Distributed Proxy: A Design Pattern for the Incremental Development of Distributed Applications

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 n°9, 1000-029 Lisboa, PORTUGAL
Rito.Silva@inesc.pt,
WWW home page: http://www.esw.inesc.pt
[2] HKS - Hibbitt, Karlsson & Sorensen, Inc
1080 Main Street, Pawtucket, RI 02860, USA

Abstract. Developing a distributed application is hard due to the complexity inherent to distributed communication. Moreover, distributed object communication technology is always changing, todays edge technology will become tomorrows legacy technology. This paper proposes an incremental approach to allow a divide and conquer strategy that copes with these problems. It presents a design pattern 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. Moreover, the lack of performance measures at the beginning of the project and the existence of various distributed object communication technologies providing different features, recommend that choosing and introducing the technology should be delayed until performance measures are obtained during tests and simulations.

Herein, we propose an incremental approach for this problem which allows the transparent introduction of distributed object communication. In a first phase 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 technologies. In a second phase, the application is transparently enriched with physical distributed mechanisms. During this second step the distributed communication technology is chosen and the implementation is tuned for the specific application needs. The approach also allows an intermediate step where a quick implementation using a distributed communication technology is done to test the application in a real distributed environment before the final implementation. In this case the chosen distributed communication technology should allow a rapid prototyping.

Usually, object distributed communication involves the definition of proxies which represent remote services in the client space and encapsulate the remote object [1]. 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.

This paper presents a design pattern [2] for distributed object communication that uses the proxy approach. 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.

The rest of this paper is structured as follows. The next sections presents a design pattern for distributed communication using the format in [2]. Related work is presented and discussed in Sect. 10 and Sect. 11 presents the conclusions.

2 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.

3 Motivation

3.1 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 [3] class diagram in Fig. 1 shows the functionalities design of the agenda application, where distribution issues are ignored.

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` might be implemented using CORBA.

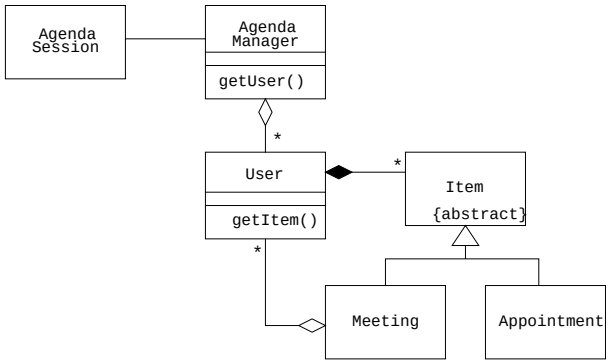

Fig. 1. Agenda functionalities design.

3.2 Issues

The design solution for distributed object communication must consider the following issues:

- **Complexity.** The problem and respective solution is complex. Several aspects must be dealt with: the specificities of the distributed 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 distributed 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.

3.3 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. The 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

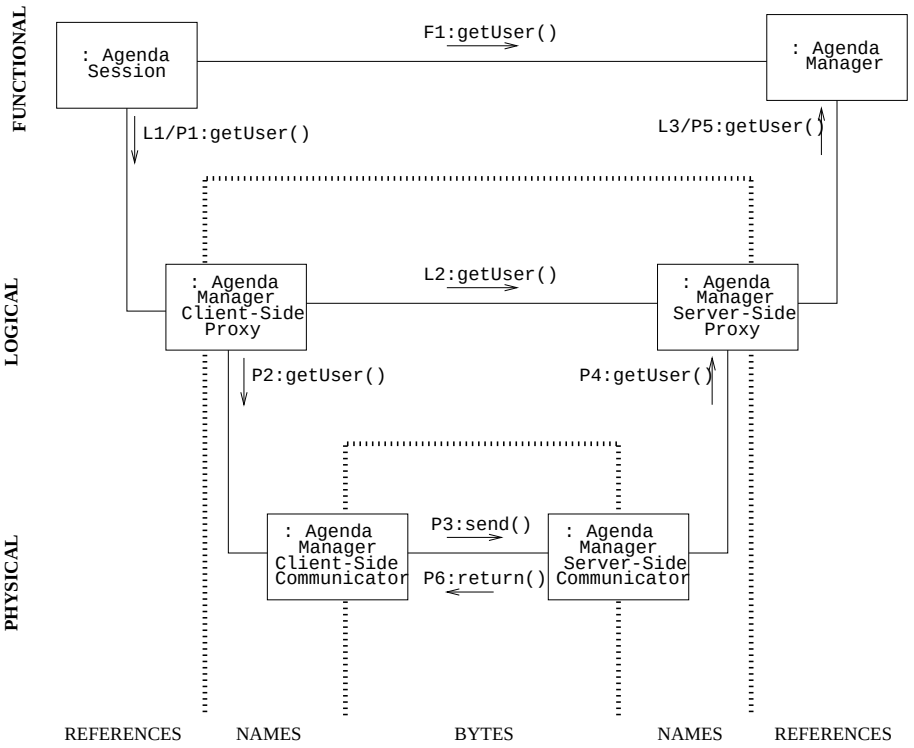

Fig. 2. Layered distributed object communication.

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 distributed communication mechanisms.

This solution takes into account the issues previously named:

- **Complexity** is managed by layered separation of problems. Logical layer supports name spaces and physical layer implements the distributed 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 by means of the physical layer, which contains the distributed 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 was 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`.

4 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 testing and debugging should be enforced.

5 Structure and Participants

The UML class diagram in Fig. 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

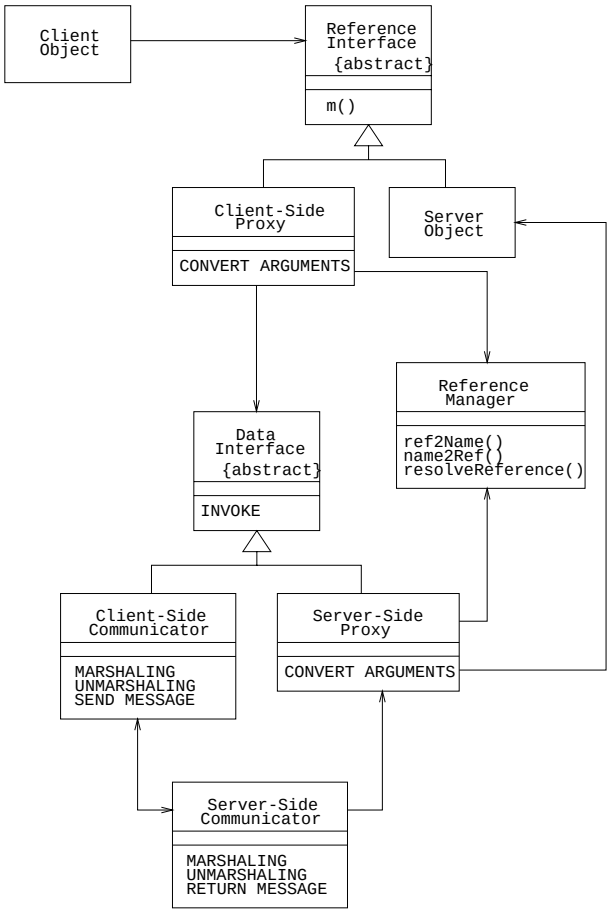

Fig. 3. Distributed proxy pattern structure.

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 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 it is responsible for `MARSHALING` and `UNMARSHALING` to, respectively, convert arguments to streams of bytes and vice-versa.
- `Reference Interface`. Defines an interface common to `Server Object` and `Client-Side Proxy`, an interface that supports method `m`.
- `Data Interface`. Defines an interface common to `Server-Side Proxy` and `Client-Side Communicator`, an interface that supports `INVOKE`.

6 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 Fig. 4 shows a physical collaboration which includes the functional and logical collaborations.

After `Client Object` invokes `m` on `Client-Side Proxy`, arguments are converted. According to the specific arguments passing semantics, it converts object references to distributed names or it creates new objects which may include distributed names. To invoke on a `Data Interface`, the `Client-Side Proxy` obtains a `Client-Side Communicator` by using `resolveReference`. In the case of a logical collaboration `resolveReference` returns a `Server-Side Proxy`. The invocation on `Data Interface` is instantiated with the converted arguments.

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

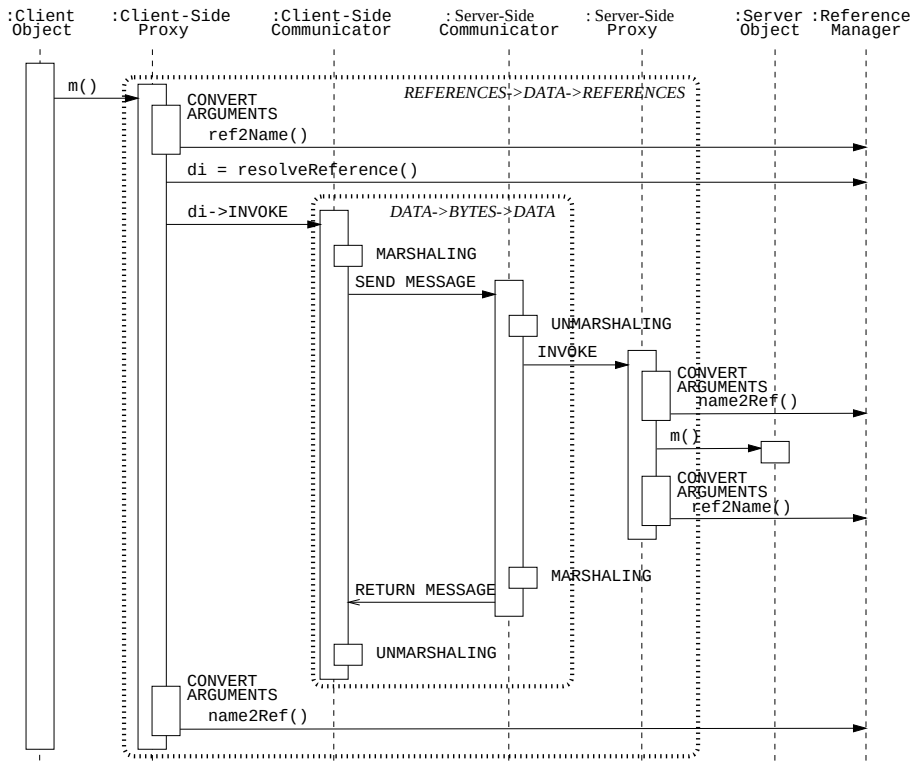

Fig. 4. Distributed proxy pattern collaborations.

7 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 detect possible bottlenecks and decide on the final implementation architecture.
- *Encapsulation of distributed communication mechanisms.* Several implementations of distributed communication can be tested at the physical layer, 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 number of classes overhead can be reduced if they are automatically generated. The performance overhead due to indirections can be reduced if the implementation uses inheritance, communicators are subclasses of proxies.

8 Implementation

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

8.1 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.

At the logical layer argument passing semantics are supported by `CONVERT ARGUMENTS` code blocks. For instance before sending a message to the physical layer, `CONVERT ARGUMENTS` code block should implement the argument passing semantics: (1) to support transparent object passing it interacts with `Reference Manager` to convert references to distributed names; (2) to support deep copy the object is passed to the physical layer where its data will be marshaled and recursively 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. After receiving a message from the physical layer and before dispatching it to the functional layer, `CONVERT ARGUMENTS` code block is responsible for converting distributed names to references and data objects to objects with references.

8.2 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 but only a change of the interfaces between layers. These interfaces must express,

make visible, some distribution aspects of the communication. This means that even in the lack of transparency the pattern keeps the qualities resulting from the decoupling it defines.

8.3 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 [4] 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 not two different remote objects with the same distributed name exist. 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 Manager`s 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 Manager`s 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.

8.4 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.

Location of remote objects may dependent on the name policies used. If names are impure then reference resolution at proxy creation time should be used.

8.5 Data Interface Implementation

The definition of the `Data Interface` is crucial since it establishes the physical layer interface which will be used by proxies. Two major variations are possible

when defining `Data Interface` and they are related with the goals of using the *Distributed Proxy* pattern:

- **Technology Encapsulation**. Distributed object communication technology, such as DCOM, CORBA and JAVA RMI, is used to develop a distributed application without tangling the functional code with distribution code.
- **Technology Implementation**. The pattern can be also applied to implement a distributed object communication mechanism like a CORBA compliant ORB.

Using the technology encapsulation approach a `Data Interface` is defined for each `Reference Interface`. In the `Data Interface` all object references are replace by distributed names references. Concrete client communicators must be defined for each defined `Data Interface` (see Sect. 8.6). In this way the code that uses the communication technology will be encapsulated in the client and server communicators and each client proxy will use its corresponding `Data Interface` class.

Using the technology implementation approach `Data Interface` will define a fixed interface for the physical layer and all the client proxies will use that interface. That interface should offer methods for marshaling and unmarshaling and to send requests and receive results. In this approach the `Data Interface` will correspond to the `Forward` class of the *Forward-Receiver* pattern [5] such that the replacement of the underlying communication technology does not have repercussions on the proxies code. Different communication mechanism will be implemented by different pairs of client-server communicators. Each client communicator implements the `Data Interface`. In this way the implementation details of the underlying communication is encapsulated in the communicators. Also, by defining a fixed interface to the physical layer automatic generation of communicators is simplified.

It is also possible to use a hybrid solution where for some classes the communication is done using a commercial distributed object communication technology and for other classes the communication is done using a specific communication mechanism implemented by the application programmer itself. This may be useful in applications where some of the application's remote classes have specific requirements that are not covered by the distributed object communication technology being used.

All three implementations provide a clean separation of the physical layer from the logical layer, allowing programmers to change the communication mechanisms without having to change the proxy´s code.

8.6 Implementation of Communicators

The implementation of the communicators depends on how `Data Interface` was defined.

Using the technology encapsulation approach the communicator implementation is very simple. Consider a CORBA implementation. CORBA IDL interfaces

are defined for each `Data interface` classes. The `Server-Side Communicator` implements the IDL interfaces delegating to the `Server-Side Proxy`. `Client-Side Communicators` contain a CORBA reference and invoke on the IDL interface. The code that interacts with CORBA is within communicators. Catching exceptions and manipulating the CORBA types (`CORBA::Any`, `CORBA::Octet`) code is encapsulated in the communicators and therefore separated from the functional code.

Using the technology implementation approach, for each type of communication, e.g., ISIS, TCP, UDP, IIOP, a concrete client/server communicator pair is defined. The complexity of communicators implementation will depends on the communication mechanisms. In this case design patterns such *Forward-Receiver* or *Acceptor*, *Connector* and *Reactor* [6] can be used.

9 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 and the physical layer using the technology encapsulation approach for CORBA.

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;
}
```

The following code shows the CORBA IDL definition for the Agenda Manager server-side communicator.

```
interface SC_User
{
  // CORBA IDL interface for server-side
  // communicator of the User class
}

interface SC_Agenda_Manager
{
  SC_User getUser(in string name);
};
```

The client-side communicator implementation of the getUser method is straightforward. Each client-side communicator inherits from its corresponding Data Interface class and from CorbaCComm that defines behavior to all the CORBA client-side communicators. In the example below the class CorbaCC_Agenda_Manager inherits from Data_Interface_Agenda_Manager. It simply narrows its CORBA::Object_ptr of the corresponding server communicator to the correct interface type, and then delegates the execution to the CORBA proxy. The method getDName, defined in CorbaCComm, obtains a distributed name for the received remote reference. In this case the distributed name is just a container for a CORBA reference that also contains some type information that is used by the Reference Manager to create the correct proxies.

```
Distributed_Name* CorbaCC_Agenda_Manager::
getUser(const String* name)
{
  SC_Agenda_Manager_ptr access;
  SC_User_ptr user_corba_ref;

   // Get the concrete CORBA reference
   access = SC_Agenda_Manager::_narrow(obj_ptr_);
   // Remote invocation
  user_corba_ref  = access->getUser(name);
  return this->getDName(user_corba_ref);
}
```

Each server-side communicator is the implementation of a CORBA IDL interface. It inherits from the IDL generated class and from `CorbaSComm`. It contains a reference to its server-side proxy where it delegates the method execution. The method `getCorbaRef`, defined in `CorbaSComm`, given a distributed name returns a CORBA reference.

```
SC_User_ptr CorbaSC_AgendaManager::
getUser(const String* name)
{
  Distributed_Name *dn =
    static_cast<Data_Interface_Agenda_Manager*>
      (sp_)->getUser(name);

  return SC_User::_narrow(this->getCorbaRef(dn));
}
```

10 Related Work

In [1] the proxy principle is described: *"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-side 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.

The system presented in [7] also identifies the need of an interaction model which is independent of the transport protocol that is used to transmit messages between endpoints. This decoupling permits performance improvements by taking advantage of the facilities provided by the specific transport protocol. DeLine [8] also defines an approach that allows a component´s functional and interactive concerns to be separated. DeLine emphasizes reuse, a component´s interaction is captured in a packager, which may either be reused directly or automatically generated from an high-level description.

Distributed communication is addressed by technology like CORBA [9] and JAVA/RMI [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. Both, RMI and most of the existing CORBA implementations apply the `Distributed Proxy` pattern or some of its variations.

The D Framework [11] defines a remote interface language, which allows the specification of several copying semantics. It is based on code generation. Due to the lack of level of detail provided by the interface language it is not possible to do optimizations at the physical layer.

The *Distributed Proxy* patterns is related to the follow design patterns:

- The *Proxy* pattern [2, 5] makes the clients of an object communicate with a representative rather than to the object itself. In particular the *Remote Proxy* variation in [5] 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 [5] 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 [5] 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 [5] supports the encapsulation of the distributed communication mechanisms. This pattern can be used to implement the *Distributed Proxy* physical layer.
- The *Serializer* pattern [12] 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 [6], *Acceptor* pattern and *Connector* pattern [13] can be used in the implementation of the *Distributed Proxy* physical layer.
- The *Naming* pattern [4] 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.

11 Conclusions

This paper describes a design pattern for distributed communication. It defines three layers of interaction: functionality, logical and physical.

The design patterns 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 and 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 *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 [14]. 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 [15] which also includes configuration [16], replication [17] and naming [4].

The *Distributed Proxy* pattern was experimented in the DISGIS project [18]. DISGIS 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* was applied to provide an object-oriented interaction model while allowing performance improvements by adapting the logical and physical layers independently of the communication technology.

References

[1] 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.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[3] Martin Fowler and Kendall Scott. *UML Destilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[4] António Rito Silva, Pedro Sousa, and Miguel Antunes. Naming: Design Pattern and Framework. In *IEEE 22nd Annual International Computer Software and Applications Conference*, pages 316–323, Vienna, Austria, August 1998.

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

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

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

[8] Robert DeLine. Avoiding packaging mismatch with flexible packaging. In *22th International Conference on Software Engineering*, pages 97–106, Los Angeles, CA, USA, May 1999.

[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] Dirk Riehle, Wolf Siberski, Dirk Baumer, Daniel Megert, and Heinz Zullighoven. Serializer. In Robert Martin, Dirk Riehle, and Frank Buschman, editors, *Pattern Languages of Program Design 3*, chapter 17, pages 293–312. Addison-Wesley, 1997.

[13] Douglas C. Schmidt. Acceptor and Connector. In Robert Martin, Dirk Riehle, and Frank Buschman, editors, *Pattern Languages of Program Design 3*, chapter 12, pages 191–229. Addison-Wesley, 1997.

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

[15] 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.

[16] Francisco Assis Rosa and António Rito Silva. Functionality and Partitioning Configuration: Design Patterns and Framework. In *IEEE Fourth International Conference on Configurable Distributed Systems*, pages 79–89, Annapolis, Maryland, USA, May 1998.

[17] 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.

[18] DISGIS. Esprit Project 22.084: DIStributed Geographical Information Systems: Models, Methods, Tools and Frameworks, 1996. http://www.gis.dk/disgis/Intro.htm.