Distributed Proxy: A Design Pattern for Distributed Object Communication

António Rito Silva[1], Francisco Assis Rosa[2], Teresa Gonçalves[2] and Miguel Antunes[1]
[1] INESC/IST Technical University of Lisbon, Rua Alves Redol nº9, 1000 Lisboa, PORTUGAL
[2] HKS - Hibbitt, Karlsson & Sorensen, Inc. 1080 Main Street, Pawtucket, RI 02860, USA
Rito.Silva@acm.org, {rosa,goncalve}@hks.com, Miguel.Antunes@inesc.pt

This pattern was workshopped at the Conference of Pattern Languages of Programs (PLoP97), Allerton Park, Illinois, USA, September 1997.

Abstract

This paper presents the Distributed Proxy *pattern, a design pattern for distributed object communication. The* Distributed Proxy *pattern decouples distributed object communication from object specific functionalities. It further decouples logical communication from physical communication. The* Distributed Proxy *pattern enforces an incremental development process, encapsulates the underlying distribution mechanisms, and offers location transparency.*

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. Motivation

2.1. Example

An agenda application has several users which manipulate agenda items, either private (appointments) or shared (meetings). A meeting requires the participation of at least two users. When an agenda session starts, it receives an agenda manager reference from which the agenda user information can be accessed. It is simple to design a solution ignoring distribution issues. The UML [4] class diagram in Figure 1 shows the functionalities design of the agenda application, where distribution issues are ignored.

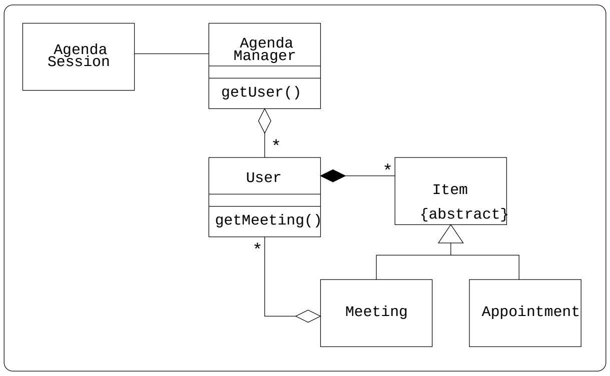

Figure 1. Agenda Functionalities Design.

Enriching this design with distribution is complex. For example we must consider different address spaces. In terms of our agenda application this means, that method `getUser` in `Agenda Manager` should return to the remote `Agenda Session` a `User` object across the network. Distributed communication implementation is another source of complexity. For instance, the communication between `Agenda Session` and `Agenda Manager` is implemented using sockets.

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.

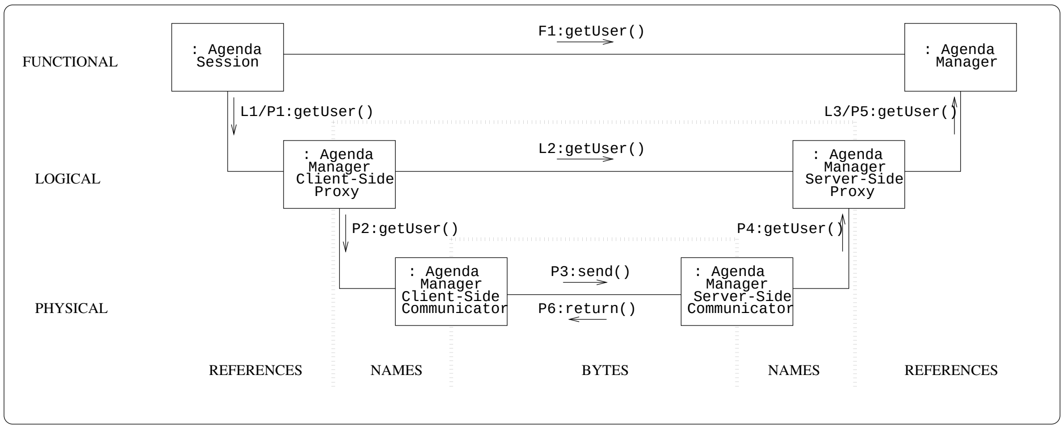

Figure 2. Layered Distributed Object Communication.

2.3. Forces

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

- **Complexity.** The problem and respective solution is complex. Several aspects must be dealt with: the specificities of the underlying communication mechanisms; and the diverse name spaces.
- **Object distribution.** Object references may be transparently passed between distributed nodes.
- **Transparency.** The incorporation of distributed communication should be transparent for functional classes by preserving the interaction model, object-oriented interaction, and confining the number of changes necessary in functionality code.
- **Flexibility.** The resulting applications should be flexible in the incorporation and change of distribution issues. The underlying communication mechanisms should be isolated and it should be possible to provide different implementations.
- **Incremental development.** Distributed communication should be introduced incrementally. Incremental development allows incremental test and debug of the application.

2.4. Solution

Figure 2 shows a layered distributed object communication which constitutes a design solution for the previous problems. In this example the `Agenda Session` object invokes method `getUser` on `Agenda Manager`.

The solution defines three layers: functional, logical, and physical. Functional layer contains the application functionalities and interactions that are normal object-oriented invocations. At the logical layer, proxy objects are introduced between distributed objects to convert object references into distributed names and vice-versa. This layer is responsible for the support of an object-oriented model of invocation, where distributed proxies are dynamically created whenever an object reference from another node is contained in a distributed message. Finally, the physical layer implements the distributed communication using the underlying communication mechanisms.

This solution takes into account the forces previously named:

- **Complexity** is managed by layered separation of problems. Logical layer supports name spaces and physical layer implements the underlying communication mechanisms.
- **Object distribution** is achieved because proxy objects convert names into references and vice versa.
- **Transparency** is achieved since logical and physical layers are decoupled from the functional layer. Functionality code uses transparently the logical layer, `Agenda Manager Client-Side Proxy` and `Agenda Manager` have the same interface.
- **Flexibility** is achieved since the physical layer, which contains the underlying communication mechanisms particularities, is decoupled from logical layer.
- **Incremental Development** is achieved since `Agenda Manager Client-Side Proxy` and `Agenda Manager` have the same interface, and the incorporation of the logical layer is done after the functional layer is developed. Moreover, `Agenda Manager Server-Side Proxy` and `Agenda Manager Client-Side Communicator` have the same interface, and the physical layer can be incorporated

after the logical layer is developed. This way, the application can be incrementally developed in three steps: functional development, logical development, and physical development. In the same incremental way we define the interaction between the participating components of the pattern. First we define the interaction `F1` at the functional level, as if no distribution is present. Then, when adding the logical layer we define interactions `L1 - L3`. Finally, when implementing the physical layer we establish the interaction chain `P1 - P6`.

3. Applicability

Use the *Distributed Proxy* pattern when:

- *An object-oriented interaction model is required between distributed objects.* Distributed objects are fine-grained entities instead of large-grained servers accessed by clients.
- *Several distributed communication mechanisms may be tested.* Moreover, the communication mechanism can be changed with a limited impact on the rest of the application.
- *Incremental development is required by the development strategy.* Incremental test and debug should be enforced.

4. Structure and Participants

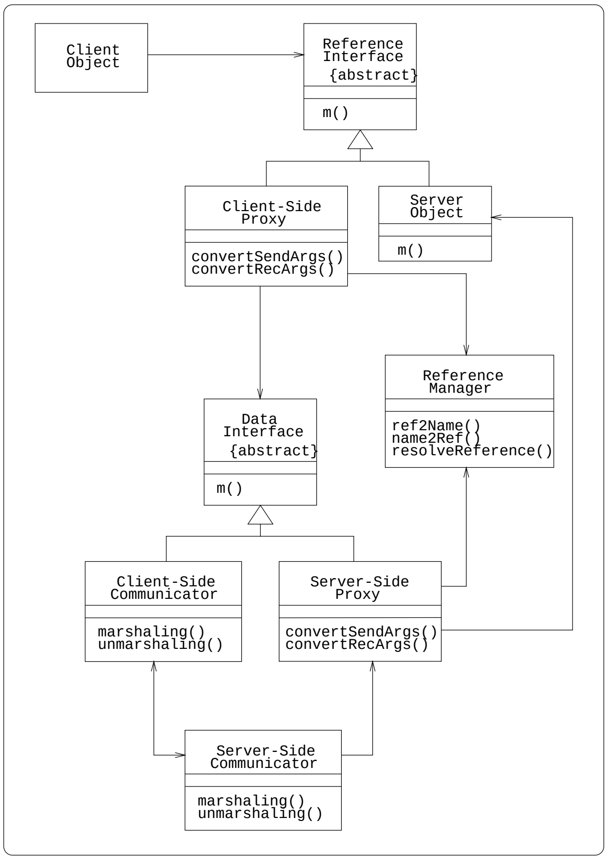

Figure 3. Distributed Proxy Pattern Structure.

The UML class diagram in Figure 3 illustrates the structure of *Distributed Proxy* pattern. Three layers are considered: functional, logical, and physical. Classes are involved in each layer: `Client Object` and `Server Object` at the functional layer, `Client-Side Proxy`, `Server-Side Proxy` and `Reference Manager` at the logical layer, and `Client-Side Communicator` and `Server-Side Communicator` at the physical layer. Two abstract classes, `Reference Interface` and `Data Interface`, define interfaces which integrate functional and logical layers, and logical and physical layers.

The pattern's main participants are:

- `Client Object`. Requires a service from `Server Object`, it invokes one of its methods.
- `Server Object`. Provides services to `Client Object`.
- `Client-Side Proxy`. It represents the `Server Object` in the client node. It is responsible for argument passing and remote object location. It uses the `Reference Manager` to convert sending object references to node independent names (distributed names) and received distributed names to object references. It also uses the `Reference Manager` to obtain a `Data Interface` where it proceeds with the invocation.
- `Server-Side Proxy`. It provides distribution support for the `Server Object` in the server node. It is the entry point for remote requests to the `Server Object`. As `Client-Side 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-Side Proxy` a `Server-Side Proxy`, when at the logical layer, or a `Client-Side Communicator`, when at the physical layer.
- `Distributed Name`. It defines an identifier which

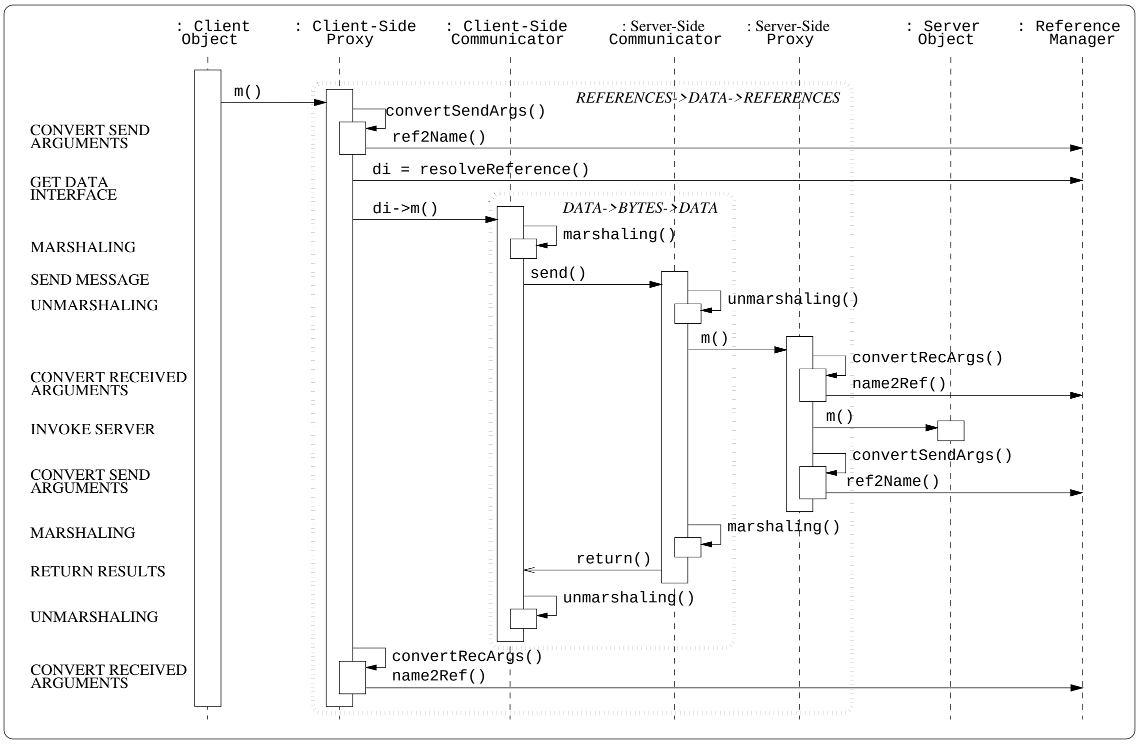

Figure 4. Distributed Proxy Pattern Collaborations.

is valid across nodes. It is an opaque object provided from outside to the `Reference Manager`.

- `Client-Side Communicator` and `Server-Side 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.
- `Reference Interface`. Defines an interface common to `Server Object` and `Client-Side Proxy`.
- `Name Interface`. Defines an interface common to `Server-Side Proxy` and `Client-Side Communicator`.

5. Collaborations

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

The UML sequence diagram in Figure 4 shows a physical collaboration which includes the functional and logical collaborations.

After `Client Object` invokes `m` on `Client-Side Proxy`, arguments are converted by method `convertSendArgs`. According to the specific arguments passing 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-Side Communicator` by using `resolveReference`. In the case of a logical collaboration `resolveReference` returns a `Server-Side Proxy`.

When invoked, `Client-Side Communicator` marshals its arguments, and sends a message to `Server-Side Communicator` which unmarshals the message and invokes `m` on `Server-Side Proxy`. `Server-Side Proxy` converts received arguments to object references using `Reference Manager` according to the specific argument passing semantics. Finally, `m` is invoked on the `Server Object`.

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

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

6. Consequences

The *Distributed Proxy* pattern has the following advantages:

- *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.

- *Allows an incremental development process.* A non-distributed version of the application can be built first and distribution introduced afterwards. Moreover, it is possible to simulate the distributed communication in a non-distributed environment by implementing communicators which simulate the real communication. Data can be gathered from these simulations to decide on the final implementation.

- *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.

- *Location transparency.* The conversion of distributed names into `Data Interface` objects, done by method `resolveReference`, gives location transparency of remote objects. That way it is possible to re-configure the application and migrate objects.

This pattern has the following drawback:

- *Overhead in terms of the number of classes and performance.* Four new classes are created or extended for each distributed object depending on whether the implementation uses delegation or inheritance, respectively. The performance overhead can be reduced if the implementation uses inheritance, communicators are subclasses of proxies.

7. Implementation

When implementing the *Distributed Proxy* pattern the following variations should be considered.

7.1. Arguments Passing

Arguments passing can have several semantics: (1) an object argument may be transparently passed between distributed nodes, a proxy is created in the receiving node; (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.

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.

7.2. Transparency

Transparency can be implemented if `Client-Side Proxy` and `Server Object` have the same interface: the interface defined by `Reference Interface`.

However, it can be the case that `Client Object` should be aware of distribution. For instance, the `Client Object` should deal with communication faults. In this situation, transparency can be relaxed by enriching the `Client-Side Proxy` interface according to `Client Object` distribution requirements.

Note that losing transparency does not imply a mix of the pattern layers. This means that even in the lack of transparency the pattern keeps the qualities resulting from the decoupling it defines.

7.3. Reference Resolution

Location of remote objects can either be at proxy creation time or at invocation time. The latter allows the remote objects to change their location.

Location of remote objects variations are supported because `resolveReference` can either be invoked only once or before each invocation. When performance is a requirement and remote objects do not migrate, reference resolution at proxy creation time can be used.

7.4. Naming Policies

There are several possibilities when implementing Reference Manager: distributed nodes can share a single Reference Manager or have its own Reference Manager.

As described in [14] there are several naming policies. A distributed name is universal if it is valid, i.e. can be resolved in all the distributed nodes. A distributed name is absolute if it denotes the same object in all distributed nodes. A distributed name is an identifier if remote objects have a single distributed name and does not exist two different remote objects with the same distributed name. A distributed name is pure if it does not contain location information. An impure name allows immediate invocation without previous resolution.

A distributed name can be sent to any distributed node if it is universal and absolute. Names with such properties can be supported by a single Reference Manager shared by all distributed nodes or by several cooperating Reference Managers which enforce their properties.

Identifier distributed names can be supported if the Reference Manager only generates new distributed names and whenever generating a new distributed name verifies that the object does not already have another distributed name.

When performance is a requirement Reference Managers can support impure names at the price of loosing object migration. Resolution of reference associated with a pure name requires that the Reference Manager collaborates with a name service that associates pure names with physical addresses where the invocation should occur. Name services are centralized or replicated entities. Impure distributed names avoid the need for a name service because during reference resolution the Reference Manager obtains the physical address from the distributed name.

7.5. Implementation of Communicators

The physical layer can be implemented using several communication mechanisms, e.g. sockets and CORBA [12]. The complexity of communicators implementation depends on the communication mechanisms used.

Communicators implemented using stream-based communication mechanisms, as sockets and named pipes, can apply the *Forwarder-Receiver* pattern [2] to encapsulate the underlying distributed communication mechanisms. Both, client- and server-side communicators use a pair of Forwarder and Receiver objects. Client-side communicators use the Forwarder object to send requests and use the Receiver object to receive the results. Server-side communicators use the Receiver object to receive messages and the Forwarder object to send the results. Due to the object-oriented interaction model of *Distributed Proxy* it is required that Receiver objects, on the server side, be reactive. Reactive Receivers can be implemented using the *Reactor* pattern [10].

A possible CORBA implementation defines IDL interfaces for Data interface classes: Server-Side Proxy and Client-Side Communicator. The Server-Side Communicator implements the IDL interfaces delegating to the Server-Side Proxy. Client-Side Communicators invoke on the IDL interface.

8. Sample Code

The code below shows the distributed communication associated with method getUser of class Agenda Manager which given the user's name, returns a User object. The code emphasizes the logical layer of communication.

A client-side proxy of Agenda Manager, CP_Agenda_Manager, which returns client-side proxies of User, CP_User, is defined. A CP_User is a subtype of Reference_Interface_User. In this case it is not necessary to convert send arguments since the only entity sent, string name, is not an object. Before invocation, a Data_Interface_Agenda_Manager, where the invocation should proceed, is obtained from the Reference Manager. After invocation, CP_Agenda_Manager converts the received distributed name into a Reference_Interface_User reference. The down-casts are necessary because object Reference Manager manipulates objects of type Data Interface and Reference Interface.

```
Reference_Interface_User* CP_Agenda_Manager::
getUser(const String* name)
{
  // empty convert send arguments

  // get data interface
  Data_Interface_Agenda_Manager*
    diam = static_cast<Data_Interface_Agenda_Manager*>(
              referenceManager_->resolveReference(this));

  // invoke
  Distributed_Name* dn = diam->getUser(name);

  // convert received arguments
  Reference_Interface_User*
    riu = static_cast<Reference_Interface_User*>(
                referenceManager_->name2Ref(dn));

  // return result
  return riu;
}
```

A server-side proxy of Agenda Manager, SP_Agenda_Manager, which returns a distributed name of a User, is defined. The SP_Agenda_Manager is a subtype of Data_Interface_Agenda_Manager and User is a subtype of Reference_Interface_User. In this case it is not necessary to convert received arguments since the only entity received, string name, is not an object. After invocation, the User object is converted to a distributed name.

```
Distributed_Name* SP_Agenda_Manager::
getUser(const String* name)
{
  // empty convert received arguments

  // invoke
  User* user = agendaManager_->getUser(name);

  // convert send arguments
  Distributed_Name*
    dn = referenceManager_->ref2Name(user);

  // return result
  return dn;
}
```

9. Known Uses

The DISGIS project [3] aims at providing effective and efficient development of Geographical Information Systems, where functions and data are increasingly distributed. Due to the large amount of data that is transmitted the *Distributed Proxy* is applied to provide an object-oriented interaction model while allowing performance improvements by adapting the logical and physical layers.

At Siemens [1] the *Distributed Proxy* pattern is applied in a system for hot rolling mill process automation. This system is a distributed system which uses CORBA as a basis. The different peers communicate via distributed proxy. Application processes use proxies to represent remote peers and the proxies use forwarder-receiver to actually perform IPC communication.

The system presented in [7] defines an interaction model which is independent of the transport protocol that is used to transmit messages between endpoints. Their use of the *Distributed Proxy* allows significant performance improvements to be made by extending the classes that provide distribution transparency to take advantage of facilities provided by the specific transport protocol.

In the DASCo pattern language [13] the *Distributed Proxy* pattern is integrated with other design patterns, *Functionality Configurer* [9], *Partitioning Configurer* [9], *Passive Replicator* [6] and *Naming* [14], to provide component distribution and replicated object distribution. An implementation is publicly available from http://www.esw.inesc.pt/˜ars/dasco.

10. Related Patterns

The *Proxy* pattern [5, 2] makes the clients of an object communicate with a representative rather than to the object itself. In particular the *Remote Proxy* variation in [2] 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.

The *Broker* pattern [2] defines a distributed architecture with decoupled components that interact by remote service invocations. The *Broker* architecture hides system- and implementation-specific details from the users of components and services. The *Distributed Proxy* pattern separates functional, logical and physical communications but allows programmers to write code in any of the layers.

The *Client-Dispatcher-Server* pattern [2] supports location transparency by means of a name service. The *Distributed Proxy* pattern also provides location transparency when method resolveReference is invoked on Reference Manager before each distributed invocation.

The *Forwarder-Receiver* pattern [2] supports the encapsulation of the underlying distributed communication mechanisms. This pattern can be used to implement the *Distributed Proxy* physical layer.

The *Serializer* pattern [8] streams objects into data structures and as well creates objects from such data structures. It decouples stream-specific issues, as backends, from the object being streamed. This pattern can be used to implement the Communicators marshaling and unmarshaling methods.

The *Reactor* pattern [10], *Acceptor* pattern and *Connector* pattern [11] can be used in the implementation of the *Distributed Proxy* physical layer with sockets.

The *Naming* pattern [14] describes an architecture centered on names, naming contexts and name spaces in which object denotation and identification is supported. The *Naming* pattern can be applied to implement class Reference Manager and its distributed naming policies.

Acknowledgments. The authors would like to thank Frank Buschmann for his valuable suggestions during the shepherding process.

References

[1] F. Buschmann, 1998. Personal communication.
[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.
[3] DISGIS. Esprit Project 22.084: DIStributed Geographical Information Systems: Models, Methods, Tools and Frameworks, 1996. http://www.gis.dk/disgis/Intro.htm.

[4] M. Fowler and K. Scott. *UML Destilled: Applying de Standard Object Modeling Language*. Addison-Wesley, 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[6] T. Goncalves and A. R. 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.

[7] N. Pryce and S. Crane. Component Interaction in Distributed Systems. In *Fourth International Conference on Configurable Distributed Systems*, pages 71–78, Annapolis, Maryland, USA, May 1998. IEEE Press.

[8] D. Riehle, W. Siberski, D. Baumer, D. Megert, and H. Zullighoven. Serializer. In R. Martin, D. Riehle, and F. Buschman, editors, *Pattern Languages of Program Design 3*, chapter 17, pages 293–312. Addison-Wesley, 1997.

[9] F. A. Rosa and A. R. Silva. Functionality and Partitioning Configuration: Design Patterns and Framework. In *Fourth International Conference on Configurable Distributed Systems*, pages 79–89, Annapolis, Maryland, USA, May 1998. IEEE Press.

[10] D. C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In J. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 529–545. Addison-Wesley, 1995.

[11] D. C. Schmidt. Acceptor and Connector. In R. Martin, D. Riehle, and F. Buschman, editors, *Pattern Languages of Program Design 3*, chapter 12, pages 191–229. Addison-Wesley, 1997.

[12] J. Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.

[13] A. R. Silva, F. Hayes, F. Mota, N. Torres, and P. 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.

[14] A. R. Silva, P. Sousa, and M. Antunes. Naming: Design Pattern and Framework. In *22nd Annual International Computer Software and Applications Conference*, pages 316–323, Vienna, Austria, August 1998. IEEE Press.