.inesc.pt

Abstract

Developing a distributed application is hard due to the complexity inherent to distributed communication. This paper proposes an incremental approach to allow a divide and conquer strategy. It presents a design pattern and a framework for distributed object communication. The proposed solution decouples distributed object communication from object specific functionalities. It further decouples logical communication from physical communication. The solution enforces an incremental development process and encapsulates the underlying distribution mechanisms. The paper uses a stage-based design description which allow design description at a different level of abstraction than code.

1 Introduction

Developing a design solution for distributed object communication is hard due to the complexity inherent to distributed communication. It is necessary to deal with the specificities of the underlying communication mechanisms, protocols and platforms.

Herein, we propose an incremental approach for this problem which allows the transparent introduction of distributed object communication. In a first step the application is enriched with logical distribution, which contains the distribution complexity in a non-distributed environment where debugging, testing and simulation is easier. This first step ignores the particularities of distributed communication mechanisms. In a second phase, the application is transparently enriched with physical distributed mechanisms.

For instance, consider a Meeting Scheduler System which supports the organization of meetings. Each meeting has a initiator and each of the participants define the dates for which they can not attend the meeting.

The Booch[1] class diagram in Figure 1 shows a fragment of the Meeting Scheduler System functionality design, ignoring distribution issues.

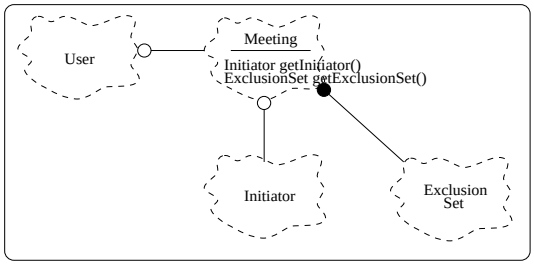

Figure 1: Meeting Functionality Design (fragment).

Enriching this design with distribution is complex. For example we must consider different address spaces: `Meeting` objects are shared entities and are deployed in a different site than `User` objects. So, the communication between `User` and `Meeting` is a distributed communication. Moreover, when users get the exclusion set of dates, method `getExclusionSet`, they would like to further interact with it, but the object reference they receive is meaningless in their address space. A different situation occurs when users get the initiator, method `getInitiator`, they would like to have a copy to reduce the number of distributed invocations during future interactions with `Initiator` object. Note that `Initiator` is a constant object, once created it will not change.

The rest of this paper is structured as follows. Next section presents a design pattern for distributed communication. Section 3 describes a three-layered framework which implements the design pattern. Re-

lated work is presented and discussed in section 4 and section 5 presents the conclusions.

2 Distributed Proxy Design Pattern

A design solution for distributed object communication is presented as a design pattern [2]. Design patterns describe the structure and behavior of a set of collaborating objects. They have become a popular format for describing design at a higher level of abstraction than code.

Contrarily to the common description of design patterns which emphasizes the pattern known uses, the description here presented emphasizes the quality aspects associated with the solution. To do so, a slightly different format is used: intent; problem; problem analysis; forces; structure; participants; collaborations; and forces resolution. Problem analysis describes concepts and solution approaches related to the problem. Forces define the qualities that should drive the design solution while forces resolution describes how the final solution fulfills the desired qualities.

2.1 Intent

The *Distributed Proxy pattern* decouples the communication between distributed objects by isolating distribution-specific issues from object functionality. Moreover, distributed communication is further decoupled into logical communication and physical communication parts.

2.2 Problem

Constructing a design solution for distributed object communication is hard due to the complexity inherent to distributed communication. It is necessary to deal with the specificities of the underlying communication mechanisms, protocols and platforms. Furthermore, distributed communication spans several nodes with different name spaces such that names may not have the same meaning in each of the nodes. In particular, invoking an object reference belonging to another node results in an error.

2.3 Problem Analysis

Usually, object-oriented distributed communication involves the definition of proxies which represent remote services in the client space and encapsulate the remote object [3]. This way, remote requests are locally answered by the proxy which is responsible for locating the remote object and for proceeding with invocation, sending arguments and receiving results.

Solutions to the problem usually involve several variations:

- **Arguments Passing** - arguments passing can have several semantics: (1) an object argument may be transparently passed between distributed sites, a proxy is created in the receiving site; (2) an object argument may be copied (deep copy); (3) an object argument is copied but, proxies are created for some of its internal references.
- **Location Time** - location of remote objects can either be at proxy creation time or at invocation time. The latter allows the remote objects to change its location.

2.4 Forces

The design solution for distributed object communication must resolve the following forces:

- **Encapsulation.** Distributed communication issues should be encapsulated from the functionality classes. Distributed communication should be transparent for functionality classes, the object-oriented interaction model should be preserved.
- **Extensibility.** The solution should be extensible to the variations described in the problem analysis section.
- **Modularity.** Distributed communication should be separated from application functionality. In particular, the underlying communication mechanisms should be isolated and it should be possible to provide different implementations.
- **Reusability.** The solution for distributed communication should be reusable in different situations.

2.5 Structure and Participants

The Booch class diagram in Figure 2 illustrates the distributed proxy structure. Three layers are considered: functionality, logical, and physical. Classes are involved in each layer. `Client` and `Server` at the functionality layer, `Client Proxy`, `Server Proxy` and `Reference Manager` at the logical

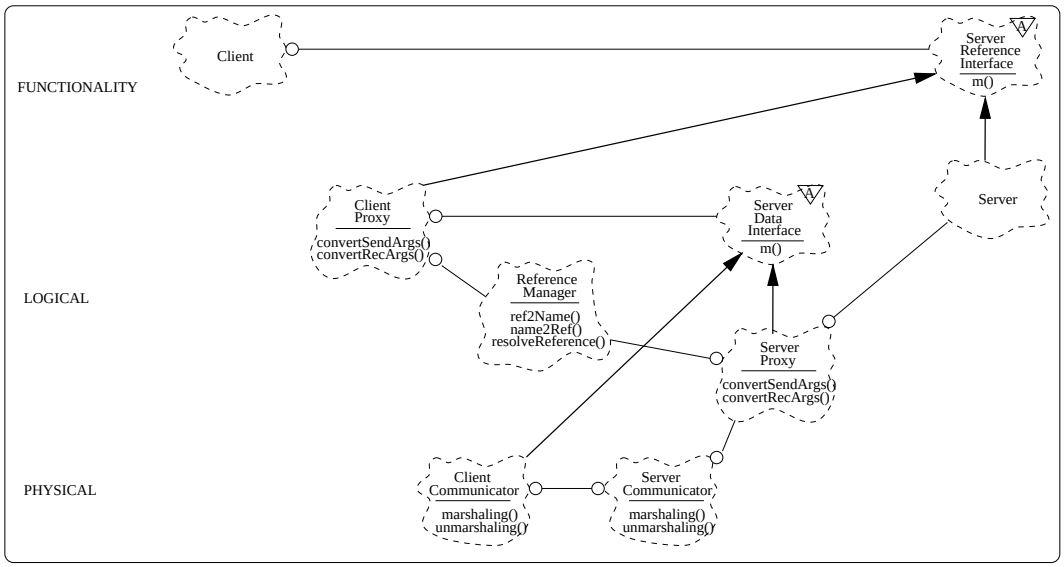

Figure 2: Distributed Proxy Structure.

layer, and `Client Communicator` and `Server Communicator` at the physical layer. Two abstract classes, `Server Reference Interface` and `Server Data Interface`, define interfaces which integrate functionality and logical layers, and logical and physical layers.

The pattern main participants are:

- `Client`. Requires a service from `Server`, it invokes one of its methods.
- `Server`. Provides services to `Client`.
- `Client Proxy`. It represents the `Server` in the client site. It is responsible for argument passing and remote object location. It uses the `Reference Manager` to convert sending object references to site independent names (distributed names) and received distributed names to object references. It also uses the `Reference Manager` to obtain a `Server Data Interface` where proceed with invocation.
- `Server Proxy`. It provides distribution support for the `Server` object in the server site. It is the entry point for remote requests to the `Server`. As `Client Proxy` it is responsible for supporting argument passing semantics.
- `Reference Manager`. It is responsible for associating object references, local and proxy, with distributed names and vice-versa. It creates new proxies, when necessary. Method `resolveReference` is responsible for returning to the `Client Proxy` a `Server Proxy`, when at the logical layer, or a `Client Communicator`, when at the physical layer.
- `Distributed Name`. It defines an identifier which is valid across sites. It is an opaque object provided from outside to the `Reference Manager`.
- `Client Communicator` and `Server Communicator`. They are responsible for implementing the distributed physical communication. For each called method, `marshaling` and `unmarshaling` methods are defined to convert arguments to streams of bytes and vice-versa.
- `Server Reference Interface`. Defines an interface common to `Server` and `Client Proxy`.
- `Server Data Interface`. Defines an interface common to `Server Proxy` and `Client Communicator`.

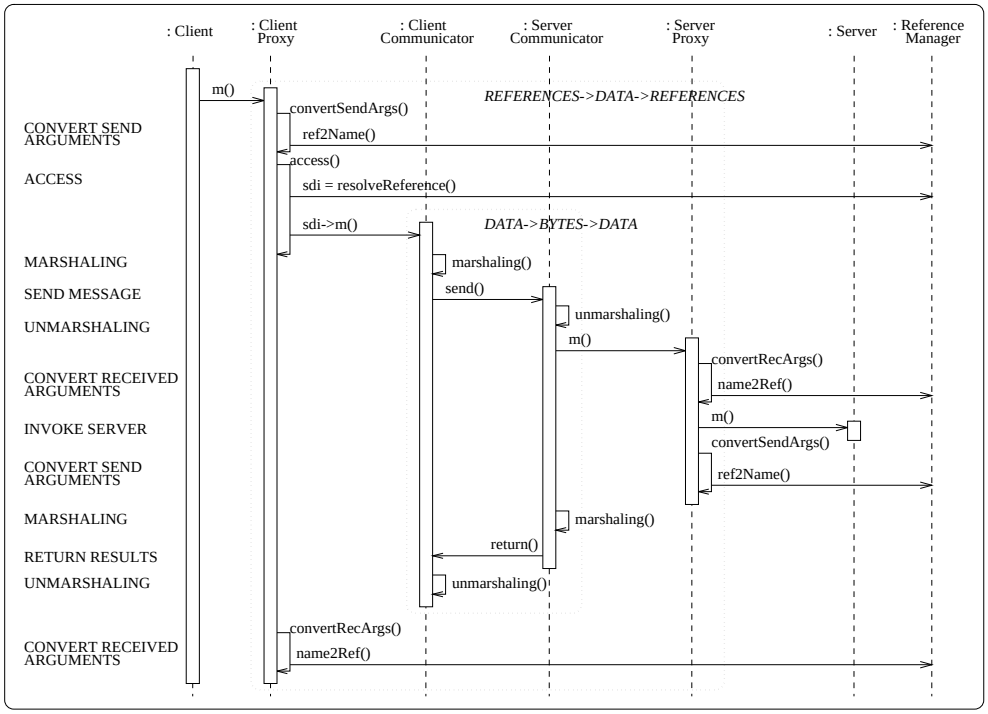

Figure 3: Distributed Proxy Pattern Collaborations.

2.6 Collaborations

Three types of collaborations are possible: functionality collaboration, which corresponds to the direct invocation of `Client` on `Server`; logical collaboration, where invocation proceeds through `Client Proxy` and `Server Proxy`; and physical collaboration, where invocation proceeds through the logical and physical layers.

The Booch interaction diagram in Figure 3 shows a physical collaboration which includes the functionality and logical collaborations.

In a first phase, after `Client` invokes `m` on `Client Proxy`, arguments are converted by method `convertSendArgs`. According to the specific arguments semantics, it converts object references to distributed names or it creates new objects which may include distributed names. Before invoking `m`, instantiated with the new arguments, it obtains a `Client Communicator` by using `resolveReference`. In the case of a logical collaboration `resolveReference` returns a `Server Proxy`.

When invoked, `Client Communicator` marshals its arguments, and sends a message to `Server Communicator` which unmarshals the message and invokes `m` on `Server Proxy`.

In a third phase `Server Proxy` converts received arguments to object references using `Reference Manager` according to the specific argument passing semantics. Finally, `m` is invoked on the `Server`.

After invocation on the `Server`, three other similar phases are executed to return results to `Client`.

In this collaboration two variations occur when transparently sending an object reference: there is no name associated with the object reference in the client side; and there is no reference associated with the distributed name in the server side. In the former situation the object reference corresponds to a local object, and `Reference Manager` is responsible for creating a `Server Proxy` and associating it with a new distributed name. In the latter situation the distributed name corresponds to a remote object, and `Reference Manager` is responsible for creating a `Client Proxy` and associating it with the distributed name. Note that in the physical collaboration proxy creation includes communicator creation.

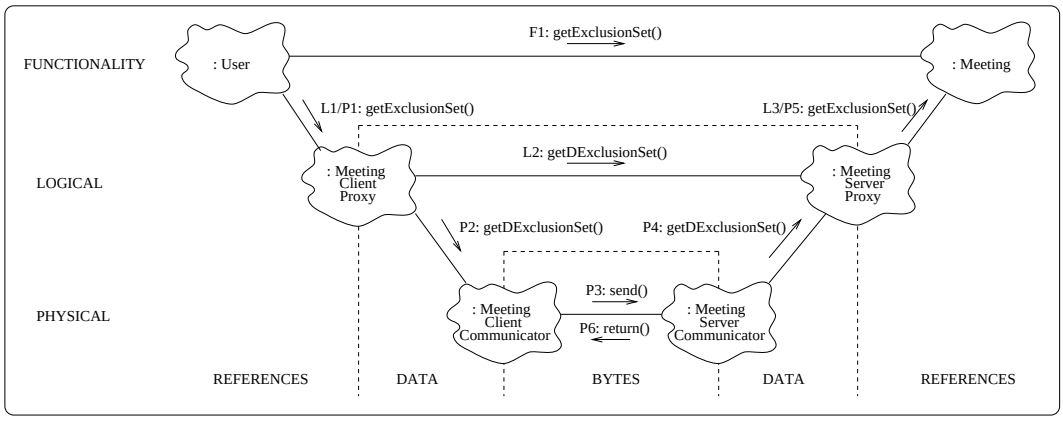

Figure 4: Layered Distributed Object Communication.

2.7 Forces Resolution

Figure 4 shows a layered distributed object communication which constitutes a design solution for the previous problems. In this example the `User` object invokes operation `getExclusionSet` on `Meeting`.

This solution takes into account the forces previously named:

- **Encapsulation** is achieved since logical and physical layers are encapsulated from the functional layer. Functionality code uses transparently the logical layer, `Client Proxy` and `Server` have the same interface.

- **Extensibility** is achieved because different argument passing semantics are supported as well as location time variations.

  Argument passing semantics are supported by redefining `convertSendArgs` and `convertRecArgs`. For instance for `convertSendArgs`: (1) to support transparent object passing it interacts with `Reference Manager` to convert references to distributed names; (2) to support deep copy a new object is created that contains the object data and recursively includes the data of all the objects it refers to; (3) to support partial copy a new object is created that contains the object data and associates distributed names with some of the objects it refers to. Redefinition of `convertRecArgs` converts distributed names to references and data objects to objects with references.

  Location time variations are supported because `resolveReference` can be invoked only once or before each invocation.

- **Modularity** is achieved by layered separation. Logical layer supports references managing and physical layer implements the underlying communication mechanisms.

- **Reusability** is achieved due to modularity. Functionality, logical and physical layers can be reused independently.

3 Distributed Proxy Framework

In the previous section a pure pattern description, independent of implementation details, was presented. In this section we present a C++ object-oriented framework which implements the pattern. For this implementation a three-layered framework [4] is used. The three layers are: pattern, composition and application. The Pattern layer contains the modules which implement the design patterns. The Composition layer contains the composition modules which result from the combination of different pattern modules. At the Application layer composition modules are integrated within the application and pattern modules are specialized according to the application-specific requirements.

Objects in pattern modules can be classified according to their properties: *Stable Objects* represent the objects which are encapsulated within pattern modules

and do not need to be specialized. *Variable Objects* represent the objects which need to be specialized for integration or customization. Integration objects are used for integration at composition and application layers. Customizable objects are used for customization at the application layer.

Composition module is responsible for integrating pattern modules and enforcing composition constraints. The composition modules should also offer a simple interface for final integration in the application and hide the particularities of inter-pattern dependencies from programmers at the application layer.

In the application layer, composition modules are integrated in the final application and pattern variations may be customized.

Since the paper describes a single pattern, the presentation will stress pattern and application layers.

3.1 Pattern Layer

Figure 5 represents the distributed communication module.

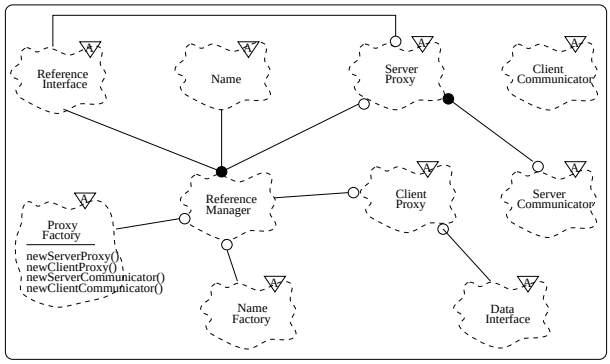

Figure 5: Distributed Proxy Module.

Two additional classes, `Proxy Factory` and `Name Factory`, are introduced. Class `Proxy Factory` is used by `Reference Manager` when generating proxy and communicator objects such that it does not need to be changed when moving from logical to physical communication or when introducing new distributed objects. Methods `newServerCommunicator` and `newClientCommunicator` are protected methods invoked by, respectively, `newClientProxy` and `newServerProxy`. By default these methods return null at the logical layer. Class `Name Factory` provides the name management services required by `Reference Manager`.

Class `Reference Manager` and the module collaborations are the stable part. Classes `Name` and `Name Factory` are integration classes at the composition layer. All other classes will need to be customized.

To support location time variation two subclasses of `Client Proxy` are defined, `CClient Proxy` and `IClient Proxy` (not shown). `IClient Proxy` invokes `resolve Reference` for each access while `CClient Proxy` invokes it only before the first access. At the application layer, the programmer should reuse the subclass that fulfills the application requirements.

When defining a physical layer it is necessary to create subclasses of `Client Communicator` and `Server Communicator`. The framework provides physical layer implementations on CORBA/ORBIX [5], sockets/ACE [6] and group-communication/ISIS [7]. For instance, a CORBA/ORBIX implementation defines a `CORBA Client Communicator` which contains a reference to a CORBA object. In this implementation it is not necessary to define a CORBA specific subclass of `Server Communicator`

3.2 Composition Layer

At the composition layer, modules implementing different patterns are combined. Classes `Name Factory` and `Name` are involved in the composition of distributed proxy with the naming pattern. A description of this pattern can be found in [8].

3.3 Application Layer

Figure 6 shows classes customization for the meeting example.

In order to enrich the functionality layer with logical distributed communication it is necessary to define `Meeting Reference Interface`, `Meeting Data Interface`, `Meeting Client Proxy`, `Meeting Server Proxy` and `Meeting Proxy Factory`. Class `Meeting Reference Interface` defines the interface, the user is client of: methods `Initiator getInitiator()` and `ExclusionSet getExclusionSet()`. It integrates functionality and logical layers because it is transparent to the user which

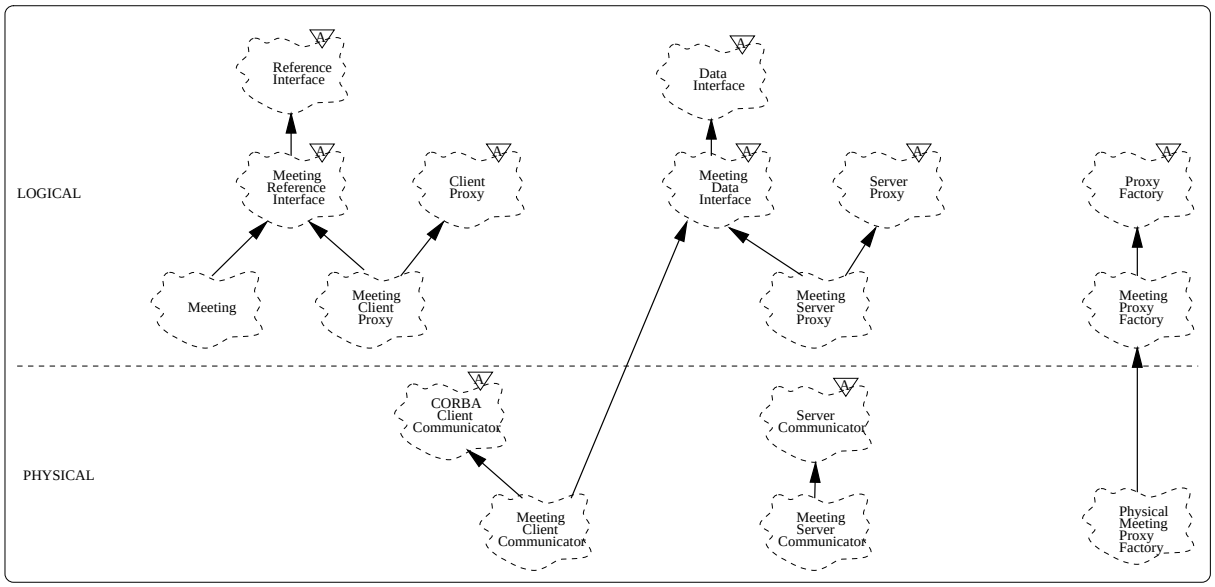

Figure 6: Classes Customization.

subclass he is using, `Meeting` or `Meeting Client Proxy`. Class `Meeting Data Interface` defines the data interface, methods `DataInitiator getDInitiator()` and `Name getDExclusionSet()`. It integrates logical and physical layers. Class `Meeting Client Proxy` defines method `getInitiator` to create a new `Initiator` object with the data returned by `GetDInitiator`. It also defines method `getExclusionSet` to invoke method `name2Ref` instantiated with the name returned by `getDExclusionSet`. Class `Meeting Server Proxy` is defined to do the inverse actions. Class `Meeting Proxy Factory` is defined to generate meeting proxies, methods `newServerProxy` and `newClientProxy`.

Physical distributed communication is incrementally introduced within the application. It is necessary to define `Meeting Client Communicator`, `Meeting Server Communicator` and `Physical Meeting Proxy Factory`. To define class `Meeting Client Communicator` the programmer should define a CORBA/IDL interface for `Meeting Data Interface`, and delegate execution on it. Class `Meeting Server Communicator` is the implementation of the IDL interface and delegates method execution to `Meeting Server Proxy`. Class `Physical Meeting Proxy Factory` is defined to generate meeting communicators, methods `newServerCommunicator` and `newClientCommunicator`.

4 Related Work

In [3] it is described the proxy principle: *"In order to use one service, potential clients must first acquire a proxy for this service; the proxy is the only visible interface of the service"*. The presented design applies and extends this principle by relaxing transparency and defining the logical layer. The former allows several argument passing semantics, transparency is preserved from a syntactic point of view because the server class and client proxy have the same interface, but different semantics occurs when communication duplicates non-constant objects. The latter allows incremental introduction of distribution with testing, debugging and simulation in a non-distributed environment.

Distributed communication is addressed by technology like CORBA [9] and JAVA [10]. In CORBA the implementation of distributed communication is encapsulated by an IDL (Interface Definition Language) and object references are dynamically created and passed across nodes. JAVA RMI (Remote

Method Invocation) defines remote interfaces which can dynamically resolve distributed methods invocations. A problem with these approaches is on their lake of composibility. Both, CORBA and JAVA, offer composed solutions which, for instance, strongly couple distributed communication with exception handling, while distributed proxy is a minimal solution which can be composed with other solutions. Another problem is that is is difficult to do communication optimization at the physical layer, as it is done by communicators.

The D Framework [11] defines a remote interface language, which allows the specification of several copying semantics. It is based on code generation. It uses JAVA RMI for automatic creation of proxies so, it is not clear how does it compose with naming. Another aspect is the lack of level of detail provided by the interface language it is not possible to do optimizations at the physical layer.

In the *Proxy pattern* [2, 12] clients of an object communicate with a representative rather than with the object itself. In particular the *Remote Proxy* variation in [12] corresponds to the logical layer of `Distributed Proxy`. However, `Distributed Proxy` allows dynamic creation of new proxies and completely decouples the logical layer from the physical layer.

5 Conclusions

This paper describes a design pattern and a framework for distributed communication. It defines three-layers of interaction, functionality, logical and physical, presenting the following qualities:

- *Decouples object-functionality from object-distribution.* Distribution is transparent for functionality code and clients of the distributed object are not aware whether the object is distributed or not.

- *Encapsulation of underlying distribution mechanisms.* Several implementations of distributed communication can be tested at the physical layer. Moreover, different implementations of communicators can be tested, e.g. sockets and CORBA, without changing the application functionalities. Portability across different platforms is also achieved.

- *Allows an incremental development process*. A functionality version of the application can be built first and logical and distribution introduced afterwards. The functionality version allows the test and debug of application functionalities ignoring distribution issues. The logical layer introduces distributed communication ignoring distribution communication mechanisms and preserving the object-oriented communication paradigm. At the logical layer testing, debugging is done in a non-distributed environment. Moreover, simulation of the distribution communication mechanisms can also be done in a non-distributed environment by implementing proxies which simulate the real communication mechanisms. Finally, at the physical layer the application is enriched with distributed communication mechanisms. Note that data gathered from simulations at the logical layer may help to decide on the final implementation.

The stage-based description used allows to describe design at a different level of abstraction than code. The design pattern describes the distributed communication solution independently of implementation details and its design qualities are inherent to the proposed design structure and collaborations. The framework description shows how the design qualities are preserved by a particular implementation where composition specific techniques as inheritance and delegation are used. This stage-based description allows design patterns reuse and code reuse. The former corresponds to pattern implementation and the latter to framework specialization.

The distributed proxy pattern was defined in the context of an approach to the development for distributed applications with separation of concerns (DASCo) initially described in [13]. Distributed communication is a DASCo concern and the presented design pattern define a solution for it. Moreover, it is part of a pattern language for the incremental introduction of partitioning into applications [14] which also includes configuration [15], replication [16] and naming [8].

The distributed proxy three-layered framework is publicly available from the page `http://albertina.inesc.pt/~ars/dasco`.

References

[1] Grady Booch. *Object-Oriented Analyis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[3] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *The 6th International Conference on Distributed Computer Systems*, pages 198–204, Cambridge, Mass., 1986. IEEE.

[4] António Rito Silva. Development and Extension of Frameworks. In Saba Zamir, editor, *Handbook of Object Technology*. CRC Press, 1998.

[5] IONA Technologies. Building Distributed Applications with Orbix and CORBA, 1996.

[6] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *11th and 12th Sun User Group Conferences*, San Jose, California and San Francisco, California, December 1993 and June 1994.

[7] K. Birman, R. Cooper, and B. Gleeson. Programming with process groups: Group and multicast semantics. Technical report, Teach. rep. TR-91-1185. Cornell University, Ithaca, New York, USA, January 1991.

[8] Pedro Sousa and António Rito Silva. Naming and Identification in Distributed Systems: A Pattern for Naming Policies. In *The 3^{rd} Conference on Pattern Languages of Programming, PLoP '96(Washington University technical report #WUCS-97-07)*, Allerton Park, Illinois, September 1996.

[9] Jon Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.

[10] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[11] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, PARC Technical report, February 1997.

[12] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[13] António Rito Silva, Pedro Sousa, and José Alves Marques. Development of Distributed Applications with Separation of Concerns. In *Proceedings of the 1995 Asia-Pacific Software Engineering Conference*, pages 168–177, Brisbane, Australia, December 1995.

[14] António Rito Silva, Fiona Hayes, Francisco Mota, Nino Torres, and Pedro Santos. A Pattern Language for the Perception, Design and Implementation of Distributed Application Partitioning, October 1996. Presented at the OOPSLA'96 Workshop on Methodologies for Distributed Objects.

[15] Francisco Assis Rosa and António Rito Silva. Component Configurer: A Design Pattern for Component-Based Configuration. In *The 2^{nd} European Conference on Pattern Languages of Programming, EuroPLoP '97*, pages 13–32, Kloster Irsee, Germany. Siemens Technical Report 120/SW1/FB, 1997, July 1997.

[16] Teresa Goncalves and António Rito Silva. Passive Replicator: A Design Pattern for Object Replication. In *The 2^{nd} European Conference on Pattern Languages of Programming, EuroPLoP '97*, pages 165–178, Kloster Irsee, Germany. Siemens Technical Report 120/SW1/FB, 1997, July 1997.