The Object Filter and Access Control Framework

Viviane Hays, Marc Loutrel, and Eduardo B. Fernandez *

{vivvv | loutrelm@hotmail.com, ed@cse.fau.edu}

Dept. of Computer Science and Eng.

Florida Atlantic University

Boca Raton, FL 33431

Abstract

The amount of data available in the Internet is growing every day. It has become necessary to protect and filter out this data for institutional and legislative reasons.
We assume here that data is stored as objects in a distributed environment where the objects need to be shared. This framework provides first client request authentication.
In many distributed object systems, once authenticated a client can directly access objects in any way. The Object Filter and Access Control Framework avoids this by constraining a client to access objects in specified ways defined by the client rights.

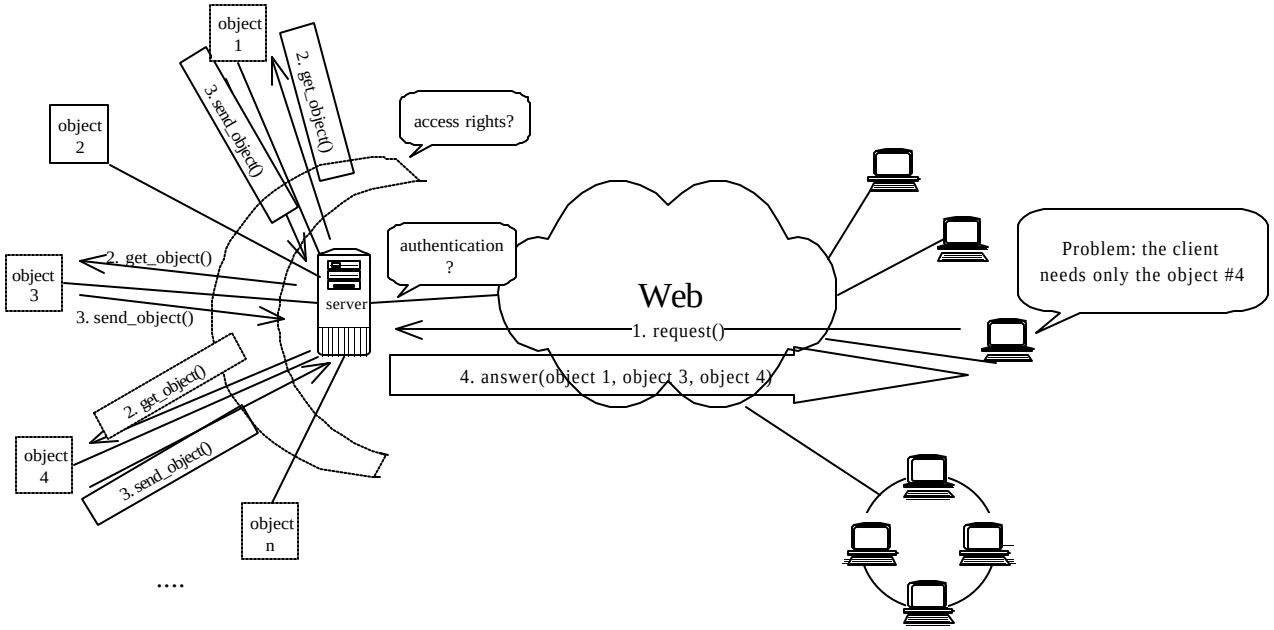
1. Intent

This framework[1] combines the functions of authentication, access control, and data filtering in a distributed environment.

2. Motivation

In distributed systems, data or services requested by clients may need to be filtered to control the type of data provided. Also, these services may need to be restricted to be used by only some users in specific ways. The services or data are provided by distributed objects (Figure 1). An example would be a news translation service from English to German. Users of the service could specify what type of news they do not want to receive. News could sometimes be restricted for legal or security reasons. The users would pay for this service, so only registered users could access it. Furthermore, different groups of users could have different rights (some could add comments on the news, for example).

[1] We use the word 'framework' to indicate a combination of patterns, perhaps the term 'composite pattern' [10] would have been more precise.



1. request() : a request from a client to a server for some data.
2. get_object() : a request to the object made by the server. The server sends this message to every object which matches the request of the client.
3. send_object : each object requested by the server returns itself to the server.
4. answer(object i, object j, …) : the server provides the client with all the objects which match the request.

Figure 1 - Problem

For situations like this, a distributed[1] environment presents the following problems:
- A client that asks a server for some object needs to be authenticated.
- Access rights must be checked for every client to see if it is authorized to access the requested object in the intended way.
- The objects sent back[2] to the client must satisfy filtering policies so they need to be filtered before sending them to the client. Sending all the requested objects to the client for local filtering is clearly inefficient.

[1] The environment may not be strictly distributed, the objects could be in different address spaces in the same node.
[2] Depending on the case, objects, object references, or specific data needs to be sent to the customer.

3. Forces

The forces behind this framework include:

- The clients should be allowed to apply their own policies to filter the data they want to receive.
- Legislation or institution policies may require further filtering.
- The filter should be reusable on different platforms.
- The clients must be authenticated before receiving access to objects.
- The clients may have different access rights on the shared objects.
- The clients should be able to send several requests in the same connection in order to improve efficiency..
- The distribution architecture should be transparent to the users of the framework.

4. Applicability

Use the Object Filter and Access Control Framework when:
- You want to provide registered clients with a filtered data stream, where the filtering policy may be programmable.
- Several network protocols may be available to the clients in order for them to obtain a service.
- Some or all the objects contain sensitive data and must be protected from illegal access.
- The objects provide services that require previous registration to access them.

5. Solution

Figure 2 presents a solution to the problem of Figure 1. This solution includes:

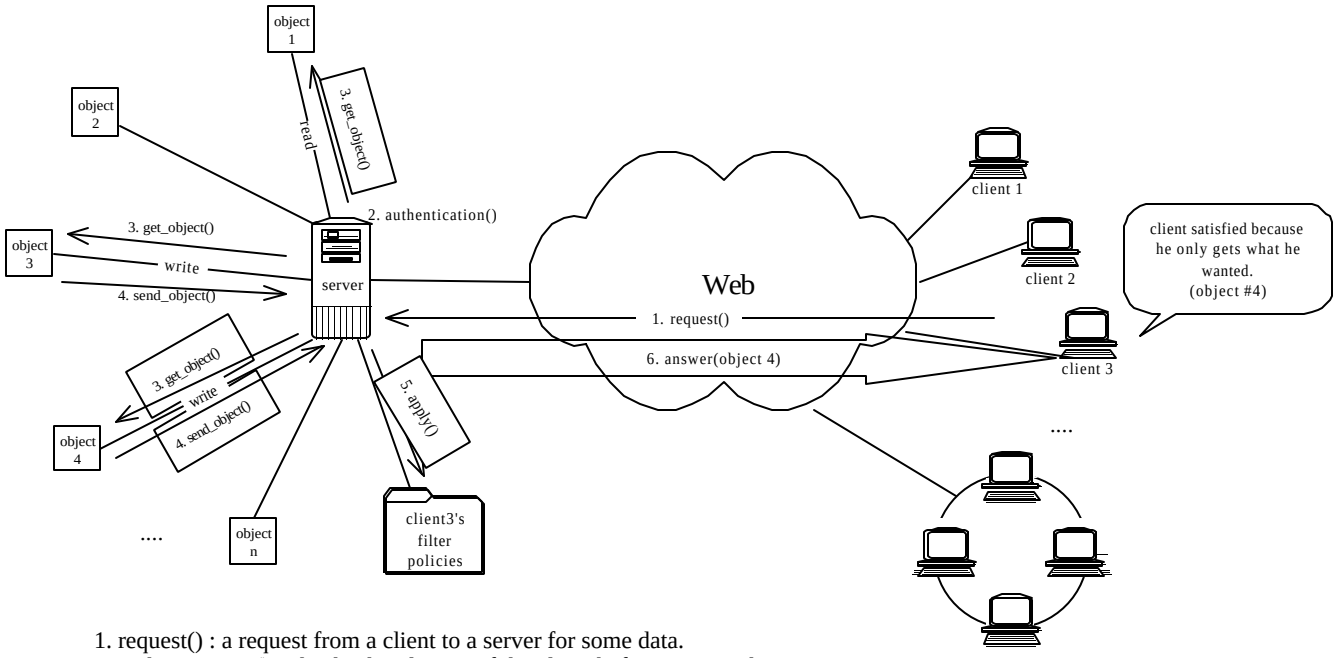
- Every client is authenticated.
- Client's access rights on the objects are checked.
- Objects are filtered according to what the client wants and to general policies.

Although a centralized server is shown in Figure 2 these functions don't need to be applied by the same node, e.g., authentication could be done by a firewall, filtering by a web server, access rights could be checked by a database server.

The resultant system is a combination of features from four different patterns plus a security model structure:

- Data Filter [5]—Filters the contents of client requests in a distributed environment according to predefined policies.
- Bodyguard [3]—Controls access to shared objects in a distributed environment.
- RPC Client [7]—An extension of the Proxy pattern when the proxy is a client-side proxy and the communication protocol is a pipelined request/response protocol.

- Authenticator [2]—Performs authentication of a requesting process before deciding access to a distributed object.



1. request() : a request from a client to a server for some data.
2. authentication() : checks the identity of the client before passing the request.
3. get_object() : a request to the object made by the server. The server sends this message to every object that matches the request of the client. At this point, the filter policies haven't been applied yet.
4. send_object : each object (or reference) requested by the server is returned to the server (objects 3 and 4).
5. apply() : the server applies the policies of the client on the objects (objects 3 and 4) to send it only what it is looking for.
6. answer(object i, object j, …) : the server provides the client with the objects that match the request. In our example, we supposed that object 4 is the only object left after all the filter policies have been applied.

Figure 2 - Solution

The structure of this framework (Figure 3) is mainly based on two important classes:

- The Connection class: Its main activity is to authenticate a Client to a Coordinator and to multiplex several requests on the same Concrete Connection by assigning them different ids.
- The Coordinator class manages the filtering process. This class aggregates the Bodyguard class, the Policy class, the Authentication class and the Policy Applicator class. This class is basically a controller that delegates work to its component classes.

The roles of all these classes are explained below.

6. Participants

· **Client:** An abstract class that uses the Connection Class to filter out the unwanted data. A client uses a Filter Policy Factory to create its own predefined Filter Policies and send them to the Coordinator so that this is able to reprogram and/or update its strategy.

· **Concrete Client:** A concrete class derived from the Client Class. Each Concrete Client defines its own policies. The Coordinator uses these policies to filter the requested object when the Bodyguard allows the client access to it.

• **Filter Policy Factory:** An abstract class that allows a Client to create predefined policies.

· **PolicySetting:** A class that contains user-defined filtering policies.

· **Connection:** This abstract class represents a single connection (independent of network protocols) between a Client and a Coordinator. This Class assigns an ID at each request so that it is possible to multiplex several requests on the same call. It also delivers the answer to the corresponding request.

· **Concrete Connection:** A concrete class derived from the Connection Class. This class contains the server address and the specific network protocol to establish the connection between the Client and the Coordinator.

· **Authenticator:** An abstract class that allows the Coordinator to use a variety of authentication methods to authenticate a client.

· **Concrete Authenticator:** A concrete class derived from the Authenticator class. This class corresponds to a concrete authentication method.

· **Coordinator:** The Coordinator manages the access and filtering process. When access rights are granted for the requested object, the Coordinator sends it to the Policy Applicator and then back to the connection after all the policies have been applied. This class is also very important because it has the responsibility of thread synchronization. The Coordinator also receives and processes *append_policy(), delete_policy()* and *update_policy()* requests from the Concrete Client objects. The Concrete Client issues *activate_policy()* and *desactivate_policy()* requests to the Coordinator. These requests are performed on the User Filter Policy owned by the Client. Every Concrete Client object interacts with a Coordinator object. The Coordinator has the responsibility to marshal procedure arguments and to unmarshal procedure results.

· **Policy Applicator:** A managed resource class that is dedicated to an activated User Filter Policy. It contains the iterative evaluation control for User Policies and evaluation rules for the Client's strategy.

.

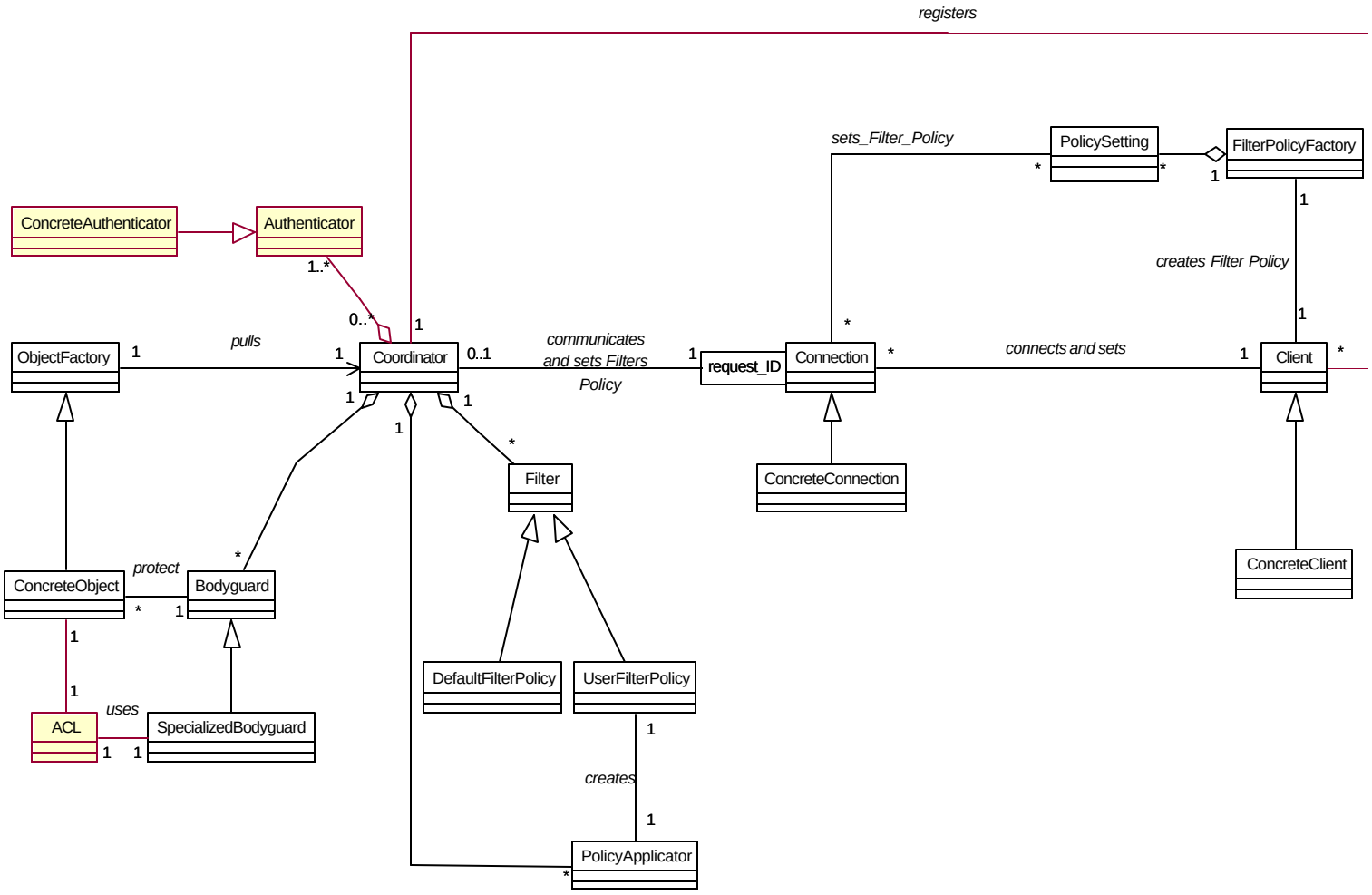


Figure 3. Class diagram for solution

· **Bodyguard:** An abstract class that works as a mediator between the Coordinator and the Concrete Object. The Bodyguard conserves the independence of the Concrete Object from the Coordinator. Once a Client is authenticated to a Coordinator the Bodyguard protects the Concrete Objects against improper access. When a Concrete Object can't be accessed, the Bodyguard will notify the Client

· **Specialized Bodyguard:** A concrete class derived from the Bodyguard. It dispatches validations and assignments of access rights on each object. It applies different authorization models to control access.

· **Object Factory:** An abstract class that creates Concrete Objects.

· **Concrete Object:** A concrete class derived from the Object Factory that corresponds to a shared object in a multithreaded architecture. Its objects embody the data and services requested by the users.

· **ACL:** This class defines the access rights of Concrete Clients on a Concrete Object. For this purpose the Access Matrix [13] and Role Base Access Control [11] can be used.

· **Policy:** An abstract class that permits to distinguish the policies that belong to the Client from those defined by the system. It has the subclasses:

· **User Filter Policy:** It is a concrete class derived from the Policy class. It is created by the Client and contains user-defined policies. The Client can append, delete, and activate a User Filter Policy providing a programmable filter behavior.

• **Default Filter Policy:** A concrete class derived from the Policy class. It corresponds to the policies applied by the filter on which the Client doesn't have any control.

7. Collaborations

We describe with Sequence Diagrams several Use Cases of the framework including Client Subscription, Policy Activation, Access to an Application Object, Delete/Update a Policy and Add a New Policy. In all these cases, a Client needs first to be connected and authenticated. Several patterns have already been designed for these purposes [4, 12], specifically:

Connection:
Connections are established between two endpoints. Usually, endpoints involve two roles:
- a passive role often played by the server and which corresponds to the initialization of an endpoint of communication.
- an active role often played by the clients and which corresponds to the initialization of a connection with one or more endpoints of communication.

Authentication:
A mechanism that provides identification and authentication from a client to a server. The protected distributed objects are accessible only after a successful authentication.

Client Subscription (Figure 4):

Before using a Coordinator, a Concrete Client has to subscribe. For this purpose, he uses a Concrete Connection and sends a *subscribe_client()* request through it. The Client will use later the Concrete Connection to send a request to the Coordinator. For this request, the Concrete Connection just sends the request to the Coordinator (no ID is assigned to the request).
After getting his ID, the Client can register with the Coordinator so that he will be able to use later the Coordinator to access objects. The Client is now associated with the Coordinator. This service sends a *success()* message to the Concrete Connection which transmits it to the Concrete Client to end the Client Subscription.

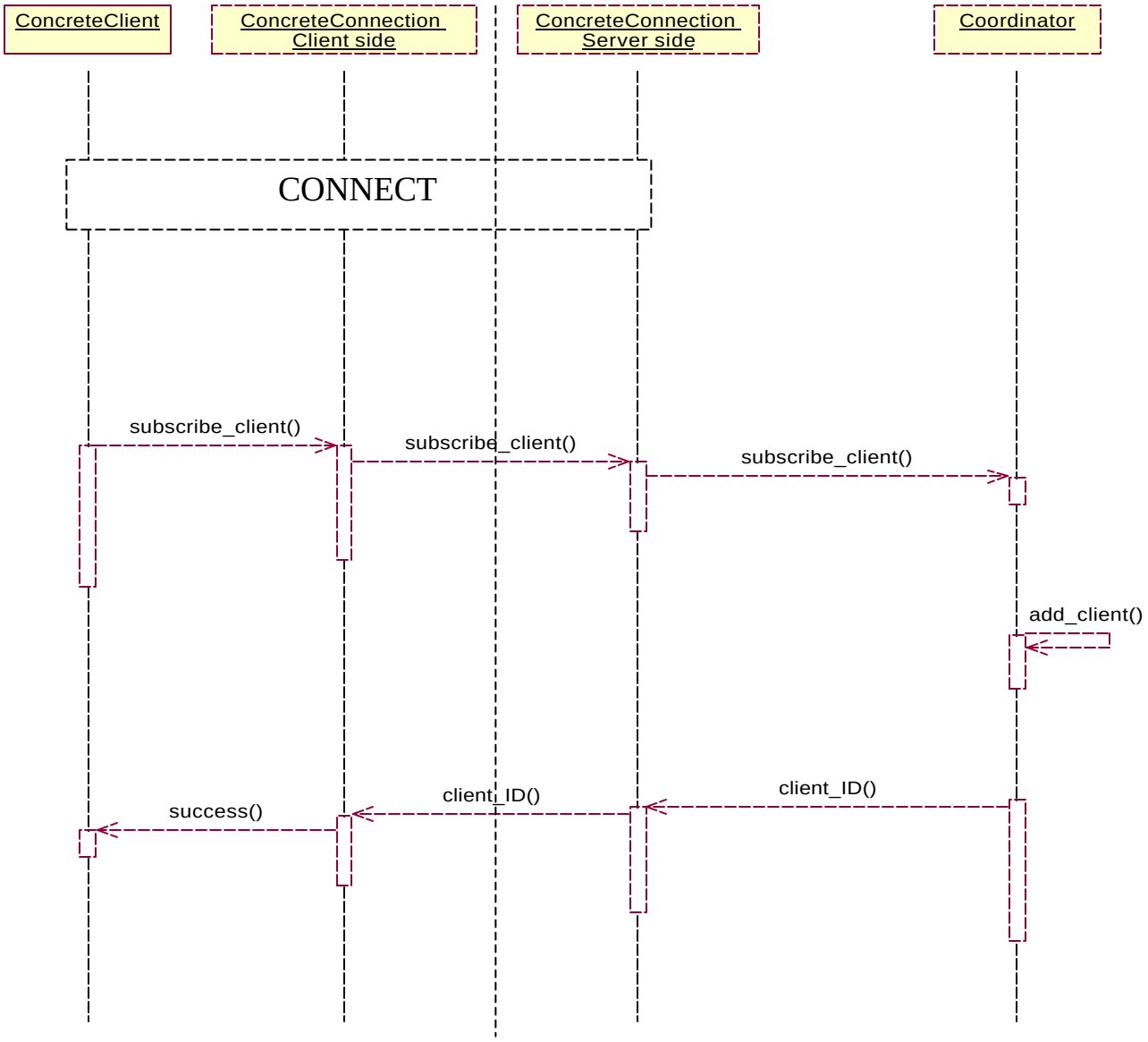


Figure 4 – Client Subscription

Policy Activation (Figure 5):

Before activating a User Filter Policy a Concrete Client has to be authenticated. This authentication is made between the Concrete Connection and the Coordinator.
The Coordinator checks if the Client is the owner of the policy. Then the Coordinator activates the User Filter Policy (if it isn't already activated) by sending an *activate_policy()* request to the User Filter Policy.
Once activated, the User Filter Policy creates a Policy Applicator for this user to perform the filtering of his objects. To conclude the Policy Activation, a *success()* is sent back to the Concrete Connection that looks for the request ID and transmits the *success()* message to the Client.

Delete / Update a Policy:

To delete or update a User Filter Policy, the procedure is the same. Instead of *activate_policy()*, we use *delete_policy()* or *update_policy()*.

Access to an Application Object (Figure 6):

After being registered with a Coordinator, a Concrete Client can send requests to it.
He sends the *request()* message to the Concrete Connection. The Concrete connection, which manages the requests, assigns an ID to the request (*assign_request_ID()*) and then sends the request to the Coordinator. The Concrete Client needs to be authenticated and identified to the Coordinator before using it. The Coordinator authenticates the Client using one of the Concrete Authenticator. The Coordinator checks if the Concrete Client is registered. If the Client is recognized (*success()*), the Coordinator creates a Specialized Bodyguard and then assigns it to the requested Concrete Object.
The Coordinator sends the request to the Specialized Bodyguard that checks the client's access rights for the Concrete Object. If the Concrete Client has access to the Concrete Object, the Coordinator executes the Default Filter Policy and the Policy Applicator. The Policy Applicator retrieves the Concrete Object and executes each User Filter Policy's algorithm on it. The filtered Object is sent back to the Concrete Connection through the Coordinator. When the Concrete Connection receives the filtered Object, it delivers the filtered Object to the Concrete Client.

Add a New Policy (Figure 7):

To add a new User Filter Policy, a Concrete Client has to be authenticated after connecting. This authentication is made between the Concrete Connection and the Coordinator.
After this the Concrete Connection requests the Filter Policy that the Client wants to append. If the Filter Policy is found, it is returned to the Concrete Connection so that it can be added by the Coordinator to the User Filter Policy. A *success()* is sent back to the Concrete Connection that looks for the request ID and transmits the *success()* message to the Client.

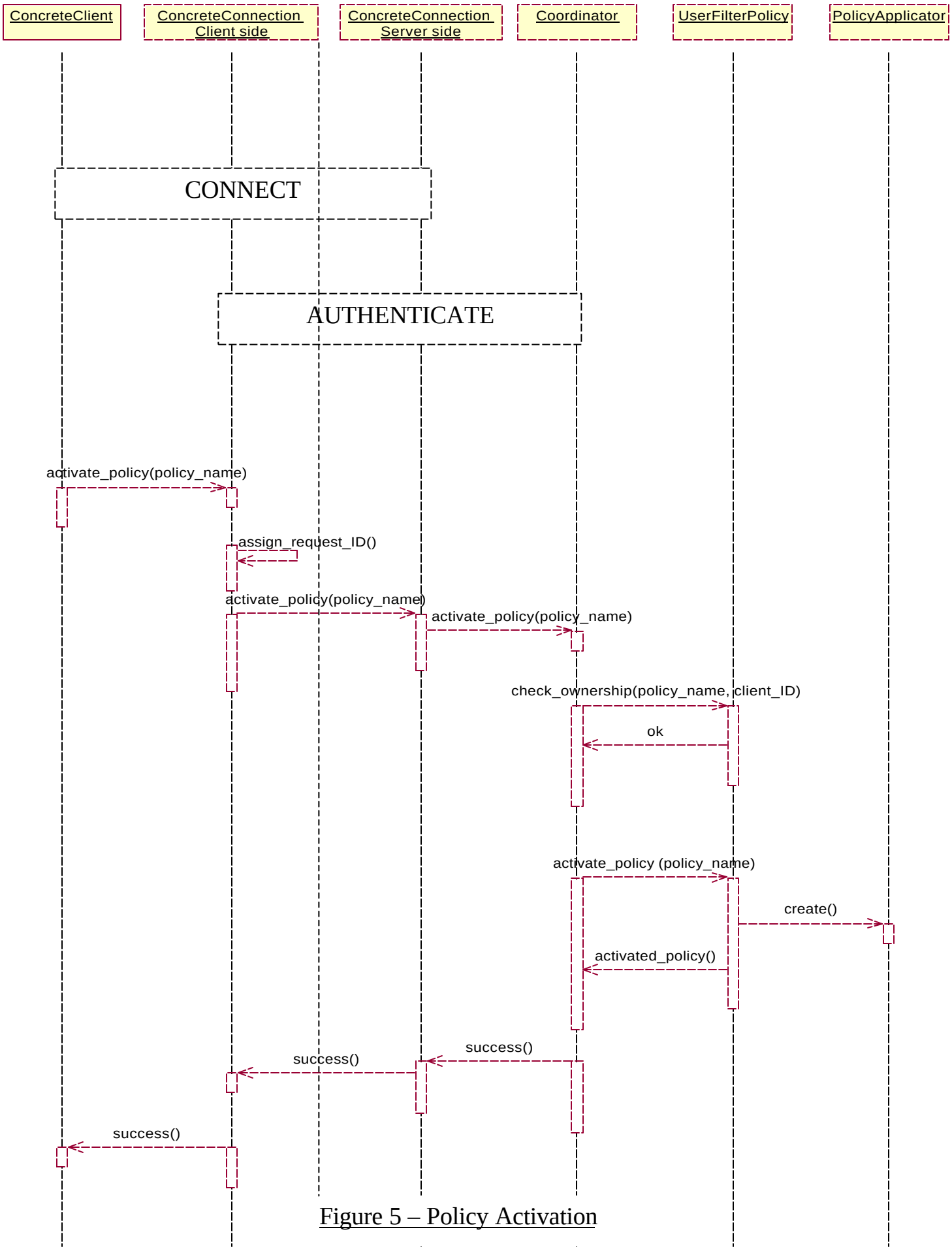


Figure 5 – Policy Activation

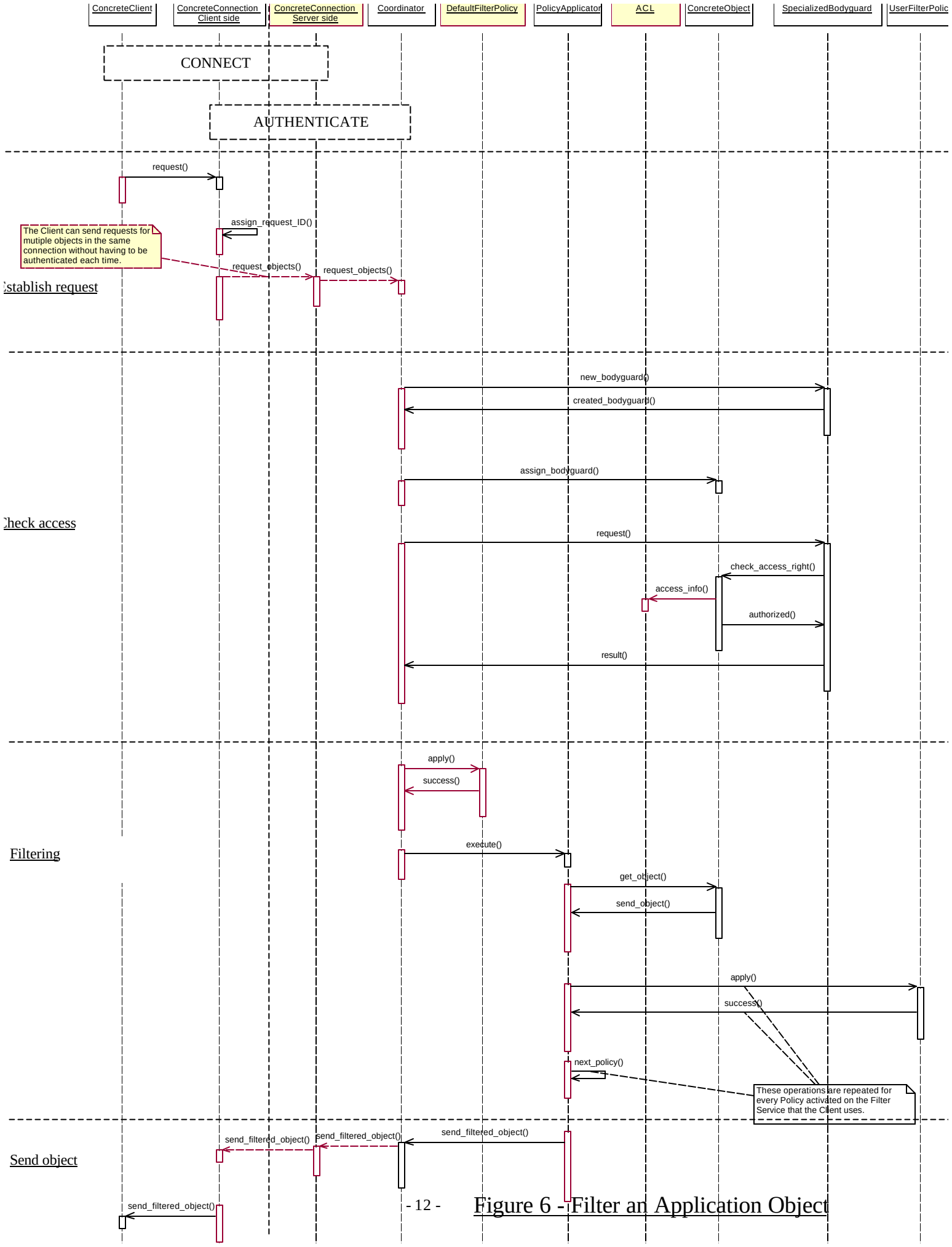


Figure 6 - Filter an Application Object

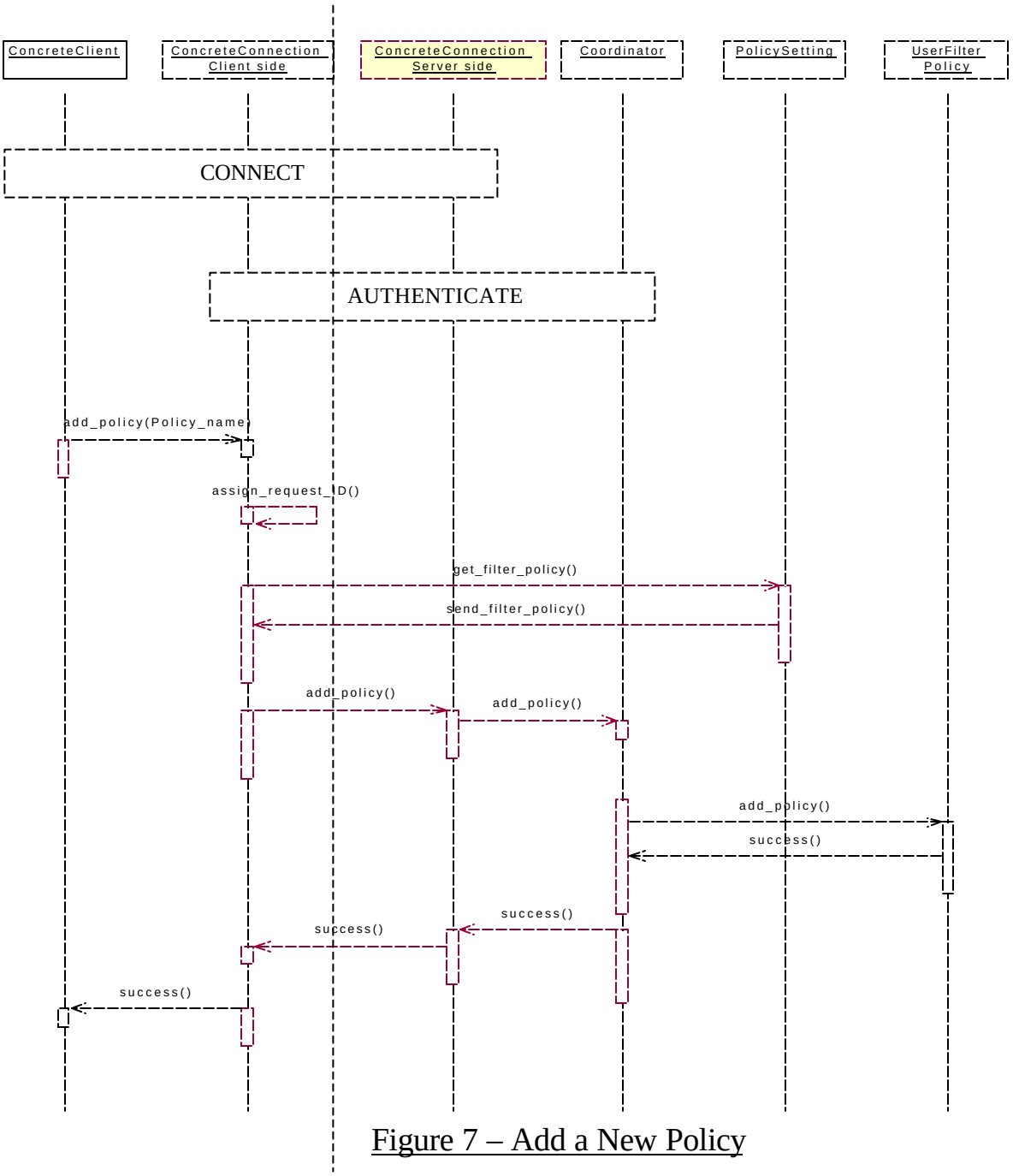


Figure 7 – Add a New Policy

8. Consequences

The Object Filter and Control Framework provides the following benefits:

- The client can implement his own filter policies, update them and remove them from a service, which makes this framework extensible.
- This framework has a high degree of concurrency and can be efficiently implemented in a multithreaded environment. Note that multithreading does not bring higher performance unless properly used.
- In this framework dependent layers are separated from independent layers resulting in an approach independent of the network protocol and the objects. The abstract classes and the system classes (connection, policy,…) define a generic architectural layer independent of specific systems, while the user-provided concrete classes define an application-dependent layer.
- A service grants access to an object only after identification and authentication of the client and only specific rights.
- Access to an object is granted dynamically for each client who requests it.

Some possible drawbacks are:

- Performance might be affected by the complexity of the functions. The decoupling of functions may bring extra overhead.
- Implementation may be complex, but the structure is highly reusable, so this is a one-time extra work.

9. Related Patterns

As indicated earlier, his framework combines aspects of four patterns that have appeared recently: the Data Filter Architecture [5], the Bodyguard [3], the RPC Client [7], and the Authenticator [2].
Compared to the Data Filter Architecture Pattern [5], we can notice that each client must be identified and authenticated to a Coordinator before using this service. Furthermore, objects cannot be accessed directly by a client and access rights to them are granted dynamically by the Bodyguard. Another main difference is that all the network specifications are encapsulated in the Concrete Connection class.
The framework also uses several Strategy patterns [6] to indicate varieties of filters policies, authentication approaches, authorization models, and connection methods. Finally, patterns such as those in [4] and [12] would be used to start the connections.

10. Known uses

The Orbix system developed by IONA Technologies is an implementation of the CORBA architecture and uses filters for its distributed objects[1]. Hewlett-Packard’s e-speak provides authorized services available to registered (and authenticated) users in a distributed environment [8]. The WebSphere platform from IBM incorporates all these functions [9].The references of the individual patterns used here also describe uses of them in specific systems.

Acknowledgements

We thank Dan Zhou and our shepherd Manfred Lange for their useful comments that significantly improved this paper. The comments of the group at the Writers Wokshop of PLoP 2000 were also very valuable.

References

[1] Seán Baker, *CORBA Distributed Objects Using Orbix*, Addison-Wesley 1997.

[2] F. Lee Brown, James DiVietri, Graciela Diaz de Villegas, and Eduardo B. Fernandez, "The Authenticator pattern", *Procs. of PLOP'99*, http://st-www.cs.uiuc.edu/~plop/plop99.

[3] Fernando Das Neves and Alejandra Garrido, "Bodyguard", Chapter 13 in *Pattern Languages of Program Design 3*, Addison-Wesley 1998.

[4] Alexandre Delarue and Eduardo B. Fernandez, "Reactor-Acceptor-Connector", *Procs. of PLOP'99*, http://st-www.cs.uiuc.edu/~plop/plop99.

[5] Robert Flanders and Eduardo B. Fernandez, "Data Filter Architecture Pattern", *Procs. of PLOP'99*, http://st-www.cs.uiuc.edu/~plop/plop99.

[6] Eric Gamma, Matt Helm, Ralph Johnson, and John Vlissides, *Design Patterns-Elements of reusable object-oriented software*, Addison-Wesley 1995.

[7] Mark Heuser and Eduardo B. Fernandez, "RPC Client: A Pattern for the Client-Side Implementation of a Pipelined Request/ Response Protocol", *Procs. of PLOP'99*, http://st-www.cs.uiuc.edu/~plop/plop99.

[8] Hewlett-Packard Company, *e-speak*, http://www.e-speak.hp.com

[9] IBM Corporation, *WebSphere software platform for e-business*, http://www.ibm.com/websphere

[10] D. Riehle, "Composite design patterns", *Procs. of OOPSLA'97*, 218-228.

[11] R.Sandhu et al., "Role-Based Access Control models", *Computer* , vol. 29 , No2, February 1996, 38-47.

[12] Douglas C. Schmidt, "Design Patterns for Initializing Network Services", http://www.cs.wustl.edu/~schmidt/patterns.html

[13] Rita C. Summers, *Secure Computing: Threats and Safeguards*, McGraw-Hill, 1997