Distributed Proxy: A Design Pattern for Distributed Object Communication

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

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

An example motivates for the problems and respective forces involved in distributed object communication.

2.1 Example

A distributed 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 Booch[1] class diagram in Figure 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 operation `getUser` in `Agenda Manager` should return to the remote `Agenda Session` a `User` object across the network. Another source of complexity is need to implement distributed communication. For instance, the communication between `Agenda Session` and `Agenda Manager` is implemented using sockets. From a user's perspective, however, all distribution issues should be hidden. They just want to manipulate agenda items.

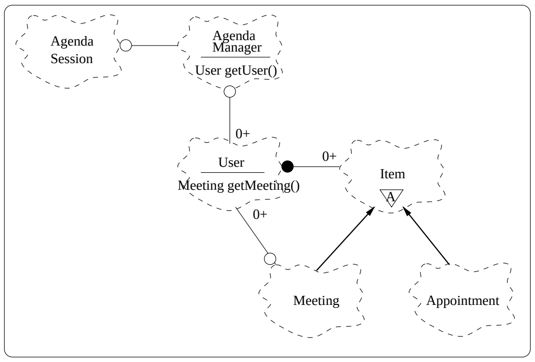

Figure 1: Agenda Functionalities Design.

2.2 Problem

Construct 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 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: 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 functionality classes by preserving the interaction model, object-oriented interaction, and confining the number of changes necessary in functionality code.

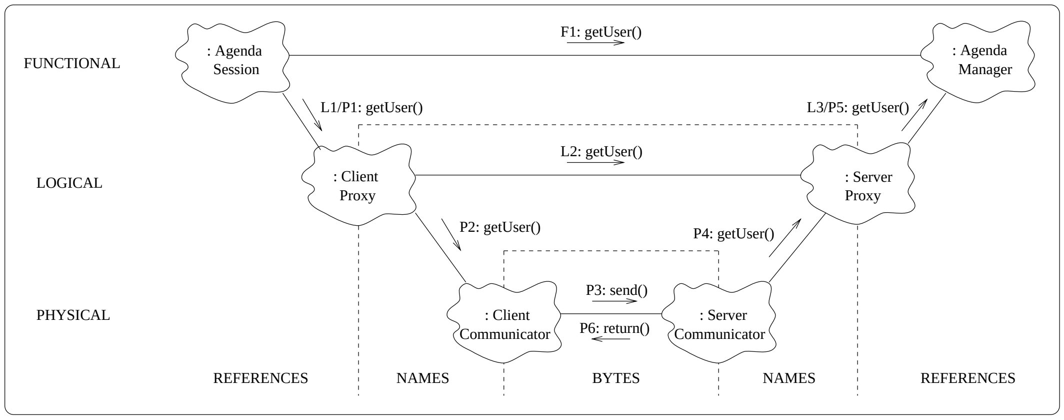

Figure 2: Layered Distributed Object Communication.

- **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 operation `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, `Client 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 `Client Proxy` and `Agenda Manager` have the same interface, and the incorporation of the logical layer is done after the functional layer is developed. Moreover, `Server Proxy` and `Client 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.

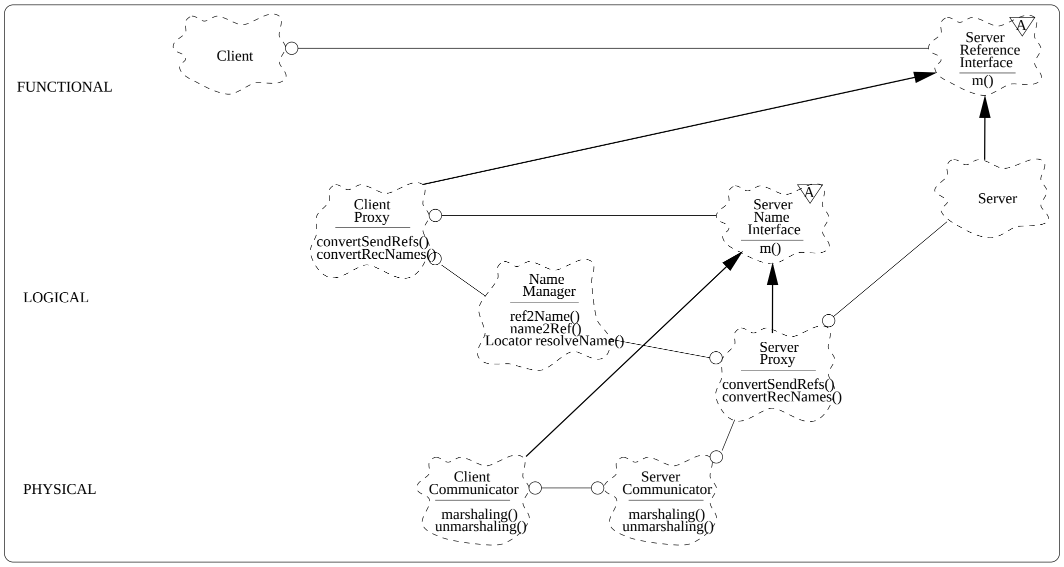

Figure 3: Distributed Proxy Pattern Structure.

- *Incremental development is required by the development strategy.* Incremental test and debug should be enforced.

4 Structure and Participants

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

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 node. It uses the `Name Manager` to convert sending object references to distributed names and received distributed names to object references. In particular, the method `convertSendRefs` is responsible for converting object references to distributed names, the method `convertRecNames` for converting distributed names to object references. It also gets a `Locator` from `Name Manager` to proceed with invocation. A pair of `convertSendRefs` and `convertRecNames` is defined for each method.
- `Server Proxy`. It provides distribution support for the `Server` object in the server node. It is the entry point for remote requests to the `Server`. As `Client Proxy` it is responsible for reference and name conversions.
- `Name Manager`. It is responsible for the distributed naming policies, e.g. Unique Universal Identifiers (UUID). It associates object references to distributed names and vice-versa. In particular, the method `ref2Name` is responsible for converting an object reference to a distributed name, the method `name2Ref` for converting a distributed name to an object reference. It also associates distributed names with `Locators`. In particular, the method `resolveName` is responsible for converting a distributed name to a locator.
- `Locator`. Defines an address where execution can proceed. A `Locator` can be logical, in which case it is an object reference to a `Server Proxy`, or physical, in which case it includes a distributed address, e.g. socket address.
- `Client Communicator` and `Server Communicator`. They are responsible for implementing the distributed communication. For each called method, `marshaling` and `unmarshaling` methods are defined to convert distributed names and data to streams of bytes and vice-versa.
- `Server Reference Interface`. Defines an interface common to `Server` and `Client Proxy`.
- `Server Name Interface`. Defines an interface common to `Server Proxy` and `Client Communicator`.

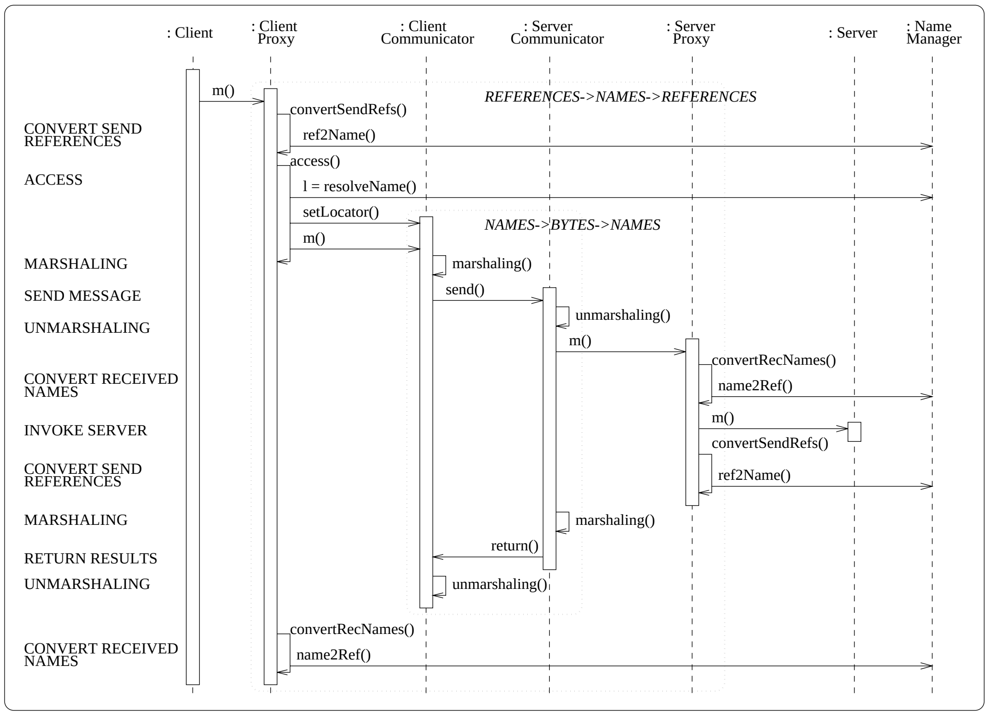

Figure 4: Distributed Proxy Pattern Collaborations.

5 Collaborations

Three types of collaborations are possible: functional 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 4 shows a physical collaboration which includes the functional and logical collaborations.

In a first phase, after `Client` invokes `m` on `Client Proxy`, object references are converted into distributed names by `convertSendRefs`. Before invoking `m`, instantiated with distributed names, in `Client Communicator` it is necessary to get the `Locator` associated with `Client Proxy` by using `resolveName`. This `Locator` will be used to access the `Server Proxy`, if it is a logical locator, or the `Server Communicator`, if it is a physical locator.

When invoked, `Client Communicator` marshals data and distributed names, 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 distributed names to object references using `Name Manager`. 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 possible variations occur when there is no name associated with the object reference in the client side and when there is no reference associated with the distributed name in the server side. The former situation means that the object reference corresponds to a local object, and method `convertSendRefs` has to create a `Server Proxy` and associate it with a new distributed name in the `Name Manager`. In the latter situation method `convertRecNames` has to create a `Client Proxy` and associate it with the distributed name in the `Name Manager`.

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 com-

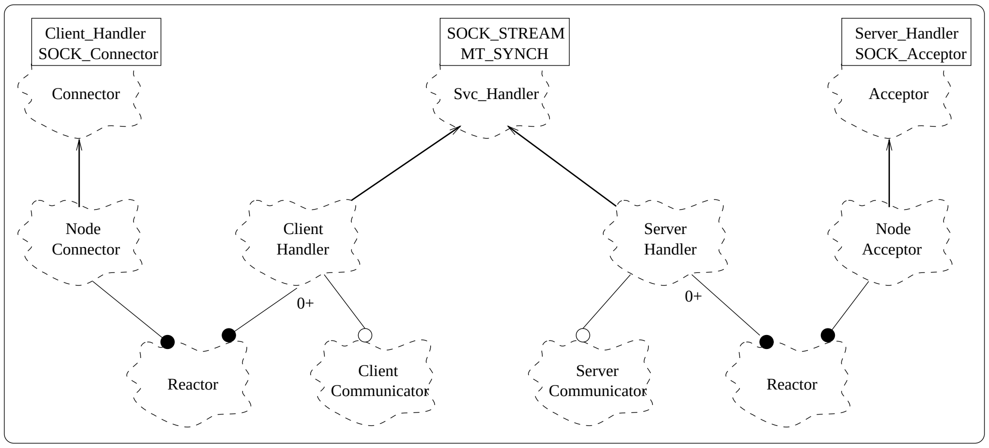

Figure 5: ACE Implementation Structure.

munication 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 convertion of distributed names into locators, done by operation `resolveName`, gives location transparency of remote objects. That way it is possible to re-configure the application and migrate objects.

This pattern has the following drawbacks:

- *There is an overhead in terms of the number of classes and performance.* Four new classes are created or extended for each distributed communication 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.

6 Implementation

6.1 Variations of Naming Policies

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

As described in [2] there are several naming policies. According to name policies, names can be universal or local, absolute or relative and pure or impure. A distributed name is universal if is valid in all the distributed nodes, it exists in all `Name Managers`. A distributed name is absolute if it denotes the same object in all distributed nodes, method `resolveName` returns the same `Locator` in all nodes. Finally, a distributed name is pure if it does not contain location information.

To support dynamic re-configuration and migration the distributed names should be universal, absolute and pure. A distributed name can be sent to any distributed node, because it is universal, and it denotes the same object everywhere, because it is absolute. Names with such properties are called Unique Universal Identifiers (UUIDs). UUIDs can be supported by a single `Name Manager` shared by all distributed nodes or by several cooperating `Name Manager`s enforcing distributed names properties.

When performance is a requirement `Name Manager`s can support impure names with the price of loosing re-configuration. Impure distributed names avoid the locator conversion. The distributed name is itself a locator.

6.2 Implementation of Communicators

The physical layer is implemented using the underlying communication mechanisms. In this section it is described a possible implementation of Communicators on top of the ACE [3] framework. ACE (Adaptive Communication Environment) is an object-oriented network programming framework that encapsulates operating system concrete mechanisms with C++ classes. In this implementation the ACE features for interprocess communication will be used.

This physical architecture implementation considers a pair of (unidirectional) sockets between two communicating distributed nodes. One socket – called in-socket – is used to receive service request messages and the other – called out-socket – to send service request messages. Sockets are encapsulated by the ACE `Service Handler` objects.

Figure 5 presents some relevant ACE classes used in this implementation. The `Reactor` class from ACE's *Reactor pattern* [4] is used to register, remove and dispatch `Service Handler` objects. Two reactors are needed, one for each node. In the client node a `Node Connector` object, subclass of ACE's `Connector` [5], is used to establish the communication with server node, it generates a `Client Handler` object which encapsulates the out-socket to the server node. The `Client Handler` is used to send messages associated with invocations and to receive its results. In the server node a `Node Acceptor` object,

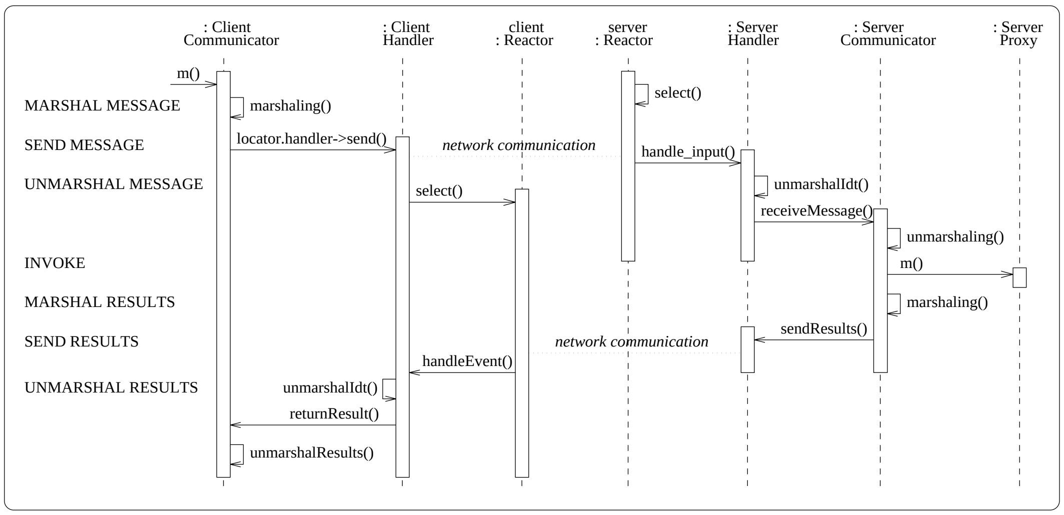

Figure 6: ACE Implementation Collaboration.

subclass of ACE's `Acceptor` [5], is used to accept connections from calling nodes, it generates a `Server Handler` object which encapsulates the in-socket from the client node. The `Server Handler` is used to receive messages associated with invocations and to return its results.

Figure 6 shows the ACE implementation of the collaborations between communicators.

In a first phase `Client Communicator` creates a message with the `Server Proxy` name, method name and arguments. The message includes the `Client Communicator` identification. The method `marshaling` is responsible for marshaling all the data. Afterwards, it uses the locator to identify the `Client Handler` where method `send` sends the message. When the message is received in the server node, `server :Reactor` dispatches the `Server Handler` object by invoking `handle_input`. `Server Handler` identifies the `Server Communicator`, using method `unmarshalIdt`. Method `unmarshaling` is responsible for unmarshaling the message. Finally, method `m` is invoked on `Server Proxy`.

In a second phase, after invocation execution on `Server Proxy`, the results are returned back in a similar manner. `Client Handler` identifies the `Client Communicator`, using method `unmarshalIdt`, because its identification is included in the sent message. To simulate a synchronous call, the `Client Communicator` blocks in a condition which is signaled by `Client Handler`. The method `returnResult` is responsible for signaling the condition.

7 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 proxy of `Agenda Manager`, `CP_Agenda_Manager`, which creates client proxies of `User`, `CP_User`, is defined. Also their server proxies, `SP_Agenda_Manager` and `SP_User` are defined. Methods `getUserConvertSendRefs` of `CP_Agenda_Manager` and `getUserConvertRecNames` of `SP_Agenda_Manager` do not need to be defined since the only entity sent, string `name`, is not an object. However, method `getUserConvertSendRefs` of `SP_Agenda_Manager` converts the object reference of `User` to a distributed name. If a distributed name does not exist it creates a server proxy and a distributed name for `User`.

```
void SP_Agenda_Manager::
getUserConvertSendRefs(User *user,DName **dnameUser)
{
  // obtains user distributed name associated
  // with user object
  *dnameUser = nameManager_->ref2Name(user);

  // a distributed name does not exists,
  // it is not distributed
  if (!*dnameUser) {
    // creates new server proxy
    SP_User *spUser = new SP_User(user);

    // creates new distributed name
    // server proxy argument serves for future location
    *dnameUser = nameManager_->newDName(spUser)

    // associates user object with distributed name
    nameManager_->bind(user,*dnameUser);
  }
}
```

Method `getUserConvertRecNames` of `CP_Agenda_Manager` converts the received distributed name into a `CP_User` reference. It may be the case that the `CP_User` does not exist and has to be created.

```
void CP_Agenda_Manager::
getUserConvertRecNames(DName *dnameUser,CP_User **cpUser)
{
  // obtains user client proxy associated
  // with distributed name
  *cpUser = nameManager_->name2Ref(dnameUser);

  // a client proxy does not exists
  if (!*cpUser) {
    // creates new client proxy
    *cpUser = new CP_User(dnameUser);

    // associates client proxy with distributed name
    nameManager_->bind(*cpUser, dnameUser);
  }
}
```

The redefinition of `Client Proxy`'s `access` method is also operation specific. Client proxy uses its distributed name to obtain a logical `Locator` object. Afterwards it accesses the object. The logical `Locator` object includes the address of the `Server Proxy` object.

```
DName *CP_Agenda_Manager::
getUserAccess(const String* name)
{
  // obtains logical locator
  Locator *loc =
    nameManager_->resolveName(dnameAgendaManager_);

  // accesses server proxy
  // returns a distributed name
  return loc.sp->getUser(name);
}
```

In the physical layer a physical `Locator` object is returned by the `resolveName` method. `Client Communicator` knows the internal `Locator`'s structure and uses its information to proceed with access across the network. In this case method `getUserAccess` is redefined in a subclass of `CP_Agenda_Manager`.

```
DName *P_CP_Agenda_Manager::
getUserAccess(const String* name)
{
  // obtains logical locator
  Locator *loc =
    nameManager_->resolveName(dnameAgendaManager_);

  // sets locator at client communicator
  communicator_->setLocator(loc);

  // invokes communicator which returns a distributed name
  return communicator_->getUser(name);
}
```

8 Known Uses

CORBA [6] uses the *Distributed Proxy pattern*. Implementation of distributed communication is encapsulated by an IDL and object references are dynamically created and passed across nodes. Moreover, CORBA implementations support co-location for the purpose of debugging. Co-location implements the logical layer of *Distributed Proxy pattern* because code executes in a centralized node.

In the DASCo pattern language [7] the *Distributed Proxy patterns* is integrated with other design patterns, *Component Configurer* [8] and *Passive Replicator* [9], to provide component distribution and replicated object distribution, respectively.

9 Related Patterns

The *Proxy pattern* [10, 11] makes the clients of an object communicate with a representative rather than to the object itself. In particular the *Remote Proxy* variation in [11] 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 *Client-Dispatcher-Server pattern* [11] supports location transparency by means of a name service. The `Distributed Proxy pattern` also provides location transparency when the `Name Manager` supports unique universal identifiers.

The *Forwarder-Receiver pattern* [11] supports the encapsulation of the underlying distributed communication mechanisms. This pattern can be used to implement the `Distributed Proxy` physical layer.

The *Reactor pattern* [4], *Acceptor pattern* and *Connector pattern* [5] can be used in the implementation of the `Distributed Proxy` physical layer as shown in the implementation section.

The *Component Configurer pattern* [8] decouples component configuration from component functionality. It describes reconfigurable communication entities called Plugs. These Plugs are an implementation of `Distributed Proxy`.

References

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

[2] Pedro Sousa and António Rito Silva. Naming and Identification in Distributed Systems: A Pattern for Naming Policies. In *Conference on Pattern Languages of Programs, PLoP '96*, Allerton Park, Illinois, September 1996.

[3] 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.

[4] 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.

[5] Douglas C. Schmidt. Acceptor and Connector: Design Patterns for Active and Passive Estabishment of Network Connections. In Dirk Riehle Robert Martin and Frank Buschman, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

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

[7] 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.

[8] 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*, Kloster Irsee, Germany, July 1997.

[9] 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*, Kloster Irsee, Germany, July 1997.

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

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