

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.

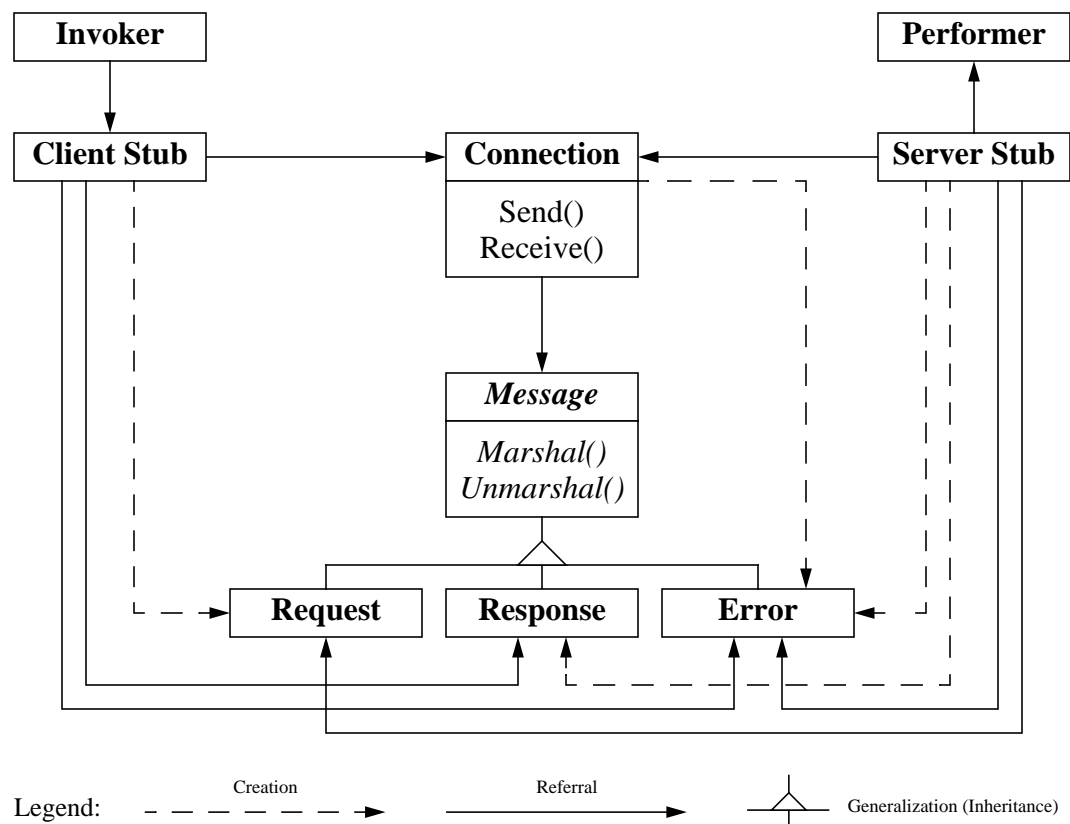
The Remote Operation pattern is a refinement of the Proxy [GHJV94] pattern in that it further decouples the network interactions from the invoker and the performer of the operation, in respect to both the client and the server sides of Remote Operation.

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

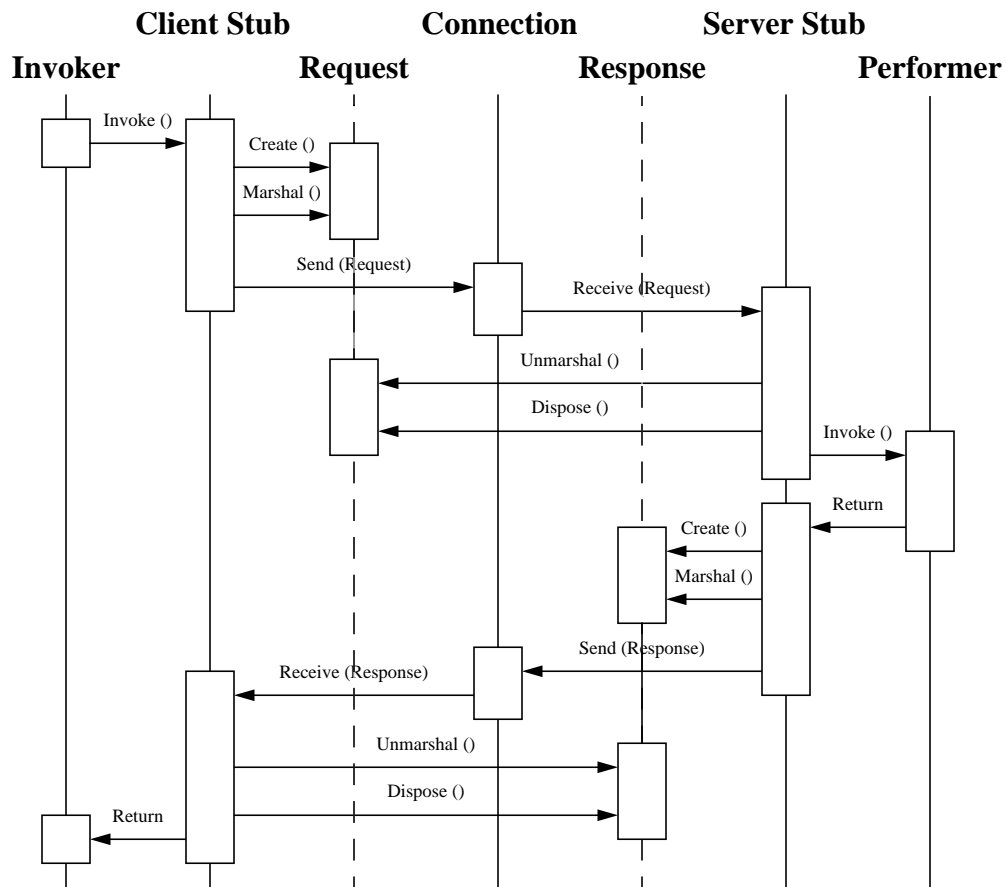
The diagram below shows the sequence of events in a typical use of the Remote Operation pattern. The different participants collaborate as follows:

- 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 [TK97] 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.

References

- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. 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.
- [Tes96] Jean Tessier. An application framework for OSI network management interfaces. Master's thesis, Université de Montréal, Montreal, Quebec, Canada, April 1996. In French.
- [TK97] Jean Tessier and Rudolf K. Keller. Manager-Agent and Remote Operation: Two key patterns for network management interfaces. In *Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences*, Washington University Department of Computer Science, wucs-97-07, pages 4.8.1-4.8.14, February 1997.
- [TS92] Adrian Tang and S. Scoggins. *Open Networking with OSI*. Prentice-Hall, Inc., 1992.

Contacts

<http://www.iro.umontreal.ca/~keller/Layla>

Jean Tessier, AT&T Laboratories, Advanced Technologies Division, Holmdel, NJ.

Jean.Tessier@att.com

Rudolf K. Keller, Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal (Québec) H3C 3J7, Canada.

keller@iro.umontreal.ca