The Authenticator Pattern

F. Lee Brown, Jr.
James DiVietri
Graziella Diaz de Villegas
CyberGuard Corp.
Fort Lauderdale, FL 33309

Eduardo B. Fernandez
Dept. of Computer Science and Eng.
Florida Atlantic University
Boca Raton, FL 33431

Abstract

A server system acting as a repository of objects available to a variety of unrelated distributed clients is likely to require a means to restrict access based on the identity of the requesting client. Identification and authentication protocols are often missing from currently available distributed object systems.

The Authenticator pattern describes a general mechanism for providing identification and authentication to a server from a client. It has the added feature of allowing protocol negotiation to take place using the same procedures. The pattern operates by offering an authentication negotiation object which then provides the protected object only after authentication is successful.

1 Intent

The Authenticator pattern performs authentication of a requesting process before deciding access to distributed objects .

2 Motivation

The registry services of commonly used distributed object systems generally allow any requestor with access to the registry to obtain any distributed object whose name is known. Some means to grant or deny access to individual requestors using a login-and-authenticate protocol is necessary for distributed object systems that service varied requests.

This is particularly important in distributed systems with a variety of accesses, both internal (Intranet) and external (Internet).

Remote objects may also need to be configured account for varying capabilities between the client and server systems.

The forces behind this pattern include:

- The clients may have different rights on the remote objects. Before deciding access, the requestor must be authenticated.
- The remote object needs to adapt to the capabilities of the two systems involved. Those must be reconciled through a negotiating process prior to constructing the object.
- A generic approach, not dependent on specific authentication approaches, is necessary.

3 Applicability

The Authenticator pattern is useful

1. when identification and authentication is required for access to remote objects;
2. when a variety of authentication methods may be used;
3. when additional protocol negotiations (encryption selection, software version supported, etc.) is required prior to obtaining a remote object; and the underlying distributed system does not support these requirements.

4 Structure and Participants

The Authenticator pattern uses a distributed object accessible remotely that will identify and authenticate the requesting agent ("requestor") and possibly perform some protocol negotiation. If and only if the authentication and negotiation is successful will the authenticating object create and make available another distributed object representing the object that the requestor really wants.

The Authenticator pattern consists of the following components.

1. *Authenticator.* This abstract class defines the interface used to authenticate a connection or negotiate session parameters. An application defines a concrete class implementing this interface to provide a specific authentication or negotiation protocol. The resulting class should be instantiated and registered with the applicable naming service as a distributed object accessible throughout the network. The `authenticate` method is used by the requestor after obtaining a reference to the Authenticator object. When authentication is successful, the Authenticator class creates an instance of a distributed object that can now be accessed by the remote requestor using the `get` method. The proposed implementation of the pattern uses an *Object Factory* class passed to the Authenticator constructor to create the protected object.
2. *Object Factory.* This abstract class contains only one method, `create`. The implementation of this method creates the protected object. It may also perform other actions specified by the Authenticator as a result of the negotiations.
3. *Requestor.* Although not strictly a part of the pattern, the remote requestor object is assumed to be implemented in a manner matching the implementation of the concrete Authenticator class, so that the values passed as arguments to the `authenticate` method and the returned values can be used to complete the authentication.

Figure 1 shows the participants of the Authenticator pattern.

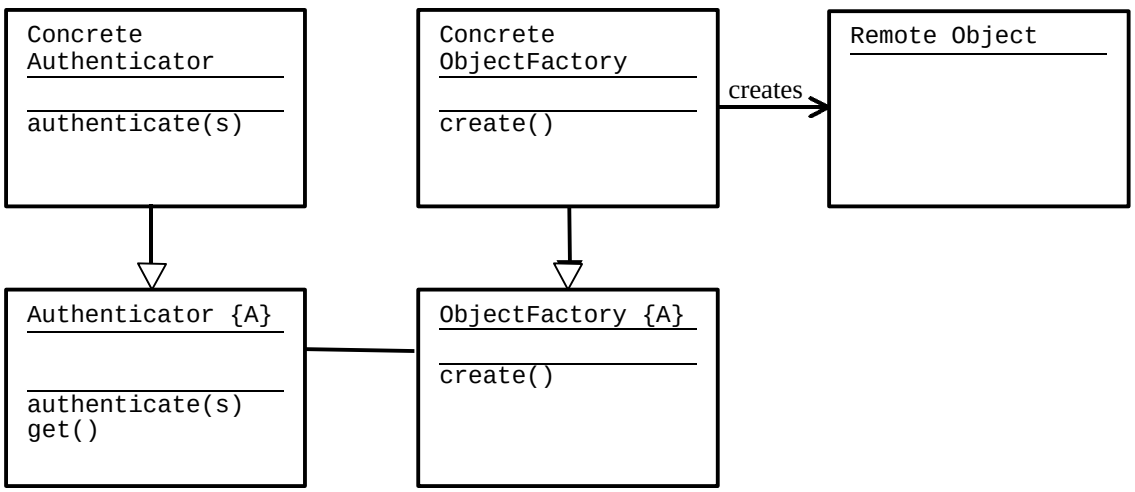

Figure 1 – Authenticator Pattern Participants.

5 Collaborations

There are four phases of operation for the Authenticator pattern.

1. *Initialization.* The Authenticator object must be registered with the distributed object system's naming service for remote access. This phase of the operation is system specific and external to the pattern, but nevertheless tightly coupled with it.
2. *Connection.* When a remote object, a requestor, obtains a reference to the Authenticator object, it uses the `authenticate` method to pass a string to the Authenticator implementation. The return value is another string which the requestor uses to determine if the authentication or negotiation succeeded or failed, or whether (and how) to construct another string for another use of `authenticate`. The requestor continues to call `authenticate` until the authentication or negotiation is completed.
3. *Creation.* When the Authenticator implementation recognizes a successful authentication, it creates the protected object in preparation for handing that to the requestor as a response to the `get` method. The expected process is for the `authenticate` method to invoke an object factory method. However, the Authenticator implementation and the object factory can use any means to make one or more objects accessible.
4. *Acquisition.* The requestor finally uses the `get` method to get a reference to the protected object.

The following diagram illustrates the operation of the pattern.

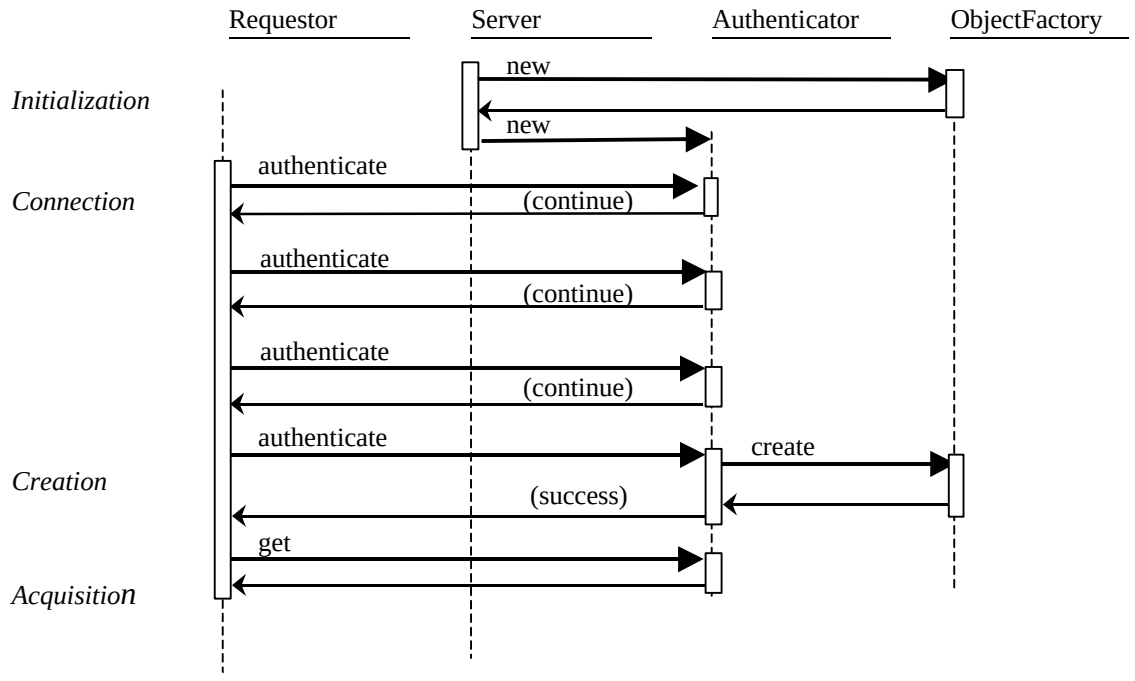

Figure 2 – Authenticator Pattern Sequence Diagram.

To summarize:

- *Initialization.* Server creates and registers an Authenticator implementation object with the registry, passing an object factory object to Authenticator as a constructor parameter.
- *Connection.* Requestor calls registry lookup to get a reference to the Authenticator implementation object.
- Requestor calls `authenticate(`*string*`)`.
- Authenticator returns completion, failure, or continuation string.
- Previous two steps are repeated as necessary to complete authentication/negotiation.
- *Creation.* `authenticate` calls the object factory to create/initialize protected object prior to returning successful completion.
- *Acquisition.* Requestor calls `get` to obtain reference to protected object.

6 Consequences

The Authenticator pattern provides the following benefits.

1. *An option for applications that require complex identification and authentication.* The Authenticator pattern allows for the implementation of different authentication methods, thus allowing multiple clients to use their own authentication methods.
2. *An option for applications with complex protocol negotiation.* An application may require some negotiation with the server before it can be granted access to an object. The negotiation can support encryption method negotiation, key exchange, greatest common denominator version negotiation, parameterization of the protected object (i.e., a parameter to pass to the object factory for the protected object construction), and any other requirements of the application.
3. *Multiple clients accessing remote objects concurrently.* Providing that the operating system and hardware platform support multiple CPU's, this pattern can allow multiple clients to be granted access to remote objects concurrently.
4. *It provides a generic approach, not dependent on the client or the accessed object.*

The Authenticator pattern has the following liabilities:

1. *The abilities of this pattern are logically features of the distributed object system.* A distributed system may already provide features that allow iterative authentication or protocol negotiation. This pattern may overlap those features of the underlying system. However, if we are designing a new system, this is not a problem.
2. *Complicated implementation and debugging.* It may be complicated to program the authentication and negotiation and then to debug the server side of the authentication pattern, especially if there is concurrent client access.

7 Implementation

Some implementation issues to consider when designing an Authenticator pattern implementation are discussed below.

1. *Security.* The object factory class is hidden within the Authenticator implementation for security. The Authenticator implementation is a remote object accessible to the untrusted requestor client, the object factory is not a remote object and is therefore inaccessible to the client. (If not remote objects, then some other means may be necessary to restrict access to the object factory.) The Authenticator implementation must not provide a means to invoke the object factory without completing the negotiation.
2. *Negotiation.* The design of the authentication and negotiation must take into account concurrent access from multiple clients (if, for example, the Authenticator is a Singleton), the possibility of dropping a network connection or timing out, incorrect or out of sequence responses, and other unintentional or intentional failures. It must also provide an efficient means of indicating the current status of the negotiation to the client:, usually

one of succeeded, failed, or still in progress. The results of the `authenticate` method must be sufficient to guide the client to the next step of the negotiation.

3. *Parameterized object creation.* It may be necessary to parameterize the creation of an object by the object factory. The parameter list for the object factory's `create` method shown in the example below is empty, but that is not a requirement as long as the Authenticator implementation and the object factory class agree. The Authenticator implementation may build a parameter list from data supplied by the client during negotiation, or it may provide an additional method for the client to use to specify parameters.

4. *Creation of multiple objects.* Once the authentication is complete, is exactly one object accessible or can the client create any number of objects? If any number of objects can be created, must they all be instances of the same class? The example below shows exactly one object being made accessible. This is enforced by creating the object in the `authenticate` method so that multiple calls to the `get` method return the same object. Alternatively, the `get` method could construct a new instance each time it is called if the `authenticate` method has indicated that the authentication has completed successfully.

8 Sample Code

A simple Java language implementation is shown here. The two abstract classes that form the basis of the pattern are shown as Java interfaces.

```
// Object factory interface.  The create method constructs and returns
// an object as determined by the implementing class.
//
public interface ObjectFactory
{
      public Object create();
}


// Authenticator interface for a remote object.  The implementation
// provides specific authentication requirements.
//
public interface Authenticator
      extends Remote
{
      // Potential response strings for the authenticate method.
      //
      public final static String AUTHENTICATED = “AOK”;
      public final static String AUTHENTICATION_FAILED = “ERR”;
      public final static String AUTHENTICATION_CONTINUE = “MORE”;

      // Accept a key string that should be the next response from
      // the client in the negotiation protocol, and return a prompt
      // string or status indication.
      //
      public String authenticate(String key)
            throws RemoteException;

      // Return the protected object.  This method returns null until
      // the negotiation is successfully completed.
      //
      public Object get()
            throws RemoteException;
}
```

An application must implement the Authenticator and ObjectFactory interfaces. The following Java code shows a sample framework that might be used by an application-specific implementation.

```
// Object factory implementation.  The create method returns a remote
// object of a predetermined type.
//
class ObjectFactoryImpl
      implements ObjectFactory
{
      public ObjectFactoryImpl()
      {
      }

      public Object create()
      {
            MyRemoteObject obj = null;
            try
            {
                  obj = new MyRemoteObject();
            }
            catch (RemoteException x)
            {
                  System.err.println("RemoteException error: " + x);
                  return null;
            }
            return ((Object) obj);
      }
}

// Authenticator implementation.  This implementation defines the
// authentication requirements and any other protocol negotiation.
// Successful completion of the negotiation immediately creates an
// instance of the protected object which can then be retrieved
// using the get method.
//
class AuthenticatorImpl
      extends UnicastRemoteObject
      implements Authenticator
{
      private ObjectFactory factory;
      private Object robject;

      public AuthenticatorImpl(ObjectFactory inFactory)
            throws RemoteException
      {
            super();
            factory = inFactory;
            robject = null;
      }

      public Object get()
            throws RemoteException
      {
            return robject;
      }
```

```
	public String authenticate(String key)
		throws RemoteException
	{
		String response = new String("");

		/* Application-specific authentication */

		if (response.equals(Authenticator.AUTHENTICATED))
		{
			robject = (Object) factory.create();
		}

		return response;
	}
}
```

The Authenticator implementation would be registered with the Java RMI naming service [9] using a statement such as the following:

```
Naming.rebind(name, new AuthenticatorImpl(new ObjectFactoryImpl()));
```

Because the constructor and the `get` method of the Authenticator interface are likely to be implemented the same way regardless of the application, it would be useful to supply a DefaultAuthenticator class that implements the Authenticator interface. Applications could then derive from that class.

The Authenticator implementation will surely need to take into account sequential and concurrent authentication requests. The framework shown above implies a single object used by all remote clients; multiple threads or a means to obtain separate remote Authenticator objects will be required in most real-world applications.

9 Known Uses

Most HTTP servers allow free access to any resource they control, or control access based on the (possibly forged) source address of the requestor. In spite of this, many web resources have found it useful to employ a login and password dialog. An equivalent or even more powerful authentication mechanism, such as is provided by the Authenticator pattern, is similarly useful for non-interactive access to remote resources.

The Authenticator pattern can be used to create a dialog between the requestor and the server prior to creating or granting access to a protected object. The negotiation can support identification (e.g., login), authentication (e.g., simple password, challenge response, multiple challenge responses), encryption method negotiation, key exchange, greatest common denominator version negotiation, parameterization of the protected object (i.e., a parameter to pass to the object factory for the protected object construction), and any other requirements of the application.

This approach was used in the design of some internal software at Cyberguard, Fort Lauderdale, FL, and is being considered for some products. It is clear that, because it satisfies a very basic need, there must be other implementations in practice.

10 Related Patterns

Authentication is just one aspect of security. It must be complemented with some authorization mechanism that determines role rights or a similar security mechanism [2]; a pattern such as the Bodyguard [1] could be used for that purpose.

The Authenticator pattern is a variation on the Abstract Factory pattern [3] in that it is basically a factory class. However, instead of an Abstract Factory implementation that statically determines what object to create, the Authenticator uses an iterative negotiation to determine *if* an object (and perhaps also *what* object) it should provide.

Because of the separation of distribution aspects from application aspects, this pattern is in the same category as Schmidts' patterns [4-8].

References

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

[2] E.B. Fernandez and J. C. Hawkins, "Determining role rights from use cases", *Procs. 2nd ACM Workshop on Role-Based Access Control*, 1997, 121-125.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[4] D. C. Schmidt. "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching." *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[5] D. C. Schmidt. "Acceptor: A Design Pattern for Passively Initializing Network Services." *C++ Report*, vol. 7, November/December 1995.

[6] D. C. Schmidt,. "Connector: A Design Pattern for Actively Initializing Network Services." *C++ Report*, vol. 8, January 1996.

[7] D. C. Schmidt. "A Family of Design Patterns for Application-Level Gateways." *Theory and Practice of Object Systems*, J. Wiley & Sons, vol. 2, no. 1, December 1996.

[8] D. C. Schmidt. "Acceptor and Connector: Design Patterns for Initializing Communication Services." *The 1st European Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, July 1997.

[9] "Java Remote Method Invocation." http://java.sun.com.products/jdk/1.2/docs/guide/rmi/spec/rmi-intro.doc1.html. Sun Microsystems, Inc., Redmond, Washington, 1977.