# Securing the Broker Pattern

Patrick Morrison and Eduardo B. Fernandez

Dept. of Computer Science & Engineering, Florida Atlantic University,
777 Glades Road, Boca Raton, FL 33341-0991
morrison@fau.edu, ed@cse.fau.edu

## *Abstract*

We consider how to add security to the *Broker* distribution pattern. We do so by examining how systems using *Broker* are secured in practice, and by then revising the pattern to reflect this knowledge. The results are presented as a new pattern, *Secure Broker*. This paper serves as a model to illustrate how a pattern can be made secure by adding specific functions.

## *1. Introduction*

The *Broker* pattern [Bus96] describes a way to structure distributed software by decoupling components that interact through remote service invocations. It does not, however, describe how to protect communications between the distributed components. Secure distributed communications have obvious application in many problem domains, notably commerce, medical, financial and military applications. Access control is also necessary to enforce authorization restrictions and is not considered here.

There is a great deal of design pattern advice, e.g. [Bus96, Kir04, Sch00] on how to build distributed systems. There is also a great deal of experience with securing distributed systems, e.g., [And01b, Kau02]. However, much of the experience gained in securing distributed systems has not worked its way back into the design patterns. This paper bridges that gap by updating the *Broker* design pattern with knowledge collected from looking at how systems built using *Broker* are secured. The paper produces a Secure Broker pattern and it illustrates one approach to securing patterns.

Section 2 summarizes the *Broker,* while Section 3 enumerates *s*ecurity issues with this pattern. In section 4, CORBA and .NET implementations of *Broker* are examined for their security attributes. The knowledge gained from these implementations is applied to the *Broker* pattern to evolve it into *Secure Broker*. Section 5, presents the *Secure Broker* pattern. We expect the reader to be acquainted with general aspects of distributed systems and security. It would be particularly helpful to be have access to the *Broker* pattern [Bus96].

## 2. Broker Described

The *Broker* architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions[Bus96].

Figure 1 shows its class diagram, relating Clients and Servers through a broker and proxies.

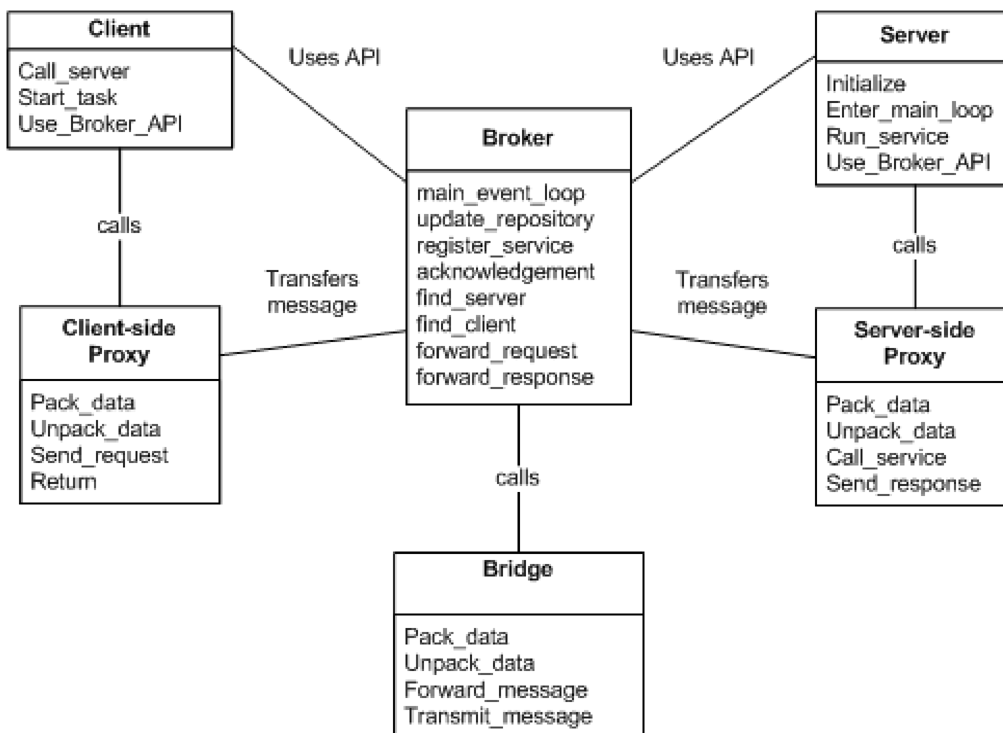


Figure 1: Broker Pattern class diagram [Bus96]

Proxies insulate their callers, Client and Server, from the implementation details of communications. The Bridge class implements a similar concept for communications between Brokers.

There are two basic use cases for Broker, illustrating its role in structuring transparent communications between clients and servers: Server Registration and Client Requests Service. These use cases are illustrated through sequence diagrams, in figures 2 and 3. See [Bus96] for details.

Three examples of Broker implementations are CORBA, DCOM, and the World Wide Web. In the later, each browser acts as a Broker enabling client applications to access web servers, which play the Server role.

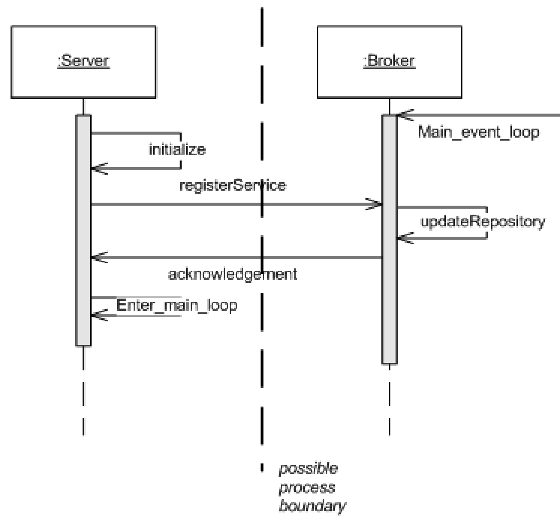


Figure 2: Server Registration.

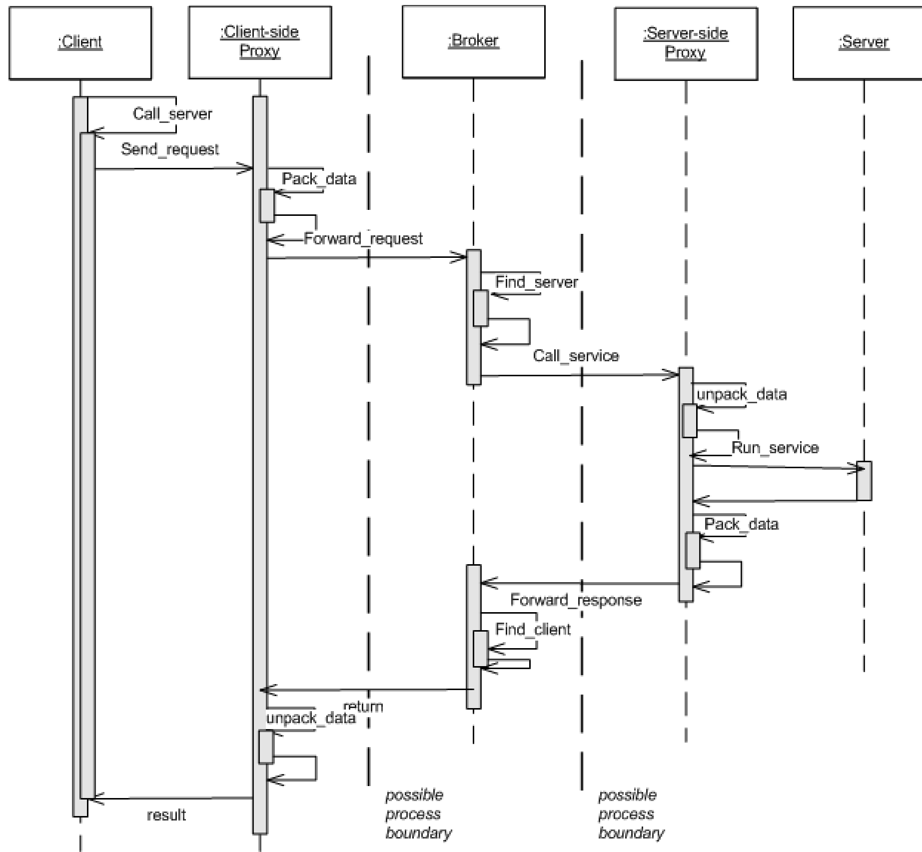


Figure 3: Client Requests Service.

## *3. Security Issues with the Broker Pattern*

*Broker* decouples communications from application concerns, but does not address security issues; if not addressed, these can compromise an application's usefulness.

### Attacks on Broker

Based on discussions in [And01b] and [Kau02] we classify attacks on the Broker's related components as Forgery, Betrayal and Denial of Service.

#### Forgery

If a rogue server can portray itself as valid to the Broker, it can appear to service client requests while also compromising client data, or perform a wide variety of other attacks on unsuspecting clients. Phishing is a form of forgery. Likewise, if a rogue Broker can portray itself as valid to Servers and Clients, it can do harm by recording traffic between clients and servers, substituting other clients and servers for valid ones, and so on. And if a client can forge its identity to a Broker, it can access services for which it does not have rights. There are a wide variety of attacks based on Forgery: redirection of traffic from official sites to forged sites; spamming while masking the source's destination; Cache Poisoning, where invalid entries are stored in the Broker's repository; and routing attacks, where traffic intended for one destination is sent to another.

#### Betrayal By Trusted Server

If a valid Server is compromised, but treated as valid by the Broker, it can do harm, in some of the ways outlined in Forgery, above.

#### Denial of Service

Without access control of the Broker's server repository, valid entries can be removed, and so they will not be accessible. And with access to the Broker's server repository, DOS attacks can be launched against member servers. By limiting the server's abilities to respond to requests, clients are disabled.

### Security Responsibilities

Therefore, in addition to Broker's role in decoupling communications from applications, the Secure Broker must:

- Protect Clients from illegitimate Servers and Brokers
- Protect Servers from illegitimate Clients and Brokers
- Protect Brokers from illegitimate Clients and Servers
- Allow for securing communications between its Clients and Servers

## *4. From Broker to Secure Broker*

Software Brokers mediate between their clients and the providers of services desired. For example, having an email server saves you from having to know the path between your machine and the machines of those with whom you'd like to exchange messages. The email server acts as a Broker. This decoupling of communication from function allows both clients and services to focus on their own roles.

You want to be sure that you can trust the Broker(s) you use. An email server that broadcast private messages, or allowed emails with forged authorship would be of limited use.

The approach for developing the *Secure Broker* from the *Broker* pattern is to evaluate several example Broker implementations that have considered security, namely CORBA and .NET Remoting. It is expected that this evaluation will identify common characteristics of solutions to security issues. It is also expected that these solution characteristics are valid in that they have been tested through implementation.

### 4.1 Example Implementation: CORBA

In order to be concrete about defenses, we choose an example *Broker* implementation, the CORBA Object Request Broker (ORB), to see how transactions are secured. To do this requires some introduction to CORBA's security architecture.

CORBA security explicitly defines the threats it is designed to address:

1. An authorized user of the system gaining access to information that should be hidden from him.
2. A user masquerading as someone else, directly or through delegation.
3. Security controls being bypassed.
4. Eavesdropping on a communication line.
5. Tampering with communication.
6. Lack of accountability due, for example, to inadequate identification of users.[OMG02, sect. 1.1.3]

This architecture can be mapped to Broker's defenses in the following way:

- [1,2,3] Protect Clients from illegitimate Servers and Brokers
- [1,2,3] Protect Servers from illegitimate Clients and Brokers
- [1,2,3] Protect Brokers from illegitimate Clients and Servers
- [4,5] Allow for securing communications between its Clients and Servers

We note that [6] indicates that an Auditing facility should be present.

## CORBA Security Architecture

CORBA, in its Security Service, approaches securing transactions by treating Clients, Servers and Brokers as *Principals* which are "a human user or system entity that is registered in and authenticated to the system" [And01a]. The distinguishing characteristic of a *Principal* is its Identity. There are several consequences of Identity: it makes the Principal accountable for its actions; it identifies the originator of a message; it identifies whom to charge for use of a system, and it allows access control/rights management to be defined. Principals may be granted "security attributes" *(Rights*). These attributes are used to determine access control for objects within the system. An object's collection of security attributes is known as its 'Credentials'. Authorization is implemented between Principals and objects through 'Security Context' objects, which carry the Identity and Credential information necessary to determine the calling Principal's rights for the called object.

The 'PrincipalAuthenticator' interface provides facilities for generating sets of Credentials and for generating Security Contexts, given a Principal, an object, and an access request. An object implementing 'Principal Authenticator' (called 'Vault' in the CORBA architecture) accepts a Principal's identity as an argument, and authenticates that Principal, returning its set of Credentials (Figure 6). Access Decision objects are responsible for binary (yes/no) access decisions, based on the applicable Principal, object, and Security Context. When a client makes a request of a server, both client and server proxies submit the request for evaluation according to security policy through these access decision objects (Figure 7). Note that no checking is done by the CORBA Broker; CORBA assumes that the proxies can be trusted.

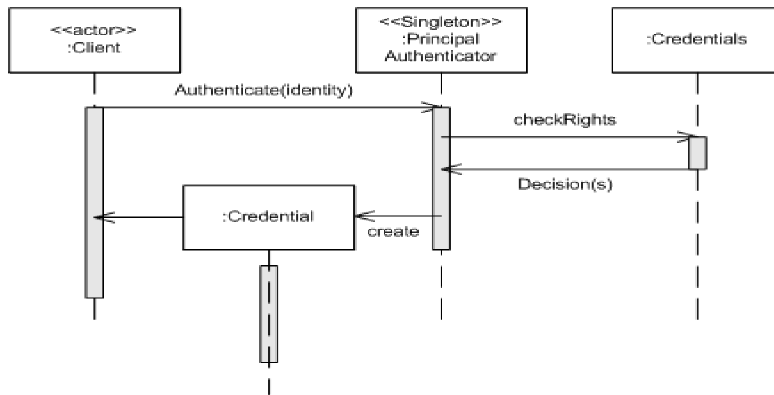


Figure 6: Subject Authentication, in CORBA

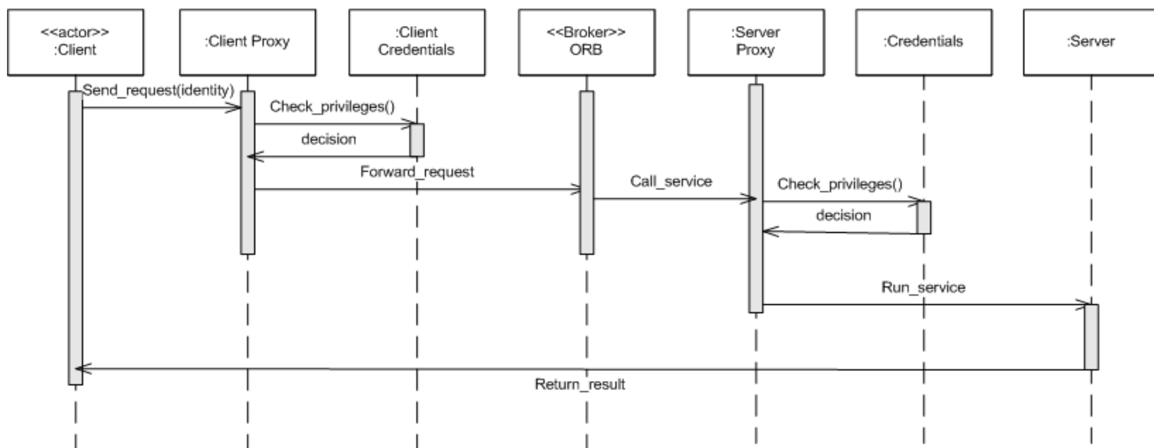


Figure 7: Secure object invocation, in CORBA

## 4.2 Example Implementation: .NET Remoting

.NET Remoting implements HTTP and TCP transport mechanisms ('Channels'). The .NET Remoting security architecture does not enumerate specific threats; rather, it provides a generic set of tools for authentication, authorization and confidentiality that must be adapted in an application's context. These tools, implemented through the GSS-API, include credentials to identify clients and servers, contexts in which these credentials are valid, and provisions for encrypted transport [MS04A].

Microsoft has illustrated how to use .NET Remoting to implement the *Broker* pattern [MS03A]. In considering .NET Remoting security, and how to apply it to *Secure Broker*, we look to the GSS-API [RFC 2743] for guidance.

## *5. Secure Broker*

*Secure Broker* amends *Broker* to provide secure interactions between distributed components.

### 5.1 Example

An organization uses an electronic messaging system, perhaps conferencing software, chat, or instant messaging. A group within the organization wants to arrange for private communications within the group. Members of the group should be able to exchange messages with each other that are not made known to the organization at large. Members have a variety of devices (laptops, PDA's, cell phones) that run the organization's messaging client.

### 5.2 Context

Distributed computing systems, homogeneous or heterogeneous, with independent cooperating components that must be secured.

### 5.3 Problem

In addition to *Broker*'s role in decoupling communications from applications, the *Secure Broker* must:

- Authenticate servers, clients and the Broker itself
- Allow servers, clients and the broker itself to authorize requests based on the requestor's identity.
- Provide means for securing messages and channels between clients, servers and the broker.

It must do this in a way that interfaces with an organization's security policies.

Forces affecting the pattern:

- Broker distributes objects, but distribution does not imply trust
- Client access to Servers may need to be restricted
- Server access to Clients may need to be restricted
- Trust for an intermediary (the Broker) can be established
- Brokering services should scale to large numbers of clients, servers.

## 5.4 Solution

Introduce identity, for authentication, as an aspect of components that must communicate securely. Provide Authorization facilities for components that must adapt and control their behavior based on whom is making the request. Implement rights assignment and management to govern how components may interact. Use Reference Monitors as ‘guards’ for components, that check rights and allow/deny access based upon a requestor’s rights.

## 5.5 Structure

The Secure Broker pattern adds three participants to Broker: Subject, Reference Monitor and Secure Channel (Figure 8).

**Subject** is an abstract class which amends the **Broker**, and **Client**- and **Server**-side proxies. It must be implemented by each component participating in a secure transaction. Its role is to provide identity to components participating in communication, and to allow components to authenticate each other. Identity management and creation is beyond the scope of description here, but it must be sufficient to uniquely identify components in the universe of possible interactions. Identity management must also be such that only an administrative entity can generate a valid identity.

The **Reference Monitor** authorizes participant requests. It is responsible for allowing and denying service requests based upon the identity of the requestor and the prevailing set of rights. Rights structure and configuration is an important topic, but is also beyond the scope of this paper. See, for example, [Fer01].

**Secure Channel** is responsible for encrypting traffic between components that may travel over links that are not limited to trusted components. For example, it may not be sufficient for a client and server to mutually authenticate and authorize a request, since the details of the request could be eavesdropped by listening to traffic between the client and server. In this case, a **Secure Channe**l should be used.

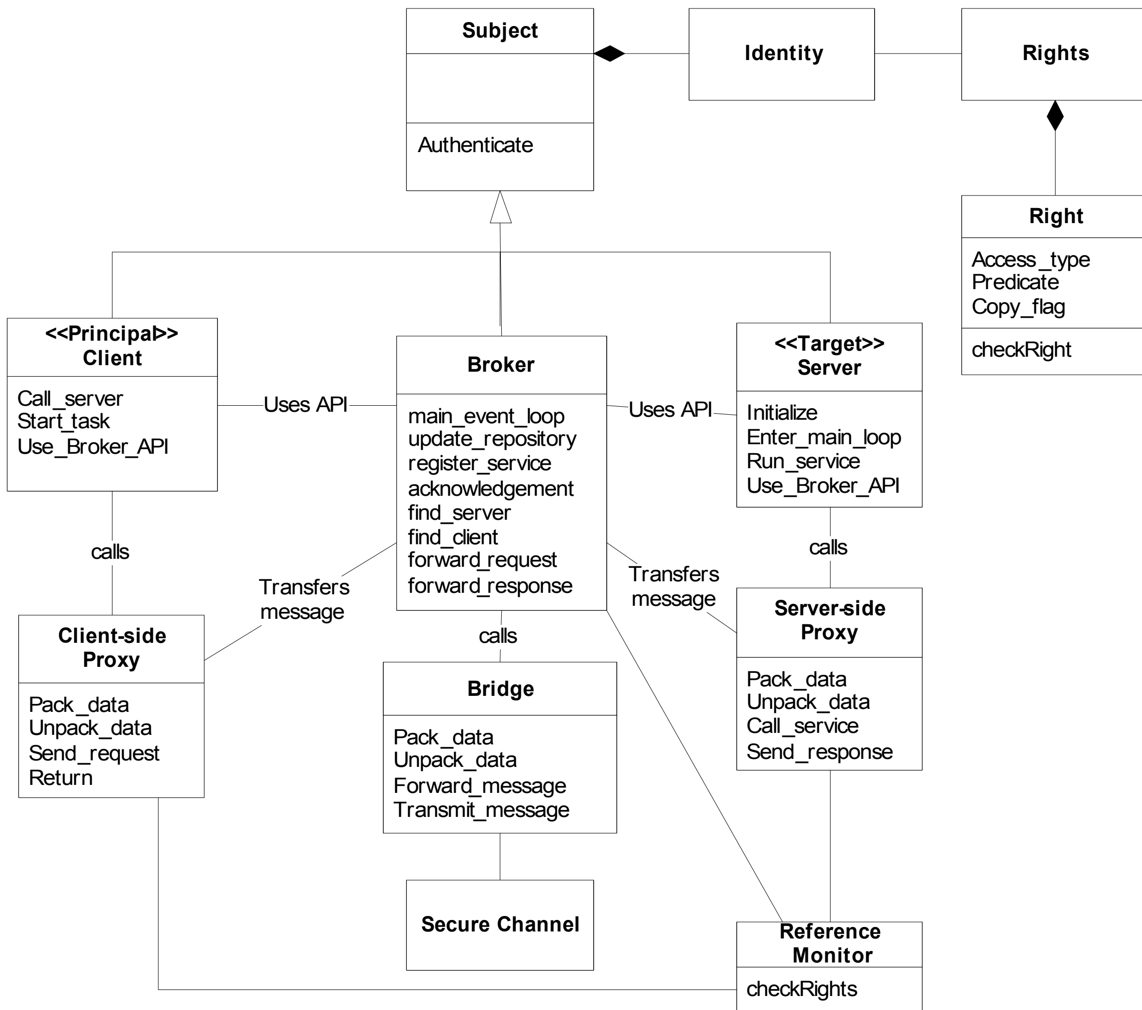


Figure 8: Class diagram for *Secure Broker*.

## 5.6 Dynamics

*Secure Broker* requires an additional use case, Subject Creation. Each client and server wishing to participate in secured communications must be assigned identity and rights.

### 5.6.1 Subject Creation

In contrast to *Broker,* where it is assumed that the Broker can be trusted, Servers, Clients and Brokers, in their roles as Principals, must be assigned identities and credentials in order to safeguard access. Therefore, we need a preliminary use case for each Principal (Subject), Subject

Creation, incorporating identity and rights assignment. The sequence diagram is shown in Figure 9:

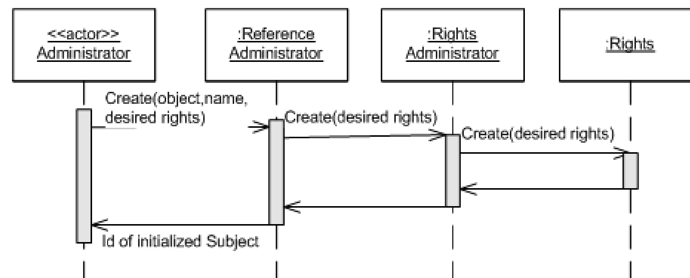


Figure 9: Subject Creation

Note that the implementation of this use case requires notions of administration and policy, something which we do not address directly. The key point is that each participant in *Secure Broker*, Client, Server, and Broker, will need to be authenticated to participate, and that each participant is assigned its Rights in the process of being authenticated.

### 5.6.2 Registration

Given a Server and Broker that have previously been authenticated, we have the following, updated, sequence diagram for Server Registration, Figure 10:

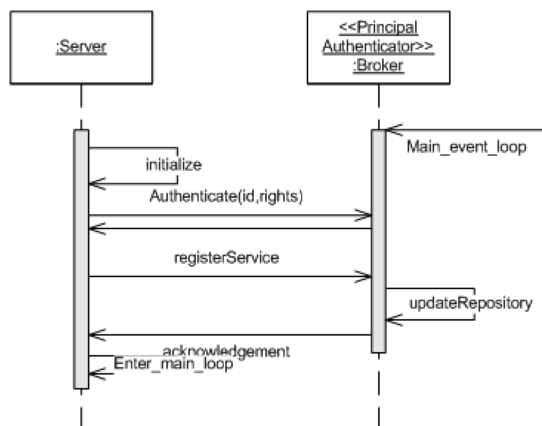


Figure 10: Secured Subject Registration

### 5.6.3 Client Requests Service

Given that the participating Client, Server and Broker have been previously authenticated and assigned credentials, service requests flow as indicated in the sequence diagram (Figure 11):

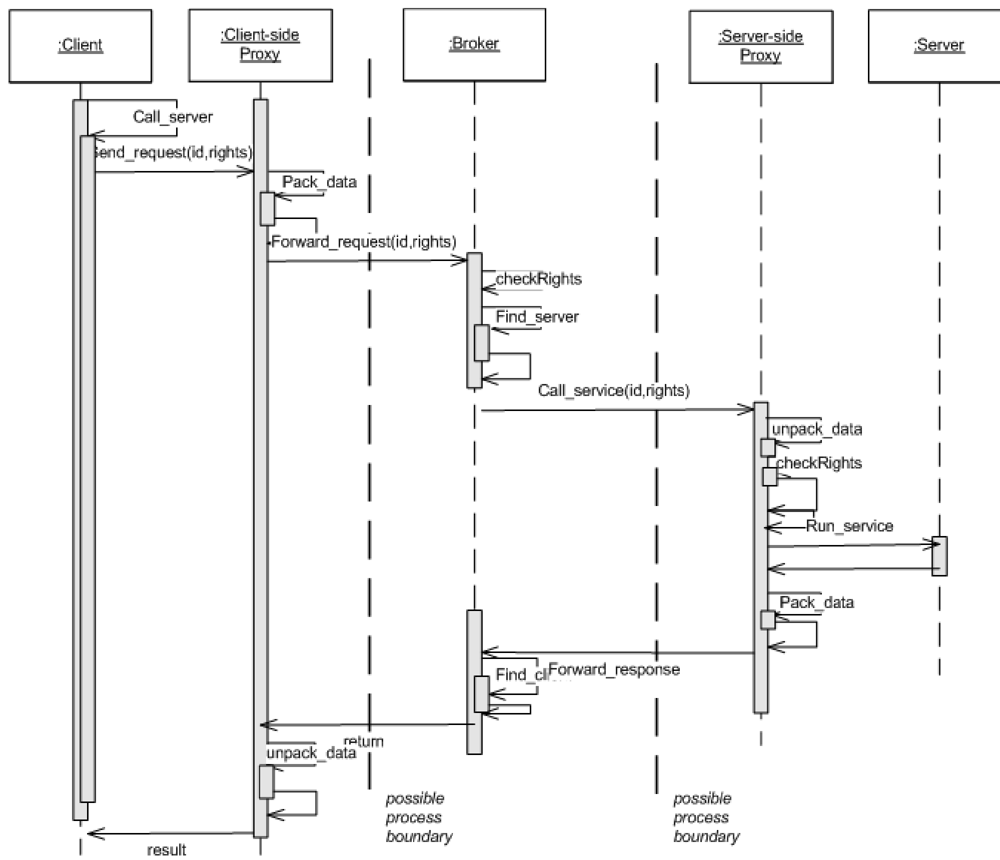


Figure 11: Secured Client Requests Service

## 5.7 Example Resolved

In the messaging example, each group member participating in the conversation is issued an identity for the system, together with rights to chat with each other member of the group. The messaging server is also assigned an identity, together with rights permitting each member of the group to send messages to each other member of the group. As client proxies send messages to the server, each message is checked by the Server's reference monitor (and possibly the server-proxy's reference monitor, depending on the configuration and security requirements). Eligible messages from authenticated participants are relayed. As some of the group members communicate over public lines, Secure Channel is set in operation for these links, to ensure message privacy.

## 5.8 Implementation

Follow the steps for *Broker*, amending them in the following ways;

1. Include identity and rights in the object model. Ensure that all participants are assigned identity, that they are authenticated in order to participate, and that rights are checked before requests are granted.
2. Implement Reference Monitor to check authorization.
3. Implement Secure Channel to protect message traffic.

## 5.9 Consequences

By requiring authentication of each Broker, Client and Server, trust can be established between transaction participants. Based on authentication, access control can be implemented, enabling restrictions on the use of privileged information and functionality.

## 5.10 Known Uses

The CORBA Security Service, Microsoft .NET Remoting and the World-Wide-Web implement at least some aspects of the pattern described here.

## 5.11 Related Patterns

[Bus96] defined the Broker pattern. [Fer01] provides a language that addresses the relationship between authentication and rights management. [Fer03] shows authentication and rights management in a distributed context. [Bro99] gives a detailed implementation of Authentication in a distributed environment. A revised version of Broker, Broker Revisited, is described in [Kir04].

## *6 Conclusion*

We believe the approach of comparing concrete implementations of patterns to the abstract originals for the purpose of carrying real-world experience back in to the patterns is a powerful one, and consistent with the spirit of patterns. We have demonstrated this approach through examining *Broker* through the lens of CORBA and .NET security for the purpose of evaluating how to add security attributes to *Broker*.

There is much work to be done here; *Broker Revisited* has updated *Broker* to more accurately reflect the current understanding of the pattern. This refactoring has delegated service discovery to the *Lookup* pattern [Kir04], elevated *Lookup*'s importance. *Lookup* is interesting in its own right; DNS, in particular, is a thoroughly analyzed example of *Lookup* that has much to offer those looking to secure systems based on *Lookup*.

Along another axis, it behooves us to examine other approaches to securing Broker, namely JINI and the World-Wide Web.

## *Acknowledgements*

The authors would like to thank Kim Canavan, Nelly Delessey, Alvaro Escobar, Maria Larrondo-Petrie, Klaus Marquardt, and Juan Pelaez for their time and ideas in review and support of the paper, and the program committee of EuroPLoP 2006 for a conference scholarship. This work was supported by an earmark grant from the Department of Defense for telecommunications security, administered by Pragmatics, Inc.

## *References*

[And01a] R. Anderson,, *CORBA Security Service Specification*, OMG 2001. http://www.omg.org/docs/formal/02-03-11.pdf

[And01b] R. Anderson,, *Security Engineering*, Wiley 2001.

[Bro99] F.L. Brown and E.B.Fernandez "The Authenticator Pattern," Procs. of PLoP'99 http://jerry.cs.uiuc.edu/%7Eplop/plop99/proceedings/Fernandez4/Authenticator3.PDF

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal*., Pattern- oriented software architecture*, Wiley 1996.

[Fer01] E. B. Fernandez and R. Pan, "A Pattern Language for security models", *Proceedings of PLoP 2001,* http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions

[Fer03] E.B.Fernandez, and R. Warrier, "Remote Authenticator/Authorizer," *Proceedings of PLoP 2003*, http://hillside.net/patterns/

[Kau02] C. Kaufman, R. Perlman and M. Speciner*, Network Security 2nd ed.*, Prentice-Hall 2002.

[Kir04] M. Kircher and P. Jain, *Pattern-oriented software architecture, vol. 3 , Patterns for Resource Management,* J. Wiley, 2004.

[MS03A] Enterprise Solution Patterns Using Microsoft .NET: Broker Pattern http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesBroker.asp

[MS04A] .NET Remoting Authentication and Authorization Sample http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/remsspi.asp

[OMG02] CORBA Security Service v1.8, sect. 1.1.3

[Sch00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, vol. 2 , Patterns for concurrent and networked objects,* J. Wiley, 2000.