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: {tessier, 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 projected 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 details and discusses 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. 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.

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 Layla development, 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 projected from NMI standards and a couple of basic patterns relevant in Layla's API interface. The patterns are implemented as a group 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 discuss in depth two of its key patterns, the Manager-Agent and Remote Operation patterns, and compare them to related patterns. Full descriptions in template format of both the Manager-Agent and the Remote Operation patterns are provided in appendices A and B, respectively.

2 Towards a Pattern Language for Network Management Interfaces

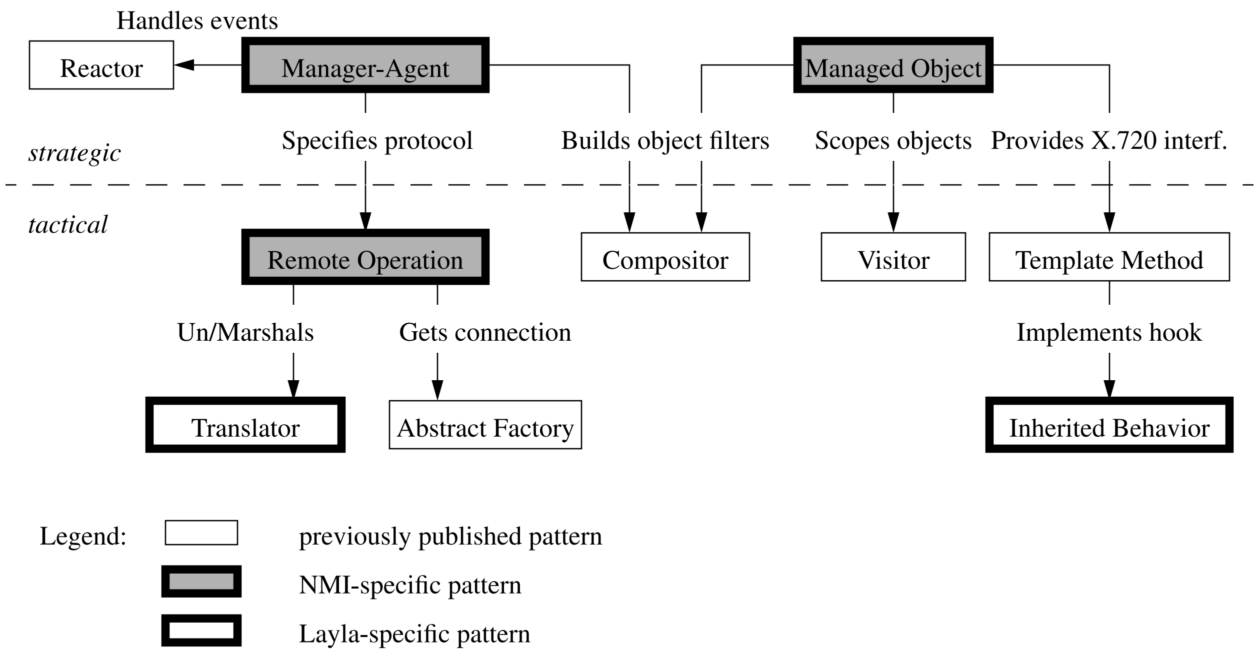
The Layla framework is built upon a heterogeneous system of design patterns (see figure below) that includes previously published patterns, new patterns projected from NMI standards and a couple of Layla-specific patterns relevant in Layla's Application Programming Interface (API). We believe that this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

In the figure, strategic patterns [Sch95a] are placed at the top, and the more generic but less critical tactical patterns are depicted in the region below the separator 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.

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 receives the stimuli coming from the management system (cf. [Sch95b]).



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 through the sending and reception of messages.

Translator: The messages used in Remote Operation must be mapped 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.

Compositor: The MIB is a complex structure of interrelated managed items. In other words, it is an aggregate of simpler components. Also, in CMIS, complex structures can be passed as part of operations 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: In CMIS, 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.

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 Compositor pattern can be used to represent complex filters used to screen the targets of operation, with the screening itself being handles 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 (see also Appendices A and B). The Managed Object, Translator, and Inherited Behavior pattern are discussed in [TKB96, Tes96]. The remaining patterns of the language are described in the design pattern literature [GHJV94].

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 pattern [Bus95], 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 (Appendices A and B)). In order to compare these 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).

Nature of interaction: Whether interactions always occur between two participants (1:1), whether they are organized as broadcasts (1:n) or as concentrations (n:1), or whether they may occur between any number of participants at any given time (m:n).

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 typically "of great importance within an integrated whole," (strategic) or rather "related to small-scale actions serving a larger purpose" (tactical). (We are aware that this criteria is difficult to apply and may heavily depend on the framework at hand.)

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

The Manager-Agent patterns differs from the other patterns by 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," according to Buschmann et al. [BMR+96], it is an "architectural framework" that provides "access to objects" through "abstraction."

The Remote Operation pattern, implicitly allowing for bidirectional communication, offers a simple and elegant way for encapsulating interprocess communication. 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 Forwarder-Receiver, or even another implementation of Remote Operation. According to Gamma et al., it is a “class structural” pattern, and according to Buschmann et al., it is a “design pattern” that “guides communication between objects” through “encapsulation.”

Design Pattern	Model of Interaction	Nature of Interaction	Control of Interaction	Importance
traditional client-server	static	1:1	superior-subordinate	strategic
Broker	dynamic	1:n	superior-subordinate	strategic
Forwarder-Receiver	static	1:1	superior-subordinate	tactical
Master-Slave	static	1:n and n:1	superior-subordinate	tactical
Manager-Agent	***dynamic***	***1:n and n:1***	***peer-to-peer***	***strategic***
Remote Operation	***static***	***1:1***	***superior-subordinate***	***tactical***

References

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. 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.

[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 Yves Raynaud Adarshpal Sethi and Fabienne Faure-Vincent, editors, *Integrated Network Management IV*, pages 480–493. Chapman & 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.

[Sch95a] Douglas C. Schmidt. A family of reusable design patterns for application-level gateways. *Theory and Practice of Object Systems*, 1995. John Wiley & Sons. Special Issue on Patterns and Pattern Languages. To appear.

[Sch95b] 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.

[Ste90] W. Richard Stevens. *UNIX network programming*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

[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. Submitted for publication.

[TS92] Adrian Tang and S. Scoggins. *Open Networking with OSI*. Prentice-Hall, Inc., 1992.

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 is motivated by the need to manage remote resources from a centralized management console. In a large system, the resource can be varied and may come from different manufacturers, providing widely different management interfaces. Each management console has to be able to communicate with each and every single component, and therefore needs to incorporate all the proprietary interfaces used in the network. In addition, each management console has to maintain its own hierarchical view of the system.

To alleviate these problems, we introduce the concept of an agent. The resources are grouped together according to some criteria (functional nature, logical relationship, manufacturer, etc.) and put under the control of an agent. The agent has the task of representing the resources under its jurisdiction to the rest of the management system. There might be more than one agent in the system, with each being responsible for a segment of the total resources.

On the other hand, the management console is seen as a manager that handles some management function across the entire system. There might be more than one manager in the system, with each being responsible for a specific management aspect. The agents can also take on some of the responsibilities for managing their resources, in essence becoming mini-managers for their subsystems.

Note that a manager can interact with multiple agents within the system in order to handle a management task. Similarly, an agent can report to more than one manager when conditions require managers being notified.

By making it the agent's responsibility to handle the various custom protocols found among the resources, the interactions between managers and agents can be handled by a single management protocol. This greatly simplifies the implementation of managers. Additionally, if the distribution of agents in a system follows the hierarchical division of the system, then all the managers simultaneously gain the same hierarchical access to the resources. This second benefit also greatly simplifies the implementation and maintenance of the managers. The proper distribution of the agents limits the disruption caused by the introduction of new resources, where such disruption no longer needs to be system-wide. The effect of new introductions is localized to the agent(s) who will have jurisdiction over the new resources.

The resources grouped under an agent form a *Management Information Base*, or MIB. The manager sees the agent as providing access to the MIB through a management protocol. The MIB can be specified using a neutral language to further set it apart from the peculiarities of its resources.

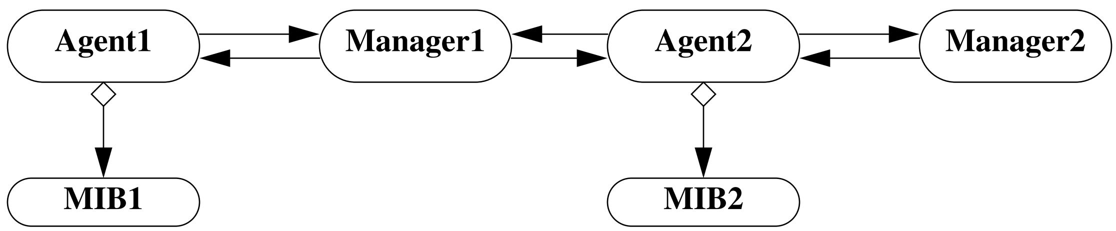
Applicability

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 implementation of adapters is not a viable solution.

Structure

A typical object structure might look like this:

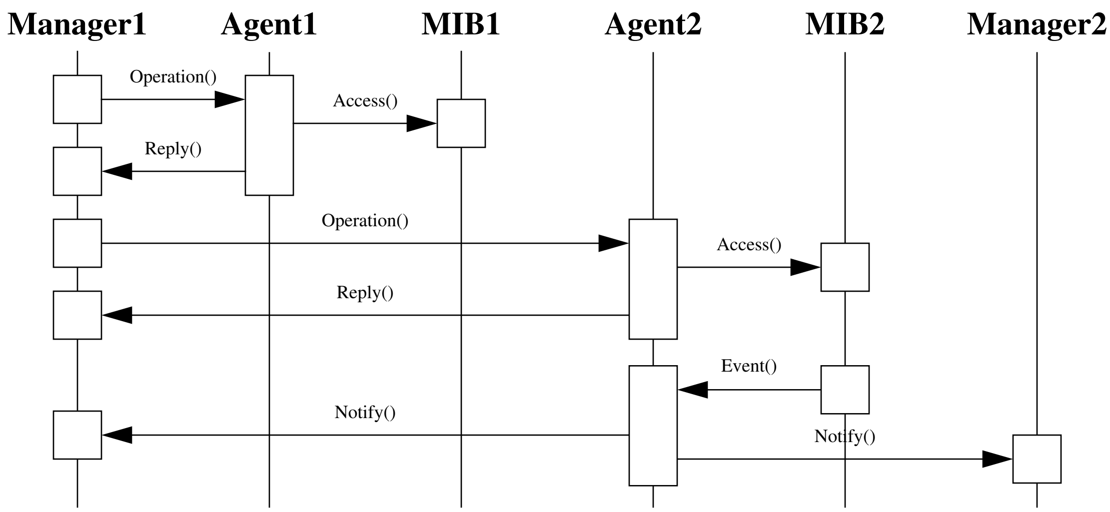


Participants

- **Manager**
  - Responsible for one or more management aspect of the whole system.
  - Applies management operation at a system-wide level.
- **Agent**
  - Responsible for managing a subset of the resources on behalf of the managers in the system.
  - Applies management operations on specific resources.
- **MIB**
  - The set of resources represented by the agent.

Collaborations

- The Manager dispatches management operations to agents in order to perform management functions.
- The Manager collects event reports coming from the agents, and may take some action in response to them.
- The Agent handles relevant management operations on the MIB on behalf of the Agent.
- The Agent monitors the MIB for reportable events, and may take some action in response to them.



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.
2. *The hierarchy of the system is expressed through the Agents.* This means that the Manager does not need to maintain its own map of the system. Modifications in one area of the system need only to be reflected in the relevant Agent, not in the Managers that manage the system.
3. *This pattern is an enhanced form of 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 provides indirect access to the resources.* The Agent, in order to portray accurately the MIB, needs to maintain either a special database or a set of translation rules. The special database can 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. On the other side, translation rules can be 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 carefully implemented.
2. *The MIB must be described by architecture-neutral means.* It is desirable that the specification of the MIB be written using a specification language that is not biased towards any specific architecture. The specification should express the details of the MIB in a manner as neutral and as precise as possible. This way, we ensure the

interoperability of distributed components, regardless of their underlying implementation.

3. *The management protocol must be architecture-neutral.* It is also desirable that the management protocol not be biased towards any specific architecture. This way, we ensure the interoperability of distributed components, regardless of their underlying implementation.

Sample Code

This pattern is derived from the literature on NMIs. The complexity of the code required for NMIs makes it difficult to extract significant examples of the pattern. The interested reader is referred to the literature on NMIs [TS92, Ste90].

Known Uses

This pattern can be found in both OSI and Internet flavors of NMIs. It has been successfully implemented in two independent NMI frameworks: Layla and OSIMIS of the University College of London [PKMB95].

Related Patterns

Observer: The Observer pattern can be used to implement the notifications that travel from the agents to the managers. In current implementations however, all data regarding the notification is carried in the initial message so as to preserve the resources of the management network.

Master-Slave: The Master-Slave pattern deals with access by a client to a number of identical and/or replicated objects, all under the supervision of the master. The client only sees the interface to the master and is not aware of the slaves. In contrast, the Manager-Agent pattern deals with selective access to a number of dissimilar resource objects. Furthermore, the manager targets specific objects through the agent, breaking the encapsulation offered by the Master-Slave pattern.

Broker: The Broker 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 even broadcasted to multiple agents. The same goes for agent notifications. The broker can also take care of aggregating multiple responses.

The Broker pattern is fundamentally asymmetrical in that it is the client who initiates communications, whereas the server is passively listening. In the Manager-Agent pattern, both parties can (and do) initiate communications in an asynchronous manner.

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

Reactor: The event loop that drives the managers and the agents can be viewed as an instance of the Reactor pattern.

Compositor: Complex parameters in the operations between managers and agents can be viewed as instances of the Compositor pattern.

APPENDIX B: 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

Distributed systems are complex. The resources are distributed across a network and the remotely located entities can only talk to each other by interacting with the network. This means that when an entity wishes to invoke an operation on a target entity, it must first find out if the target entity is local (in which case the operation can be directly invoked). If not, the invoker must find the target entity on the network and send one or more messages to its host, asking it to invoke the operation on the invoker's behalf and then send the results back.

Some retransmission method needs to be implemented in case the original request gets lost. But 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 also 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).

The Remote Operation pattern encapsulates the network interactions in a group of classes and hides those interactions from both the invoker and the performer of an operation. The invoker talks to a local entity that acts as a proxy for the remote performer of the operation. At the same time, a similar piece of software resides with the performer and acts as a proxy for the remote invoker of the operation. These local pieces of software are called stubs, the client stub residing with the invoker and the server stub residing with the performer. The stubs communicate together so that operations invoked on the client stub result in the server stub invoking the actual operation on the performer and that results flow in the opposite direction.

The stubs handle all the interactions with the network. Thanks to them, the invoker sees the remote operation as a local call, and the performer sees the operation as being invoked by a local entity. Both invoker and performer are freed from having to handle network interactions, including the marshalling and unmarshalling* of operational data.

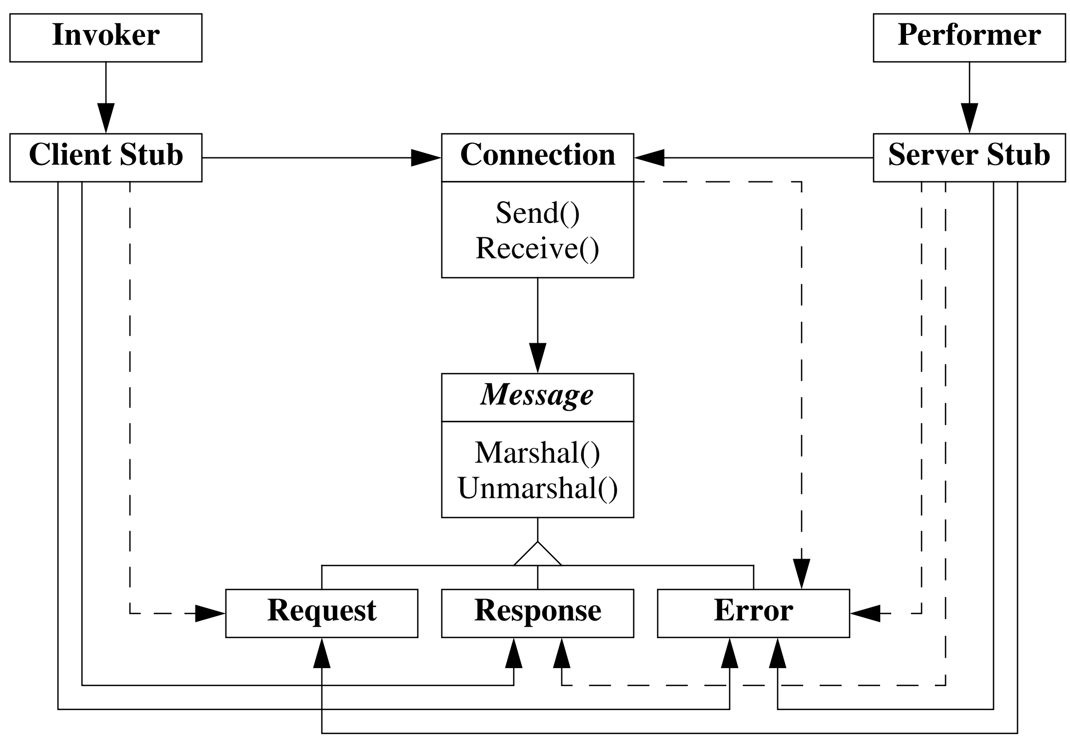
The Remote Operation pattern is a refinement of the Proxy pattern. It further decouples the network interactions from the invoker and the performer of the operation.

Applicability

The Remote Operation pattern works best when:

- Entities in a distributed system need to invoke operations on one another 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.
- More than one performer exists in the system, in which case the stubs can use a trader to select a given performer for the operation.

Structure



Participants

- **Invoker**
  - Invokes an operation on a local object, the Client Stub.
  - Sees the invocation as a regular local call.

---

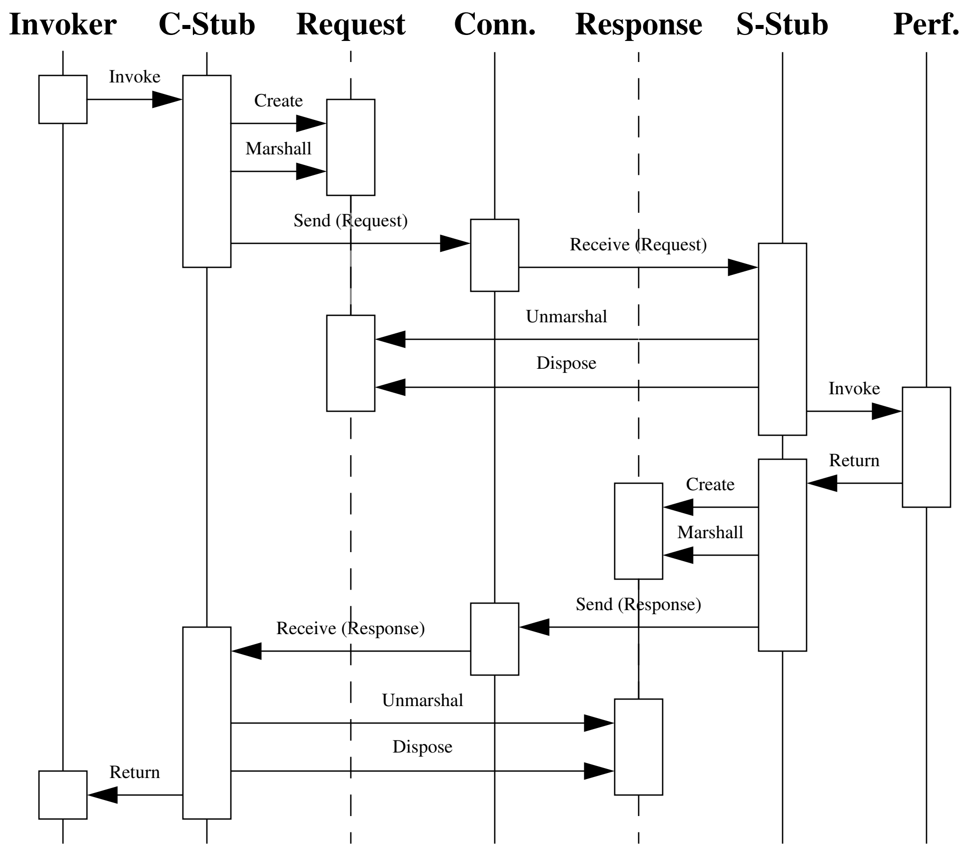
* Marshalling deals with taking data in an application-specific format and putting it in a format that is suitable for transmission. This new format is no longer application-dependent. Unmarshalling is a reverse form of marshalling.

- **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 in order to locate one.
- **Server Stub**
  - Recipient in the message passing capabilities required for the remote invocation.
  - Receives invocation requests from remote clients and returns Response or Error messages depending on the result of the invocation on the Performer.
- **Connection**
  - Established between the Client Stub and the Server Stub.
  - Ferries Messages back and forth between its end-points.
  - Generates Error messages when exceptional conditions occur in the network.
- **Message**
  - Presents a generic interface to the Connection.
  - Implements 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 or in the Performer.

Collaborations

The following figure shows the sequence of events in a typical use of the Remote Operation pattern.

- The Client Stub creates a Request message and marshals the invocation arguments in it. It also unmarshals responses from Response messages and disposes of them appropriately.
- The Server Stub unmarshals arguments from Request messages and disposes of them appropriately. It then invokes the proper operation on the Performer and collects the results. It creates Response or Error messages appropriately and marshals the results into them.
- The Connection ferries message instances back and forth, and can create error messages when there are problems in the underlying communication network.



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. *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.
3. *The network is unaware of the operations.* The Connection simply ferries messages between applications, without regard to their actual content and/or purpose.
4. *An invocation can result in the operation being performed zero, one, or more times.* If the Connection loses the initial Request, the operation may end up never being performed. The Connection must either create an Error to notify the Client Stub (and then the Invoker), or contain retransmission capabilities.
5. *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 can be provided to cancel 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 complex 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 for data communication and in the layering of TCP over IP.

Sample Code

This pattern is derived from the literature on NMIs. The complexity of the code required for NMIs makes it difficult to extract significant examples of the pattern. The interested reader is referred to the literature on NMIs [TS92, Ste90].

Known Uses

This pattern can be found in both OSI and Internet flavors of communicating processes. It has been successfully implemented in Layla, as well as a number of RPC libraries.

Related Patterns

Forwarder-Receiver: The Forwarder-Receiver pattern abstracts the particular communication mechanism from the components involved. It can be used to implement each direction of the connection object used by the invoker and the performer of the Remote Operation pattern. In Forwarder-Receiver, the clients must know the location of the servers with which they want to communicate. In Remote Operation, this is encapsulated in the client stub and the server stub, along with the necessary code to send and receive messages.

Proxy: The client stub can be seen as a remote Proxy, but the structure of the Remote Operation pattern decouples further the communication mechanism from both the invoker and the performer.

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

Translator: The Remote Operation pattern can use the Translator pattern in the client and server stubs in order to map the user data to the format supported by the connection (the marshalling and unmarshalling of data).

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