

Manager-Agent and Remote Operation: Two Key Patterns for Network Management Interfaces

Jean Tessier*

Rudolf K. Keller

Département d'informatique et de recherche opérationnelle
Université de Montréal
C.P. 6128, succursale Centre-ville, Montréal (Québec) H3C 3J7, Canada
voice: (514) 343-6782, fax: (514) 343-5834
e-mail: Jean.Tessier@att.com, keller@iro.umontreal.ca
www: <http://www.iro.umontreal.ca/~keller>

Abstract

Developing network management interfaces (NMIs) is a challenging task involving multiple software layers, application programming interfaces (APIs), specification languages and tools. In order to ease the job of NMI developers, we have developed Layla, a prototype application framework supporting Open Systems Interconnection (OSI) NMIs. Layla is based on a heterogeneous yet coherent system of design patterns that includes previously published patterns, new patterns taken from NMI standards, and a couple of Layla-specific patterns relevant in Layla's API. The paper gives a brief overview of the pattern language underlying the Layla framework and discusses in detail two key patterns of the pattern language, the Manager-Agent and the Remote Operation patterns. The former pattern captures the regrouping of resources under the supervision and control of a responsible entity, whereas the latter encompasses clients that need to invoke operations on remote objects as if they were local.

Keywords: Network Management Interface, API (Application Programming Interface), OSI (Open Systems Interconnection), Manager-Agent pattern, Remote Operation pattern, Layla.

1 Introduction

Network management systems are used to control and monitor the components of distributed systems such as communication networks, where many different subsystems need to communicate with one another. Network management is a challenging task in that it usually requires remote access to widely distributed information from various sources. Operations have to be performed on large numbers of system components. Moreover, the access interface to the components can greatly vary, depending on their nature, type, and manufacturer. We define a *network management interface* (NMI) as the upper layers of a network management system, comprising the application layer, the application programming interface, and part of the presentation layer (cf. [TS92]). The lower layers, which usually depend heavily on the execution platform at hand, are thus not part of the NMI.

This work was in part funded by the Ministry of Industry, Commerce, Science and Technology, Quebec, under the IGLOO project organized by the Centre de Recherche Informatique de Montreal, by Teleglobe Canada Inc., and by the National Sciences and Research Council of Canada.

* Author's contribution is part of his Master thesis research at Université de Montréal. Author's current affiliation: AT&T Laboratories, Advanced Technologies Division, Holmdel, NJ.

Figure 1 illustrates a sample distributed system (network) under management. The system consists of two workstations communicating through a switch. Each workstation has a communication port that is attached to each end of the connection path, which is contained in the switch. The management system includes a management console that has access and control over all the components of the network through a symbolic representation provided by the NMI. Note that the NMI includes a number of communication stacks to access the various components.

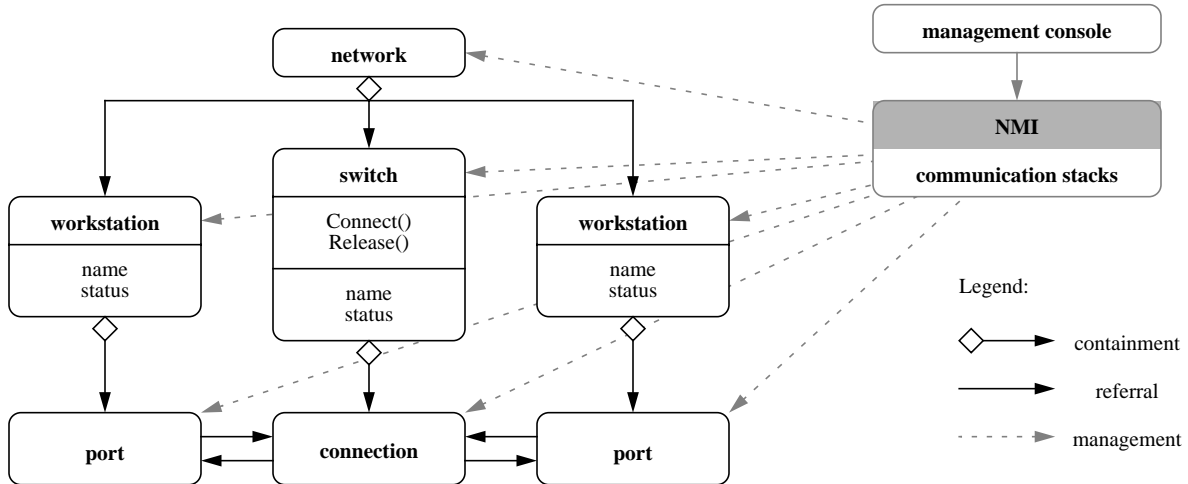


Figure 1: Sample network managed by network management system.

International standardization bodies have produced various tools for defining network management systems and their NMIs. Among the most advanced tools are the *Common Management Information Service (CMIS)* of *Open Systems Interconnection (OSI)* [TS92], and the *Simple Network Management Protocol (SNMP)* of the Internet [Ros91]. Whereas CMIS, along with *CMIP*, its protocol for information exchange between systems, is based on the object-oriented paradigm, SNMP uses tables not unlike the tables used in the relational model of databases. However, SNMP is moving towards the object-oriented paradigm, with its new version *SNMPv2* embodying some notion of inheritance.

As part of our work, we have built *Layla*, a prototype pattern-based framework for implementing NMIs in C++ [TKB96, Tes96]. We wanted to leverage off commercial implementations of standardized network management protocols, and therefore came up with a number of wrapper classes that encapsulate the specific details of any particular protocol engine. *Layla* was built for OSI NMIs and therefore includes provisions for the object-oriented nature of CMIS, which is not necessarily found in other protocols for network management. At an early stage in the development of *Layla*, we decided to take an approach based on design patterns [GHJV94, Pre94]. The resulting framework is a heterogeneous system of design patterns that includes previously published patterns and patterns adapted to NMIs, as well as new patterns taken from NMI standards and a couple of basic patterns relevant in *Layla*'s application programming interface (API). The patterns are implemented as a system of C++ classes that form the framework.

In this paper, we first give an overview of the pattern language underlying the *Layla* framework. Then, we focus on two of its key patterns, the Manager-Agent* and Remote Operation patterns, and compare them to related patterns. Next, detailed descriptions in template format of both the Manager-Agent and the Remote Operation patterns are provided. Finally, we draw a few conclusions.

* Throughout the paper, names of design patterns are underlined.

2 Towards a Pattern Language for Network Management Interfaces

The Layla framework is built upon a heterogeneous system of design patterns (see Figure 2) that includes previously published patterns, new patterns taken from NMI standards and a couple of Layla-specific patterns relevant in Layla's API. We believe that this system of patterns can serve as a basis for a pattern language for NMIs.

In Figure 2, strategic patterns [Sch96] are placed at the top, and the more generic but less critical tactical patterns are depicted in the region below the separator line (dashed line). Arrows stand for use-relationships, indicating that the originating pattern uses the destination pattern for the functionality mentioned on the label of the arrow. Below, we briefly discuss the role of the different patterns in our pattern language and outline their interaction.

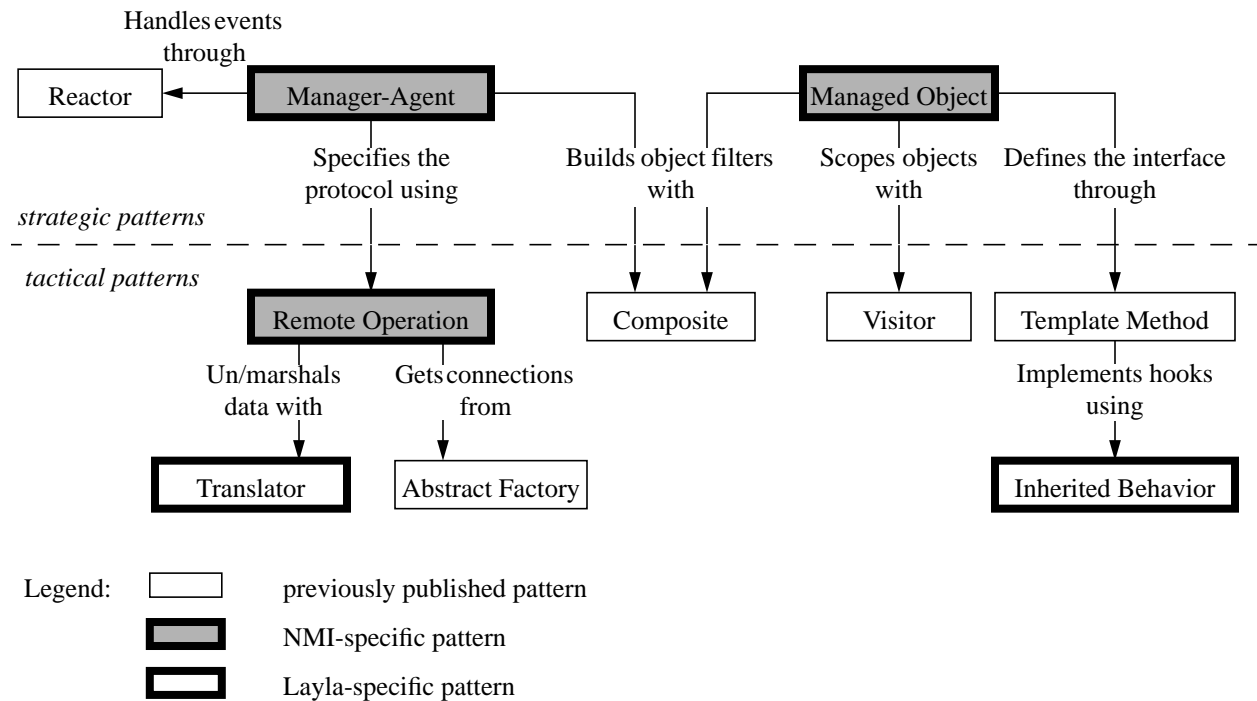


Figure 2: Pattern language of Layla.

Strategic patterns:

Manager-Agent: Managed resources are grouped into agents that are accessed by managers who perform management operations. Agents are responsible for monitoring their resources and notifying managers of exceptional behavior.

Reactor: Each process, whether a manager or an agent, has an event processing loop that drives the application and processes the stimuli received from the management system (cf. [Sch95]).

Managed Object: Agent NMIs represent their resources as a MIB (management information base), a collection of discreet manageable items. Each such items has a public interface that encapsulates the details of the actual resource.

Tactical patterns:

Remote Operation: The interactions between managers and agents are seen as operations that are executed across the management network by sending and receiving messages.

- Translator: The messages used in Remote Operation must be translated back and forth between user data and whatever formats are used by the underlying transmission mechanism.
- Abstract Factory: Generic connections are created by objects that have an intimate knowledge of the underlying transmission mechanism.
- Composite: A MIB is a complex structure of interrelated managed items. In other words, it is an *aggregate* of simpler components. Note that in CMIS, complex structures can be passed as parameters for operations in order to filter the MIB and identify target managed objects.
- Visitor: Special objects traverse the MIB in order to isolate specific targets of management operations.
- Template Method: In CMIS, each managed object has a standard interface that is inherited from a common ancestor but is implemented differently for each class of managed object.
- Inherited Behavior: As each subclass of managed objects implements the public interface differently, the behavior of the superclasses must be encapsulated so that the integrity of the object model is not compromised. This pattern lets a method in a subclass build upon the behavior of the equivalent method in the superclass.

NMIs interact with one another using the Manager-Agent pattern. Each NMI can either play a manager or an agent role. The management protocol between managers and agents is described using the Remote Operation pattern, which itself uses the Translator pattern for the marshalling and unmarshalling of arguments, and Abstract Factory to obtain connections between NMIs. NMIs handle management events using the Reactor pattern.

Agent NMIs use the Managed Object pattern to model their resources, using Template Method and Inherited Behavior to implement the required call-back methods. The Composite pattern can be used to represent complex filters used to screen the targets of operations, with the screening itself being handled by the Visitor pattern.

The Manager-Agent and Remote Operation patterns, governing a large part of our prototype framework, can be considered key patterns of an NMI pattern language and are the subject of the rest of this paper. The Managed Object, Translator, and Inherited Behavior patterns are discussed in [TKB96, Tes96]. The remaining patterns of the language are described in [GHJV94], except for Reactor, which is discussed in [Sch95].

3 Discussion of Manager-Agent and Remote Operation Patterns

Among the patterns published in the literature, the Broker [BMR⁺96], the Forwarder-Receiver [BMR⁺96], and the Master-Slave [Bus95, BMR⁺96] patterns, together with the traditional client-server model, are probably most closely related to our key patterns (some further relationships to other patterns are discussed in the respective templates in Section 4). In order to compare the above four patterns with our key patterns, we have devised a number of criteria:

- Model of Interaction: Whether each participant assumes a specific role for the whole duration of its existence (static) or whether a participant may assume different roles during execution (dynamic).

Cardinality/Direction of Interaction: Whether interactions always occur between two participants (1:1), whether they are organized as broadcast (1:n), or as concentration (n:1), or whether they may occur between any number of participants at any given time (m:n). Additionally, whether such interactions require unidirectional or bidirectional communication.

Control of Interaction: Whether one side always initiates interactions (superior-subordinate) or whether both sides may initiate interactions (peer-to-peer).

Importance: Whether the pattern is of great importance within the overall system (strategic), or rather addressing some small-scale task (tactical) [Sch96]. (We are aware that this criterion may be difficult to apply, depending on the framework at hand.)

Applying these criteria to our list of patterns leads to the table presented below (Table 1). We have rated the importance of each pattern according to what we felt would be their general importance in typical frameworks.

| Design Pattern | Roles during Interaction | Cardinality/Direction of Interaction | Control of Interaction | Importance |
|--------------------------------|--------------------------|--------------------------------------|-----------------------------|------------------|
| traditional client-server | static | 1:1 bidirectional | superior-subordinate | strategic |
| <u>Broker</u> | dynamic | 1:n bidirectional | superior-subordinate | strategic |
| <u>Forwarder-Receiver</u> | static | 1:1 unidirectional | superior-subordinate | tactical |
| <u>Master-Slave</u> | static | 1:n and n:1 bidirectional | superior-subordinate | tactical |
| <u>Manager-Agent</u> | <i>dynamic</i> | <i>1:n and n:1 bidirectional</i> | <i>peer-to-peer</i> | <i>strategic</i> |
| <u>Remote Operation</u> | <i>static</i> | <i>1:1 bidirectional</i> | <i>superior-subordinate</i> | <i>tactical</i> |

Table 1: Comparison of Manager-Agent and Remote Operation to other patterns.

The major differences between the Manager-Agent patterns and the other patterns are the dynamic nature of its interaction model and the peer-to-peer nature of its control of interaction. A process that acts as an agent in respect to a given manager process can in turn act as a manager for other agents. The managed system can be managed through layers of agent processes where the level of granularity is refined as we go deeper. But even with fixed roles, both sides can initiate asynchronous communication with one another, something that is not explicitly represented in the other patterns. According to the classification of Gamma et al. [GHJV94], Manager-Agent is of type “class structural,” while according to the classification of Buschmann et al. [BMR⁺96], it is an “architectural framework” that provides “access to objects” through “abstraction.”

Note that the Manager-Agent pattern is quite different from the Mediator [GHJV94] pattern, where the manager would correspond to the “mediator” and the agents to the “colleagues.”

For one thing, the agents in Manager-Agent do not collaborate among themselves. Rather, the management system is partitioned between managers and agents and no intra-partition communication usually occurs. Furthermore, the manager supports hierarchical control, rather than a collaboration mechanism. The control logic contained in the manager goes beyond the coordination scope of a mediator.

The Remote Operation pattern allows for bidirectional communication, whereas the Forwarder-Receiver pattern only handles unidirectional communication. The Remote Operation pattern offers a simple and elegant way to encapsulate interprocess communication in a client/server system, where a request in one direction is typically followed by a response in the opposite direction. The pattern is generic enough to decouple the application from the actual means of communication, which can in turn include other design patterns such as a pair of Forwarder-Receiver, or even another implementation of Remote Operation. According to the classification of Gamma et al., it is a “class structural” pattern, while according to the classification of Buschmann et al., it is a design pattern that “guides communication between objects” through “encapsulation.”

4 Detailed Descriptions of Manager-Agent and Remote Operation

MANAGER-AGENT PATTERN

Class Structural

Intent

Regroup resources under the supervision and control of a responsible entity. This decouples the resources being managed from the management functions operating on them.

Motivation

This pattern applies in the following context: Imagine a large system of collaborating components that cooperate in order to provide a service, such as a telecommunication network. Such a system is often managed from a central console which controls all the components in the system (cf. Figure 1) and which is typically called a *manager*. The problem is to present a unified management interface for such systems, or for subsets thereof.

This problem has to deal with the following forces:

- There can be a large variety of management functions to be performed, increasing the complexity of the manager.
- There can be an extremely large number of components to manage, which might overload any single manager.
- The components can present various types of management interfaces, functionalities, and semantics, which need to be unified somehow.
- It might be desirable to have a portion of the system manage itself in an autonomous manner.

The proposed solution is to first isolate the management functionalities in one or more *manager* objects. Their task is to handle all the management aspects of the system. Then, the system is partitioned into a number of subsystems, which are controlled by individual *agents*. The agents take responsibility for a group of “related” resources (functional nature, logical relationship, manufacturer, etc.). Moreover, they represent their respective subsystems to the

managers of the system. Managers and agents can use a single protocol to communicate with one another. Managers may interact with multiple agents within the system in order to handle a given management task. Similarly, an agent may report to more than one manager. This solution can be recursively applied to a large subsystem in order to simplify the agent in charge of it. The agent then becomes a manager for its subsystem.

The result of this solution is twofold. First, the management policy of the system is decoupled from the system itself. Furthermore, management responsibilities may be delegated to subsystems, resulting in a “divide and conquer” approach to system management.

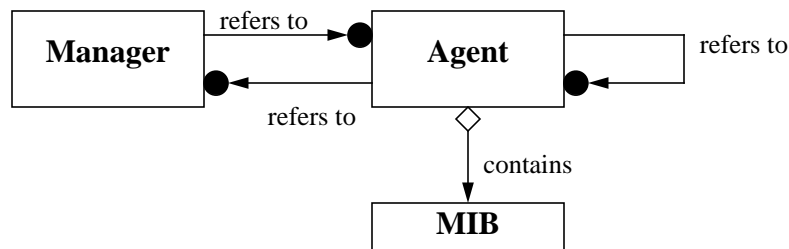
The set of resources grouped under an agent are referred to as a *Management Information Base*, or MIB.

Applicability

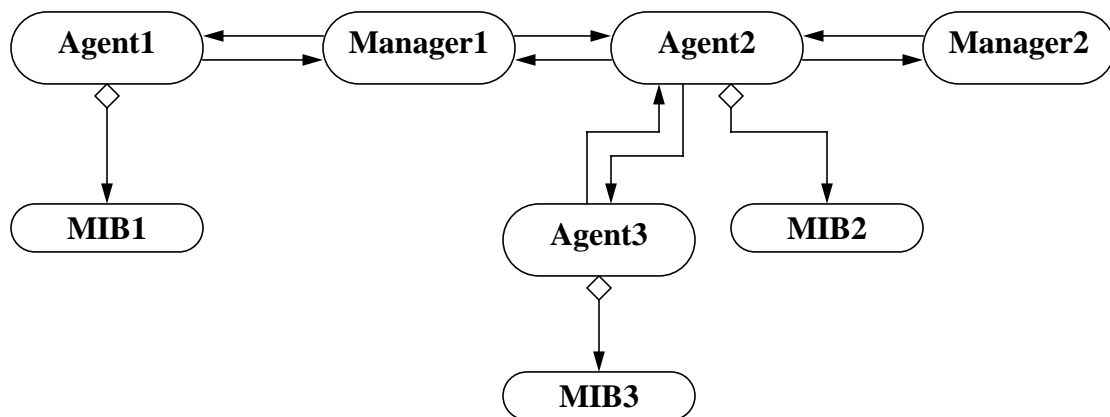
Use the Manager-Agent pattern when there is a large number of low-level resources that need to be handled similarly. The Manager-Agent works best when:

- It is impossible to modify the actual interface of the resources so as to make them conform to a single management protocol. There are many reasons why this could be so, some of them technical, political, or even financial.
- The resources are so numerous and show so much variation that the introduction of individual adapters is not a viable solution.

Structure



A typical object structure might look like this:



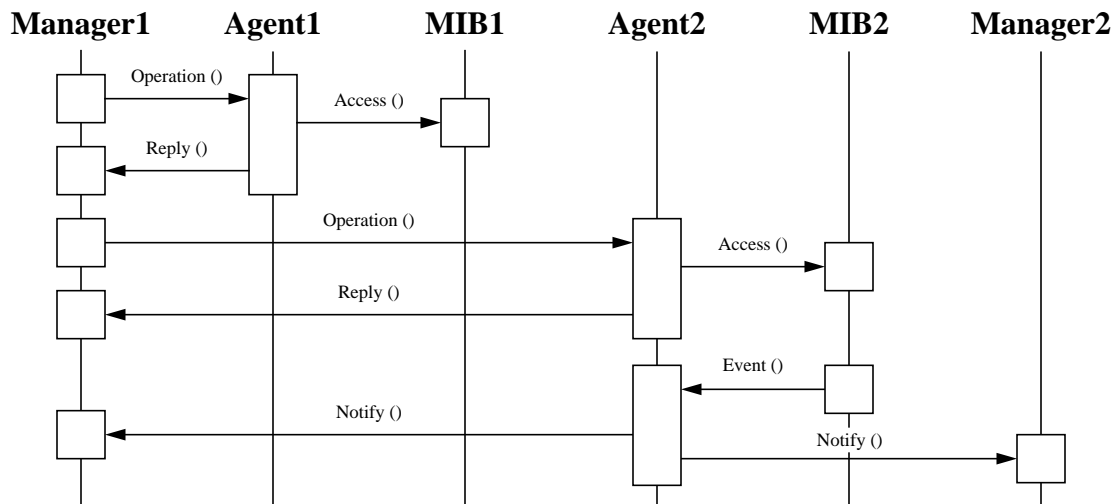
Participants

- **Manager** (Manager1, Manager2)
 - Responsible for one or more management aspect of the whole system.
 - Sees the systems as a set of Agents.

- **Agent** (Agent1, Agent2, Agent3)
 - Responsible for managing a subset of the resources in the system.
 - The resources may include other Agents.
- **MIB** (MIB1, MIB2, MIB3)
 - The set of resources represented by an Agent.

Collaborations

- The Manager dispatches management operations to one or more Agents in order to perform its (the Manager's) management functions.
- The Manager collects event reports coming from one or more Agents, and may take some action in response to them (depending on its management function).
- The Agent handles management operations on its resources on behalf of one or more Managers. Those resources may include other Agents.
- The Agent monitors its MIB and/or agents for reportable events and forwards them back to one or more Managers. The Agent may also take some action in response to those events. The resources may include other Agents.



In the above example, MIB1 and MIB2 are coupled to Agent1 and Agent2, respectively, and interact solely with them. Note that Manager1 performs operations on both Agent1 and Agent2. Note also that Agent2 reports to both Manager1 and Manager2. Managers and Agents are thus more loosely coupled than the MIBs with their respective Agents.

Consequences

The Manager-Agent pattern has the following benefits and liabilities.

1. *The Manager and the Agent use a single protocol to communicate.* This encapsulates the proprietary protocols used by the resources in the Agent and simplifies the implementation of the Manager. The Manager is then able to communicate with any and all the resources in the system, regardless of their origin or nature. Note however that this single protocol, in order to incorporate all the proprietary functionalities, can become quite complex and might introduce a lot of overhead.
2. *The hierarchy of the system is expressed through the Agents.* This means that the Managers do not need to maintain their own maps of the system. Modifications in one area of the system need only be reflected in the relevant Agents, not in the Managers that manage the

system.

3. *This pattern is a variation of the client-server architecture.* This architecture closely relates to the client-server architecture, where the agent plays the role of the server and the manager that of the client. But in traditional client-server interactions, all the interactions originate in the client-side, not the server-side. Here, both the agent and the manager can be the originator of an interaction at any given time.

Implementation

Here are some useful techniques for implementing the Manager-Agent pattern.

1. *The Agent comprises a level of indirection for accessing its resources.* The Agent, in order to portray accurately the MIB, needs either to maintain a special database or to apply a set of translation rules. In the first case, the special database may stock the proprietary values in a format that is legal for the management protocol. This database needs to be kept synchronized with the actual resources. In the second case, translation rules are applied dynamically in order to fulfill requests from a Manager. In both cases, there is an added layer of processing when accessing management data, and this layer can degrade performance when manipulating large amounts of data. The selected mechanism needs to be implemented with care.
2. *The MIB must be described by architecture-neutral means.* The specification must capture the details of the MIB in a manner that is as generic, yet as precise as possible. This is to ensure the interoperability of distributed components, regardless of their underlying origin or implementation, and to promote a unified view of the system.
3. *The management protocol must be architecture-neutral.* Again, this is to ensure the interoperability of distributed components, regardless of their underlying implementation. The protocol between managers and agents needs to be flexible enough to handle various types of data, regardless of the actual implementation of the data.
4. *The relationships between managers and agents must be maintained adequately.* One solution is for each manager and each agent to maintain a list of its collaborating opposites, but such a solution is inflexible. Instead, the Mediator [GHJV94] pattern may be used to maintain all these relationships. Moreover, the Remote Operation pattern may be used to provide location transparency. Finally, the Broker [BMR⁺] pattern can be used to provide both at the same time.

Known Uses

This pattern can be found in both CMIS, a part of OSI [TS92], and SNMP, a part of Internet [Ros91]. Both are flavors of NMIs. CMIS has been successfully implemented in two independent NMI frameworks: Layla and OSIMIS, the latter being from the University College of London [PKMB95].

Related Patterns

Observer: The Observer [GHJV94] pattern could be used to implement the notification messages that travel from the agents to the managers. In current implementations however, all data regarding notification is carried in the initial message, in order not to overburden the system with such data transfers during later execution.

Remote Operation: The set of interactions between agents and managers (the communication protocol) can be defined using instances of the Remote Operation pattern.

- Mediator:** The Mediator [GHJV94] pattern can be used to handle the communication between the managers and the agents. In this way, each side doesn't have to possess any knowledge of the other. New managers or new agents can be inserted in the system without affecting the already existing managers and agents.
- Broker:** The Broker [BMR⁺] pattern adds the concepts of service negotiation and location transparency, which are not explicitly addressed by the Manager-Agent pattern. A broker can be inserted between the managers and the agents. The broker can then direct manager operations and agent notifications to their proper destination. Operations can be routed to equivalent agents without involving the emitting manager in the selection process, or they can even be broadcast to multiple agents. The same goes for agent notifications. The broker can also take care of aggregating multiple responses. In essence, it incorporates characteristics of both the Remote Operation pattern and the Mediator pattern.
- Reactor:** The event loop that drives the managers and the agents can be viewed as an instance of the Reactor [Sch95] pattern.
- Composite:** Complex parameters in the operations between managers and agents can be viewed as instances of the Composite [GHJV94] pattern.

REMOTE OPERATION PATTERN

Class Structural

Intent

Enable clients to invoke operations on remote objects as if they were local. This pattern decouples the client from the network calls needed to access the remote object.

Also Known As

Remote Procedure Call

Motivation

This pattern applies in the following context. In a distributed system, such as a client/server architecture, the client of an operation is often removed from where that operation's implementation actually resides. The client must access the implementation through a communication network.

The problem is to make a remote operation invocation appear exactly the same as a local operation invocation, both to the client and to the implementation.

This problem has to deal with the following forces:

- Network calls are inherently more complex than local calls.
- Operating through the network is much more unreliable than operating through local process memory.
- The client and the server of an operation want to be shielded from network-specific details.

The proposed solution is to encapsulate all the network interactions in stub objects, both on the client and the server side. The stubs communicate together using connection and message

objects appropriate to the network. The client and the server interact locally with their respective stubs. The result is a system where the invocation of a remote operation is decoupled from the network interactions needed to carry it out.

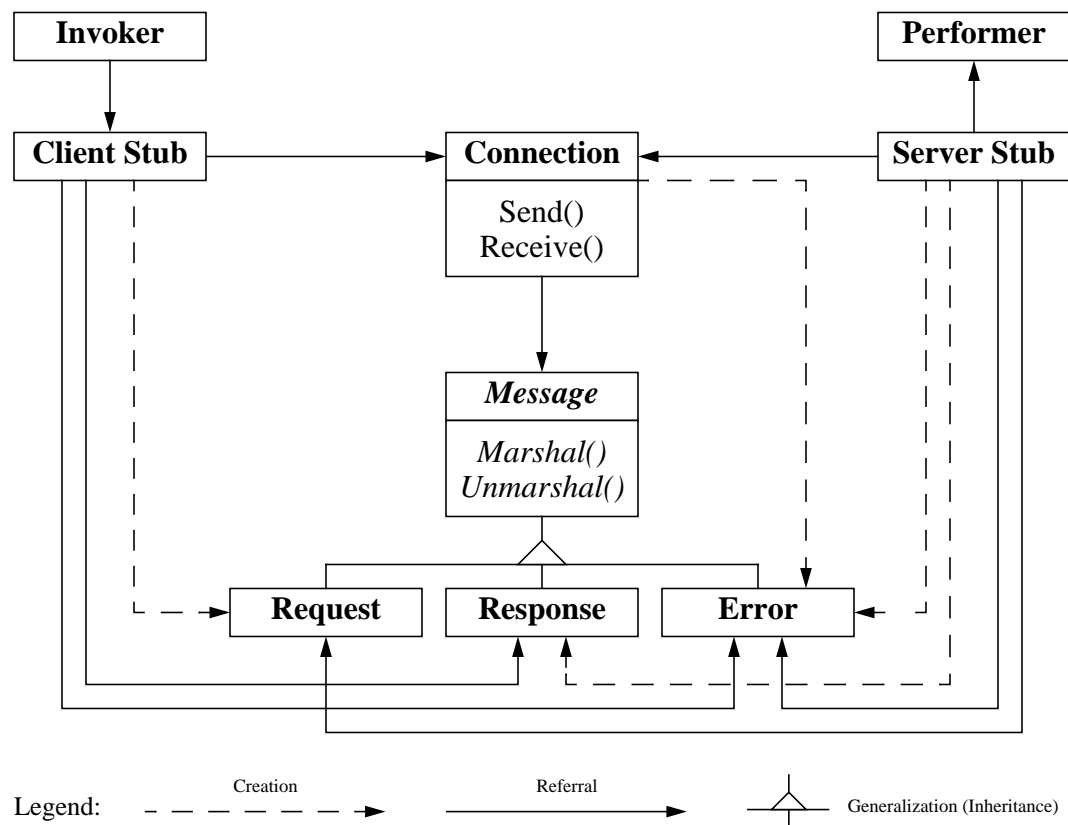
The Remote Operation pattern is a refinement of the Proxy [GHJV94] pattern in that it further decouples the network interactions from the invoker and the performer of the operation, in respect to both the client and the server sides of Remote Operation.

Applicability

The Remote Operation pattern works best when:

- Entities in a distributed system need to invoke operations on each other in a consistent way, regardless of their location in the distributed system.
- The entities are spread across multiple networks using potentially different network protocols, and remote access needs to be network-independent.
- The distributed system provides location transparency, that is the invoker does not know the exact location of the performer.

Structure



Participants

- **Invoker**
 - Invokes an operation on a local object, the Client Stub.
 - Sees the invocation as a regular local call.
- **Performer**
 - Implements the operation to be performed.
 - Gets called by the Server Stub and sees the invocation as a regular local call.

- **Client Stub**
 - Originator in the message passing capabilities required for the remote invocation.
 - Knows which remote host contains the implementation of the operation, or uses a trader to locate an appropriate host.
- **Server Stub**
 - Recipient in the message passing capabilities required for the remote invocation.
- **Connection**
 - Established between the Client Stub and the Server Stub.
- **Message**
 - Presents a *generic* interface to the Connection.
 - Provides the marshalling and unmarshalling of data.
- **Request**
 - Contains the parameters of an operation.
- **Response**
 - Contains the response from an operation.
- **Error**
 - Contains data pertaining to error conditions in the network, the Performer, or even the Client Stub.

Collaborations

The diagram below shows the sequence of events in a typical use of the Remote Operation pattern. The different participants collaborate as follows:

- The Client Stub creates a Request message, marshals the invocation arguments into the Client Stub, and sends the message via the Connection to the Server Stub. The Client Stub also unmarshals the elements of Response messages and disposes of them appropriately.
- The Server Stub, upon receipt of a Request message issued by the Client Stub, unmarshals its arguments and disposes of them appropriately. It then invokes the proper operation on the Performer and collects the results. It creates Response or Error messages depending on the result of the invocation on the Performer.
- The Connection transports message instances back and forth.
- The Connection can create Error messages when there are problems in the underlying communication network, the server side, or the client side.

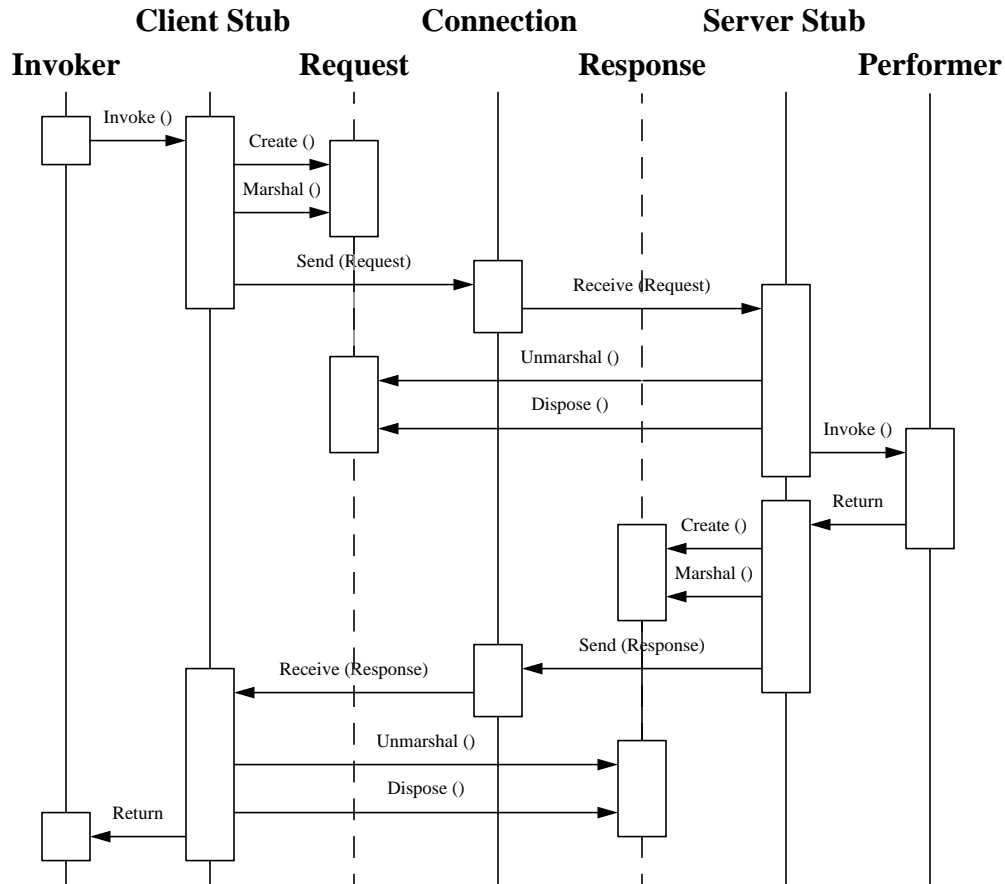
Consequences

The Remote Operation pattern has the following benefits and liabilities.

1. *The Invoker is shielded from the network.* The Invoker does not know that the invocation is actually sent across the network to a remote Performer. The Invoker doesn't even know where the operation is actually performed. To the Invoker, the whole business is no different from invoking a method on a local object, in this case the Client Stub.
2. *A remote invocation takes longer than a local invocation.* The time it takes to fulfill a remote invocation depends on the amount of overhead in the Messages and the bandwidth of the Connection.
3. *It is hard to pass object references across process boundaries.* The Client Stub and the Server Stub must handle pointers, as well as object instantiation, with extra care so that the semantics of the operations are preserved.
4. *The Performer has a broader audience.* This pattern makes the operations offered by the Performer available to all applications on the network, provided they have the proper Client

Stub available to them.

5. *The network is unaware of the specifics of the operations.* The Connection simply transports Messages between applications, without regard to their actual content and/or purpose.
6. *Network errors can cause Messages to be lost.* Some error-correction mechanism is required in order to make the whole design more reliable. See the implementation comments below.
7. *A protocol can be seen as a set of Remote Operations.* A group of related Remote Operations can be seen as a communication protocol between communicating entities.



Implementation

Here are some useful techniques for implementing the Remote Operation pattern.

1. *Synchronous vs. asynchronous operations.* In synchronous operations, the execution of the invoker is suspended until either a response or an error status is received. This is the easiest to implement. In asynchronous operations, the invoker is free to perform other processing while waiting for the results. In asynchronous operations, some mechanism must be provided allowing for the cancellation of invoked operations that have not yet terminated.
2. *Layering for communication mechanisms.* The Connection between the Invoker and the Performer can itself be an instance of the Remote Operation pattern. The process can be applied even further, creating stacks of stubs on both the client and the server sides. Each layer builds upon the ones below it and offers a more sophisticated service to the ones above it. The lower layers encapsulate the basic elements of the communication system

while the upper layers offer a high-level abstraction of that communication system. An example of this can be seen in the OSI Reference Model [TS92] for data communication and in the layering of TCP over IP [Ros91].

3. *More than one performer may exist in the system.* In this case, the stubs can use a “trader” to select an appropriate performer for the operation.
4. *Some retransmission method needs to be implemented in case some request (or result) gets lost.* However, it is then possible that more than one request makes its way to the remote host, which might cause the operation to be performed more than once. Certain operations are unaffected by the number of times they get invoked. Assigning a constant to a variable is a good example. Such operations are called *idempotent*. But adding a constant to a variable is a non-idempotent operation. Care must be taken that such operations are not performed more than once. This gives rise to the following categories of semantics:
 1. **At most once:** these semantics guarantee that the operation is either performed once or not at all. This involves putting invocation IDs in the request messages so that duplicates can be treated properly.
 2. **Exactly once:** these semantics guarantee that the operation will be performed, and so only once. This is the toughest semantics to guarantee as it also involves recovering from network failures.
 3. **At least once:** these semantics guarantee that the operation will be performed, but maybe more than once. Of course, this is only good for idempotent operations. They are the simplest semantics to implement. These semantics are sometimes called idempotent semantics or broadcast semantics (as they can involve broadcasting the request until a response is received).

Sample Code

In the following example, the `Message` class is introduced to define the abstract messaging interface. This interface is then implemented in the subclasses `Request`, `Response`, and `Error`.

```
class Message {
public:
    virtual void *Marshal() const;
    virtual void Unmarshal (void *);
};
class Request: public Message {
public:
    void *Marshal() const;
    void Unmarshal (void *);
};
// Implementation of Marshal () and Unmarshal ()
// Classes Response and Error
```

The `Connection` class can then be implemented using the abstract messaging interface and the primitives of the underlying communication mechanism.

```
class Connection {
public:
    void Send(Message *m);
    Message *Receive();
};
```

```

void Connection::Send(Message *m) {
    void *tmp = m->Marshal();
    // Send tmp using communication primitives
}
Message *Connection::Receive() {
    int error;
    void *tmp;
    // Receive tmp using communication primitives
    // Set error to non-zero if an error has occurred
    Message *r = error ? new Error: new Response;
    r->Unmarshal(tmp);
    return (r);
}

```

The `ClientStub` class, reproduced below, uses both the `Connection` class and the messaging interface in order to realize the Remote Operation pattern. We leave it up to the client to distinguish error cases from successful invocations. This can be achieved through the use of metatyping information or members of the abstract `Message` class.

```

class ClientStub {
public:
    Message *Operation(Request *r);
protected:
    Connection *connection;
};
Message *ClientStub::Operation (Request *r) {
    connection->Send (r);
    return (connection->Receive());
}

```

A similar approach can be taken to implement the `ServerStub` class. However, that class will need to receive a `Request`, invoke the proper methods on the `Performer`, and send the `Response` back.

Known Uses

This pattern is being used in OSI to define a number of interprocess protocols, such as CMIP and the protocol suite for X.400 electronic mail. As such, it is implemented in Layla, since the latter is based on CMIS and CMIP. Other implementations can be found in Sun's and HP's RPC libraries, among others.

Related Patterns

Manager-Agent: The Remote Operation pattern may be used to define the operations that can be invoked between agents and managers.

Translator: The Remote Operation pattern may use the Translator [TKB96] pattern in the Client and Server Stubs in order to translate between the client/server data and the formats supported by the `Connection` (the marshalling and unmarshalling of data).

Abstract Factory: The Remote Operation can use an Abstract Factory [GHJV94] pattern to create connection instances without needing prior knowledge of the actual transmission mechanism being used.

Conclusion

In this paper, the Manager-Agent and the Remote Operation patterns, two key patterns for NMIs, have been presented. The former pattern captures the regrouping of resources under the supervision and control of a responsible entity, whereas the latter encompasses clients that need to invoke operations on remote objects as if they were local. These two patterns form the core of a pattern language which has been developed for NMIs and implemented in the Layla application framework. They differ in several ways from the list of patterns mentioned in Section 3, complement them, and address the specifics of NMIs.

Our experience suggests that the pattern-based architecture of Layla makes NMI development considerably easier. We contend that the pattern language upon which Layla is built will be helpful for other NMI framework builders and NMI application developers alike.

References

- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, 1996.
- [Bus95] Frank Buschmann. The master-slave pattern. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 29, pages 133–142. Addison-Wesley, 1995. (Reviewed Proceedings of the First International Conference on Pattern Languages of Programming (PLoP'95), Monticello, IL, 1994).
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [PKMB95] George Pavlou, Graham Knight, Kevin McCarthy, and Saleem Bhatti. The OSIMIS platform: Making OSI management simple. In Adarshpal Sethi, Yves Raynaud, and Fabienne Faure-Vincent, editors, *Integrated Network Management IV*, pages 480–493. Chapman and Hall, 1995.
- [Pre94] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.
- [Ros91] Marshall T. Rose. *The Simple Book: An Introduction to Management of TCP/IP-based Internets*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.
- [Sch95] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 29, pages 529–545. Addison-Wesley, 1995. (Reviewed Proceedings of the First International Conference on Pattern Languages of Programming (PLoP'95), Monticello, IL, 1994).
- [Sch96] Douglas C. Schmidt. A family of reusable design patterns for application-level gateways. *Theory and Practice of Object Systems*, 1996. John Wiley and Sons. Special Issue on Patterns and Pattern Languages. To appear.
- [Tes96] Jean Tessier. An application framework for OSI network management interfaces. Master's thesis, Université de Montréal, Montreal, Quebec, Canada, April 1996. In French.
- [TKB96] Jean Tessier, Rudolf K. Keller, and Gregor v. Bochmann. Layla: A pattern-based framework for network management interfaces. Technical Report GELO-63, Université de Montréal, Montreal, June 1996. Accepted by guest editors of Comm. of the ACM, currently under review by regular editors of CACM.
- [TS92] Adrian Tang and S. Scoggins. *Open Networking with OSI*. Prentice-Hall, Inc., 1992.