Jgss - A Kerberos Security Service Provider For Secure RMI

Daniel Jiang
xjiang@mit.edu

1. Introduction

Java Remote Method Invocation (RMI) has been widely used in development of distributed Java applications. Its success, compare to systems such as Network Objects[3], has been mainly due to its flexibility and ease of use. Both of these advantages are deeply related to Java's support for potable code, which enables the code of a server proxy, or stub, to be downloaded into its client at run time, essentially providing server the ability to dynamically extend the client's class system. In fact, dynamic code downloading is two directional in RMI, a client can also extend its server's class system by providing byte code of parameter objects to the server when it invokes the server's remote methods.

Though portable code improves flexibility and extensibility in RMI, security concerns have been preventing distributed applications using RMI to be deployed across untrusted networks such as the internet. Secure Jini Extensible Remote Invocation (JERI), a framework recently proposed by Sun Microsystems Inc., extends RMI to provide fine-grained access control and network security support to distributed applications which employ dynamic code downloading. Secure JERI allows applications to choose providers using various authentication and encryption schemes to perform related network communication.

As a practical mechanism to facilitate distributed application development and deployment, one weakness of the current release of secure JERI is that only one provider - Jsse (supports SSL) has been provided. No providers support commonly used security protocols other than SSL are available. Not only the lack of alternative providers hampers users to deploy applications using secure JERI into environments where SSL is not the preferred choice, it also implies that the multi-provider design of secure JERI has not been evaluated in an environment when multiple providers are available.

In order to extend secure JERI to support security protocols beyond SSL, and to evaluate its design as a customizable framework, a new security service provider supporting Kerberos protocol, Jgss, is developed. The design and implementation of Jgss are described in detail in the following sections of this paper.

Kerberos is chosen because it is a mature protocol and has been widely deployed. Also because it is one of the very few security protocols currently supported in Java.

The new provider is named as Jgss because it uses Generic Security Service Application Program Interface (GSSAPI) Java binding [1,2] modules to wrap clear text messages into opaque tokens before placing them onto the network. Though GSSAPI is a generic interface, currently Kerberos is the only protocol supported in Java through this interface.

On nice feature supported in Jgss while not in Jsse is delegation, an important feature for multi-tier distributed applications.

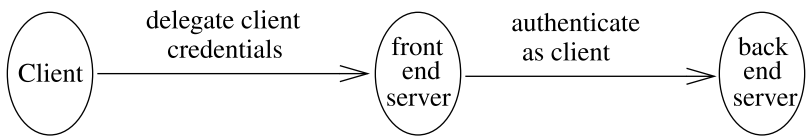

Figure 1.1 Delegation

As shown in figure 1.1, delegation allows a client to securely delegate its credentials to a front end server, which subsequently interact with other backend servers on behalf of the client.

2. Related work

Figure 2.1 shows a simplified picture of how remote calls are handled in secure JERI. First, a server exports itself through secure JERI. As a result of this exporting process the server gets back from secure JERI exporter a stub or dynamic proxy object, which it registers with some kind of registry or lookup service.

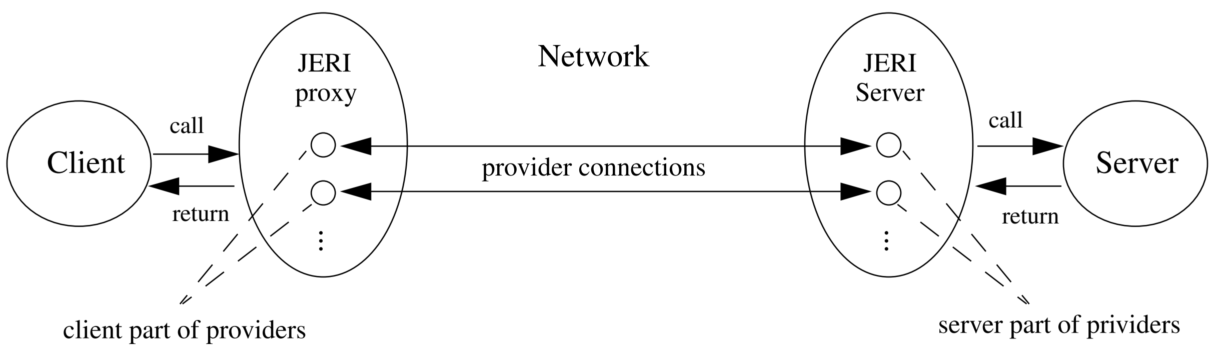

Figure 2.1 Remote method invocation through secure JERI

A client finds the proxy it needs from a lookup service, downloads the serialized proxy into its own JVM, instantiates it, and invokes remote method calls on the instantiated proxy object. The proxy then translates remote method calls into requests and responses sent to and from the server through the network.

In secure JERI, network communication is governed by network security constraints, which can be specified by both the server and client sides when applications are deployed. Secure JERI relies on its transport layer providers such as Jgss to perform the actual network communication and ensure that the constraints are satisfied. Multiple providers can be employed concurrently by secure JERI.

Secure JERI defines a set of security constraints for network communication between client side proxies and their corresponding servers. Constraints are normally specified in a configuration file and provided to an application when it is deployed. A client and its server normally have different configuration files. Following are some examples of possible constraints and their meaning:

1. ServerAuthentication.YES: server should be authenticated before any request can be sent to it.
2. ServerMinPrincipal: a constraint on the server, such that if the server authenticates itself, then it must authenticate itself as at least all of the principals specified in this constraint. This constraint only has effect if ServerAuthentication.YES is also specified.
3. ClientMaxPrincipal: a constraint on the client, such that if the client authenticates itself, then it may only authenticate itself as one or more of the specified principals. A client can use this constraint to limit how much of its identity is exposed in remote calls.

When exporting the server, secure JERI associates the exported proxy object with a set of server constraints, which will be enforced by the server-side providers handling network communication for secure JERI. Server constraints are also transferred to the client with the proxy.

After the client downloads the proxy, it associates a set of client constraints with the proxy. The combined security constraints from both client and server are enforced by client part of secure JERI providers.

Security constraints are specified either as required or preferred. For a given remote method invocation, secure JERI tries to select the best available provider to carry out the corresponding network communication. The provider selected is the one that can satisfy all required constraints and the most number of preferred constraints.

The secure JERI architecture is very flexible, in that it supports customization at various layers. Detailed discussion of this aspect of secure JERI is out of the scope of this paper; for further information, refer to [5].

3. Design

As shown in figure 3.1, in a layered hierarchy, the Jgss provider serves as the transport layer under secure JERI. To send requests across the network, Jgss uses the GSSAPI Java modules, hence referred to as GSS modules, to wrap messages into opaque tokens at the sending side, and then sends the token over the wire. At the receiving side, Jgss consults the GSSAPI modules to unwrap the tokens, and passes the decoded messages up to the secure JERI layer.

Besides sending and receiving wrapped message tokens, Jgss modules are also responsible for the following:

1. Establishing TCP socket connections between client and server.
2. Establishing GSS contexts between client and server by exchanging GSS context initialization tokens over the corresponding socket connection. We refer to the pairing of socket connection and the established GSS context associated with it as a Jgss connection.
3. Examining sets of security constraints to determine whether they can be supported by this provider.

4. Examining a set of established Jgss connections and determining whether one of them can satisfy all requirements of a given set of constraints, so can be used to carry out the corresponding remote method invocation. Otherwise, a new Jgss connection will need to be established to handle the method invocation.
5. Checking whether the caller has required permissions to open/accept corresponding TCP connections, and to authenticate as specified local principals to the peer principals involved.

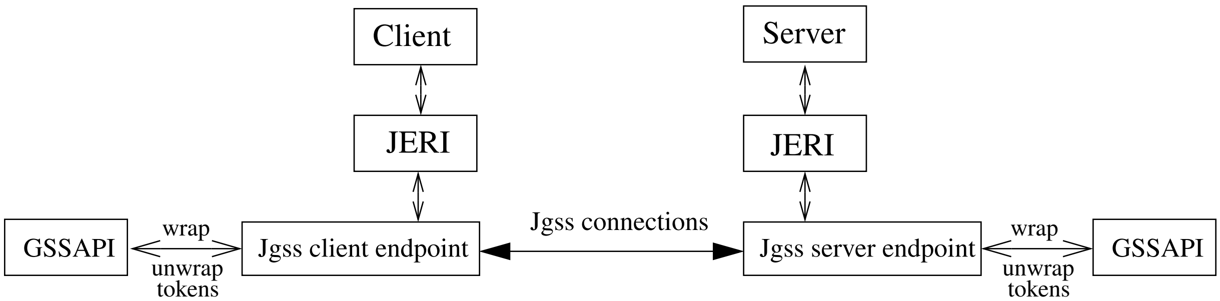

Figure 3.1 Layers and modules

As an example, suppose a client invokes a remote method through secure JERI. The constraints imposed on this method require the client and server to be authenticated as "Alice" and "Bob", and the content of the request/reply of the method invocation to be encrypted. First, secure JERI queries all available security providers, including Jgss, asking whether they are able to satisfy the constraints. Jgss needs to perform the following steps to decide:

1. Check whether the server can authenticate as "Bob".
2. Verify whether credentials to authenticate the client as "Alice" are available.

If both (1) and (2) are true, Jgss informs secure JERI that it can support the requested constraints. Suppose Jgss is then chosen by secure JERI as the provider to carry out the remote call. Jgss will then do the following to carry out the corresponding network communication:

1. Verify that the call stack of the current thread has permission to connect to the server host and port.
2. Verify that the current call stack of the thread has permission to authenticate as "Alice" to "Bob".
3. Check whether there exists an active Jgss connection which can satisfy all required constraints. If yes, reuse it, otherwise establish a new Jgss connection that can satisfy the required constraints.
4. Use the GSS context of the Jgss connection to wrap method request data as GSS tokens and send them over the connection.
5. Receive the reply GSS tokens from the connection, unwrap them using the context, and pass the result to the upper secure JERI layer.

The server Jgss endpoint performs the following:

1. If this is a new connection, accept the connection request and spawn a thread to handle it.
2. Perform the handshakes necessary to set up the GSS context with the client.
3. Use the context to unwrap the incoming tokens and pass the result to secure JERI layer.
4. The secure JERI layer will later consult the underlying Jgss connection to find out the client's principal so that it can perform permission checks on the incoming call.
5. If delegation is requested, client credentials will also be provided to server side secure JERI layer.
6. When secure JERI passes the server reply to Jgss, it uses the GSS context to wrap the reply into GSS tokens and sends them over the network.

3.1 Secure JERI request/response multiplexing

GSSAPI was designed for situations in which two peers open a network connection, establish a GSS context over it, and then exchange many messages wrapped as GSS tokens using the initially established GSS context. Hence, the relative high cost of establishing a GSS context is amortized over all exchanged messages.

Secure JERI traffic is quite different. secure JERI requests are often short messages, and the security constraints to apply to the underlying network traffic may change on a per method invocation basis. In a simple-minded implementation, one could open a new Jgss connection for each remote method call, but the cost of establishing new GSS contexts at such high frequency would be prohibitively high.

In Jgss, requests and responses for multiple remote method invocations are multiplexed over the same Jgss connection, assuming their constraints can all be satisfied by the connection. SecureConnectionManager, a helper class provided in secure JERI, is used in Jgss to implement secure JERI request multiplexing and connection pooling.

4. Implementation

Jgss is implemented by four classes in the package com.sun.jini.temp.davis.jeri.security.jgss: KerberosServerEndpoint, KerberosEndpoint, KerberosUtil, and JgssTrustVerifier. Following the transport model used in secure JERI, we refer to the server and client parts of Jgss as server and client endpoints, respectively.

4.1 Server endpoint

The Jgss server endpoint is implemented by the KerberosServerEndpoint class, along with supporting methods and nested classes defined in the KerberosUtil class.

4.1.1 KerberosServerEndpoint

KerberosServerEndpoint implements the SecureServerEndpoint interface:

```
public interface SecureServerEndpoint {
    boolean supportsConstraints(SecurityConstraints constraints);
```

```
    SecureServerEndpointListener listen(SecureRequestHandler handler)
    ...
}
```

During service export, a setup thread instantiates a KerberosServerEndpoint object and invokes its listen method, which starts a server thread and listens on a port for client connection requests. A handler is passed to the server endpoint which can be used to service the JERI requests received at the endpoint. The listen method returns the caller a listener object through which the caller can obtain a client endpoint, or Jgss client part that can be downloaded into a client as part of a JERI proxy. The caller can also shutdown the server endpoint by invoking the close method on the listener.

The supportsConstraints method takes a set of security constraints, and returns true if the constraints can be supported by this provider using the credentials currently associated with the endpoint, or false otherwise.

4.1.2 ServerConnection

ServerConnection is a nested class defined in KerberosServerEndpoint. Each time a new TCP socket connection is established with a client, the server endpoint creates an instance of this class to handle the connection. This class implements the SecureServerConnection interface:

```
public interface SecureServerConnection {
    InputStream getInputStream()
    OutputStream getOutputStream()
    Subject processRequestData(InputStream in, OutputStream out)
    void checkPermissions(Subject clientSubject)
    boolean satisfiesConstraints(Subject clientSubject,
        SecurityConstraints constraints);
    ...
}
```

The upper secure JERI layer reads in incoming requests and write outgoing responses using the streams returned by the getInputStream and getOutputStream methods of the corresponding server connection.

When secure JERI receives a request, it first invokes the callback method processRequestData on the corresponding server connection object to determine the principal of the client making this request. Then, it invokes the connection's checkPermissions method to check whether the required authentication permission has been granted to accept this client's requests. If the permission has been granted, secure JERI then reads out the object identifier (OID) and method hash of the request, finds the security constraints associated with this OID and method, and verifies that the constraints have been satisfied by the underlying server connection by calling the connection's satisfiesConstraints
method.

If (and only if) all these steps succeed, then secure JERI starts to unmarshal the parameter objects of the request. Since unmarshalling can trigger downloading of (untrusted) foreign code, it involves some security risks (in particular, of denial-of-service attacks); hence, unmashalling is performed only after the security checks outlined above have completed.

4.2 Client endpoint

The Jgss client endpoint is implemented by the KerberosEndpoint class along with some supporting methods and nested classes defined in KerberosUtil class.

4.2.1 KerberosEndpoint

KerberosEndpoint implements two interfaces, one of them is:

```
public interface SecureEndpoint {
    SecureCallContext getCallContext(SecurityConstraints constraints,
                                     Subject clientSubject)
    SecureOutboundRequest newRequest(SecureCallContext context)
}
```

As described in Section 2, to decide which provider's endpoint to use for a specific remote call, secure JERI first consults each available endpoint by invoking its getCallContext method, passing to it the set of constraints configured for the remote method. An endpoint returns a non-null call context object only if it can support all required constraints in the constraint set. secure JERI can query how many preferred constraints the endpoint will be able to support by invoking methods on the returned call context object.

If secure JERI chooses a Jgss endpoint for a given remote call, it acquires an outbound request object from the endpoint by invoking its newRequest method, passing the call context object previously returned by this endpoint as a cookie. The outbound request object is then used by secure JERI to write out the request and read in the request of the corresponding remote method invocation.

Instead of constructing the outbound request objects itself, KerberosEndpoint delegates the task to a secure JERI helper class, SecureConnectionManager. This manager provides support for multiplexing multiple remote method calls over one underlying connection. The underlying Jgss connections, however, are supplied by KerberosEndpoint.

The other interface the KerberosEndpoint implements is SecureConnectionEndpoint:

```
public interface SecureConnectionEndpoint {
    SecureConnection chooseConnection(Collection connections,
                                      SecureCallContext context);
    SecureConnection openConnection(SecureCallContext context)
    ...
}
```

A KerberosEndpoint instance passes itself to the SecureConnectionManager, which interacts with it through the SecureConnectionEndpoint interface. When the manager needs to construct an outbound request object for a remote method call, it invokes the chooseConnection method on the connection endpoint first to find out whether there is an established connection that can be reused; only if that call returns a null will the openConnection method of the endpoint be invoked to establish a new Jgss connection with the server.

4.2.2 ClientConnection

ClientConnection is a nested class defined in KerberosEndpoint. Instances of ClientConnection are provided to the SecureConnectionManager by KerberosEndpoint when the manager calls the newRequest method of the client endpoint. This class implements the SecureConnection interface:

```
public interface SecureConnection {
    InputStream getInputStream() throws IOException;
    OutputStream getOutputStream() throws IOException;
    ...
}
```

Instances of ClientConnection are used by the SecureConnectionManager to access the underlying stream based connection provided by Jgss. Kerberos authentication and encryption is performed by Jgss at a lower level; the bytes written to and read from these streams are in cleartext.

4.3 Security constraint verification

There are 14 security constraint types currently defined in Overture, out of which Jgss supports 11. They are:

Integrity
Confidentiality
ClientAuthentication
ServerAuthentication
Delegation

ClientMinPrincipal
ClientMinPrincipalType
ClientMaxPrincipal
ClientMaxPrincipalType
ServerMinPrincipal

ConstraintAlternatives

The first five type of constraints are simple constraints - i.e., if specified, they can only have a YES or NO value. The next five constraints are principal-related constraints. They each contains a set of principals or principal types representing what principals or principal types the client or server must use to authenticate to each other over a provider connection. The last constraint in the list specifies a set of alternative constraints of the same type. A ConstraintAlternatives is considered to be satisfied if any one of its contained constraints is satisfied.

Each constraint class implements a reduceBy method, which takes another security constraint instance as parameter. A null reference is returned if the two constraints conflict, i.e., if there is no way to satisfy both of them at the same time.

To check whether a given set of constraints can be satisfied by a Jgss connection, each required constraint in the set is checked in turn. The set of constraints cannot be satisfied if any one of its required constraints cannot be satisfied.

Listed below are the steps Jgss takes to check whether a single constraint can be satisfied by a Jgss connection:

1. If the constraint being checked is an instance of ConstraintAlternatives, then the alternative constraints are checked one at a time until one can be satisfied, in which case ConstraintAlternatives constraint can be satisfied, or no alternatives remain, in which case the ConstraintAlternatives constraint cannot be satisfied.
2. If the constraint is one of the implicit constraint of the Jgss provider (Integrity.YES, ClientAuthentication.YES, ServerAuthentication.YES, ClientMinPrincipalType=KerberosPrincipal.class, and ClientMaxPrincipalType=KerberosPrincipal.class), then the constraint can be satisfied.
3. The client and server principals of the Jgss connection are used to check other min/max principal constrains to determine whether they can be satisfied.
4. The reduceBy method of the constraint is called with the implicit constraints and required constraints that have been checked, If any of them returns a null reference, than the constraint can not be satisfied.

4.4 Caching

Jgss provider does extensive permission and network security constraint checking. The implementation makes use of various caching opportunities to improve performance. The following is a list of the cached items:

1. Access control contexts - remote method calls tend to be repeatedly invoked in the same access control context. By caching access control contexts, Jgss can skip permission checks when the same access control context has passed the permission check before.
2. Security constraints, client subjects (which contain client principals and their credentials), and corresponding call contexts. Since the same remote method can be invoked multiple times, related methods are often associated the same set of security constraints, and client subjects, once set, are not likely to change. The chance that a Jgss endpoint will encounter a set of security constraints and client subjects which it has seen before is quite high. In this case, a cached copy of the previously generated call context object can be reused, saving Jgss endpoint the computation necessary to redo the checks. Call context reuse must be handled with caution so as not to open security holes.

A SoftCache, which is a customized hash table implemented in Jgss, is used to handle the cache described in (2). LRU replacement policy is used when the number of inserted entries exceeds the cache capacity. SoftReferences are used so the cache does not maintain any hard links to its cached objects, allowing the garbage collector to recycle those cached objects if needed.

5. A working example

To demonstrate secure JERI using Jgss, in this section we show how a simplified broker service can be implemented and deployed. The key classes in this example are Server, Client, and BrokerService. Server implements the BrokerService interface, which is a remote interface.

```
public interface BrokerService extends Remote {
    String getInfo() throws RemoteException;
    float getPrice(String ticker) throws RemoteException;
    boolean buy(String ticker, int shares, float priceLimit)
        throws RemoteException;
    boolean sell(String ticker, int shares, float priceLimit)
        throws RemoteException;
}
```

There are two configuration files, krb-client.config and krb-server.config for this example. Client and server constraints are specified in them. In addition, krb-server.config specifies that KerberosServerEndpoint is used as server endpoint, which means using Jgss as provider in JERI. Full listing of source code and configuration files are included in appendix.

The server constraints are:

1. No constraints for getInfo, which means that the server has no security concerns when sending out the public information of the broker.
2. Integrity.YES and ClientAuthentication.YES for getPrice. This means that the server is willing to give out the price quote to any client who can authenticate as a principal known to the Kerberos server, and the server wants to ensure integrity of the call data.
3. Integrity.YES, Confidentiality.YES, ClientAuthentication.YES, and ClientMinPrincipal(KerberosPrincipal("xjiang@DAVIS.JINI.SUN.COM")) for buy and sell. This means that the server is very concerned with the security of these transactions, and requires secure JERI to provide privacy and integrity protection for its network communication with the client, and the client has to at least be able to authenticate as "xjiang@DAVIS.JINI.SUN.COM", when handling the buy or sell remote calls.

The client constraints are:

1. Integrity.YES, Confidentiality.NO, ServerAuthentication.YES, and ServerMinPrincipal(KerberosPrincipal("test@DAVIS.JINI.SUN.COM")) for getInfo. This indicates that the client want to be sure that the broker information is provided by a service that can authenticate as "test@DAVIS.JINI.SUN.COM", and that call request/response should be integrity protected, but not encrypted (to save computation).
2. Integrity.YES, Confidentiality.YES, ServerAuthentication.YES, ServerMinPrincipal(KerberosPrincipal("test@DAVIS.JINI.SUN.COM")) for other methods. This means that the client want the corresponding network traffic to be both encryption and integrity protected, and

that the server must be able to be authenticate at least as principal "test@DAVIS.JINI.SUN.COM".

Server code snip:

```
// Read in the configuration
Configuration config =
    Configuration.Provider.get(System.getProperty("config"));

// Get the RMI registry
Registry registry = LocateRegistry.getRegistry();

// Instantiate Server, which implements the BrokerService interface
Server server = new Server();

// Get the exporter specified in the configuration file
// In the configuration file, KerberosServerEndpoint is specified
// as the server endpoint used by the exporter.
Exporter exporter =
    (Exporter) config.getEntry("serverExporter", Exporter.class);

// Export the server and return a stub as the result
BrokerService stub = (BrokerService)exporter.export(server);

// Register the server stub with a registry service
registry.rebind("broker.BrokerService", stub);
```

Client code snip:

```
// The configuration contains information about constraints to
// apply to the downloaded server stub
Configuration config =
    Configuration.Provider.get(System.getProperty("config"));

// Get the registry
Registry registry = LocateRegistry.getRegistry(serverHost);

// Look up the server stub on the registry
BrokerService server =
    (BrokerService)registry.lookup("broker.BrokerService");

// get the constraints specified on a per-method basis at the
// client side
MethodConstraints mc = (MethodConstraints) config.getEntry(
    "clientConstraints", MethodConstraints.class, null);

// Verify the server proxy and associate the client constraints
// with the server stub
if (mc != null)
    server = (BrokerService)Security.verifyProxyTrust(server, mc);

// Make the remote calls
```

```
String info = server.getInfo();
```

6. Conclusion

This paper describes the design and implementation of Jgss, a Kerberos security service provider for secure JERI. Overall we feel that the secure JERI framework is well designed and provides enough hooks and helper classes for the underlying providers to be implemented and plugged into the framework smoothly. Also the Kerberos support in Java through GSSAPI provides a reasonable interface for provider like Jgss to be implemented. One problem encountered in Jgss implementation is that if there are multiple Kerberos ticket stored in a client subject, GSSAPI does not provide a method for determining exactly which ticket is used in its GSSCredential. This complicates the logic in Jgss to determine whether a cached call context object can be reused.

7. Future work

At this moment, all Jgss classes have been fully implemented and functioning. The focus of the next step is to thoroughly test the features implemented in Jgss. Unit tests will be developed for modules such as the SoftCache and the constraint checkers implemented in KerberosUtil.

More performance tuning is needed. It will be desirable to profile sizable distributed applications and benchmarks on secure JERI with Jgss, to highlight performance bottlenecks where optimization should be focused on.

References

[1] “Generic Security Service Application Program Interface, Version 2, Update 1”, RFC 2743

[2] “Generic Security Service API Version 2 : Java Bindings”, RFC 2853

[3] “Network Objects”, Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber; SIGOPS ‘93

[4] “The Kerberos Network Authentication Service (V5)”, RFC 1510

[5] “Documentation for project Davis and secure JERI”, http://developer.jini.org/exchange/projects/davis/

Appendix

Source code of BrokerService.java:

```
package broker;

import java.rmi.Remote;
import java.rmi.RemoteException;
```

```
public interface BrokerService extends Remote {
    String getInfo() throws RemoteException;
    float getPrice(String ticker) throws RemoteException;
    boolean buy(String ticker, int shares, float priceLimit)
        throws RemoteException;
    boolean sell(String ticker, int shares, float priceLimit)
throws RemoteException;
}
```

Source code of Server.java:

```
package broker;

import com.sun.jini.temp.davis.config.Configuration;
import com.sun.jini.temp.davis.config.ConfigurationException;
import com.sun.jini.temp.davis.export.Exporter;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Server implements BrokerService {

    public static void main(String[] args)
        throws ConfigurationException, RemoteException
    {
        //The configuration contains information about exporting the server.
        Configuration config =
            Configuration.Provider.get(System.getProperty("config"));

        // Get the RMI registry
        Registry registry = LocateRegistry.getRegistry();

        // Create, export, and register the server
        Server server = new Server();
        Exporter exporter =
            (Exporter) config.getEntry("serverExporter", Exporter.class);
        BrokerService stub = (BrokerService)exporter.export(server);
        registry.rebind("broker.BrokerService", stub);
        System.out.println("BrokerService server is ready");
    }

    public String getInfo() {
        return "Daniel's testing broker service info";
    }

    public float getPrice(String ticker) {
        return 89.5f;
    }

    public boolean buy(String ticker, int shares, float priceLimit) {
        return true;
    }
```

```
    public boolean sell(String ticker, int shares, float priceLimit) {
        return true;
    }
}
```

Source code of Server.java:

```
package broker;

import com.sun.jini.temp.davis.config.Configuration;
import com.sun.jini.temp.davis.config.ConfigurationException;
import com.sun.jini.temp.davis.security.MethodConstraints;
import com.sun.jini.temp.davis.security.Security;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    public static void main(String[] args)
        throws ConfigurationException, NotBoundException, RemoteException
    {
        // The configuration contains information about constraints to
        // apply to the server proxy.
        Configuration config =
            Configuration.Provider.get(System.getProperty("config"));

        String serverHost = System.getProperty("serverHost");

        // Get the RMI registry
        Registry registry = LocateRegistry.getRegistry(serverHost);

        // Look up the remote server
        BrokerService server =
            (BrokerService)registry.lookup("broker.BrokerService");

        // Verify the server proxy if constraints are specified
        MethodConstraints mc = (MethodConstraints) config.getEntry(
            "clientConstraints", MethodConstraints.class, null);
        if (mc != null)
            server = (BrokerService)Security.verifyProxyTrust(server, mc);

        System.out.println("\n\nHere comes the results:\n");

        String info = server.getInfo();
        System.out.println("getInfo() returns: " + info);

        float price = server.getPrice("IBM");
        System.out.println("getPrice('IBM') returns: " + price);

        boolean buyResult = server.buy("IBM", 20, 90.0f);
```

```
        System.out.println("buy('IBM', 20, 90.0) returns: " + buyResult);

        boolean sellResult = server.sell("IBM", 20, 95.0f);
        System.out.println("sell('IBM', 20, 95.0) returns: " + sellResult);

        System.exit(0);
    }
}
```

krb-client.config

```
/* Configuration source file for Kerberos client */

import com.sun.jini.temp.davis.config.ConfigurationFile;
import com.sun.jini.temp.davis.security.*;
import com.sun.jini.temp.davis.security.BasicMethodConstraints.MethodDesc;
import com.sun.security.auth.callback.*;
import javax.security.auth.kerberos.KerberosPrincipal;

/* JAAS callback handler to get password information during logins. */
static loginCallbackHandler = new DialogCallbackHandler();

private static pubConstraints =
    new SecurityConstraints(
        new SecurityConstraint[] {
            Integrity.YES,
            Confidentiality.NO,
            ServerAuthentication.YES,
            new ServerMinPrincipal(
                new KerberosPrincipal("test@DAVIS.JINI.SUN.COM")
            )
        }, null
    );

private static privConstraints =
    new SecurityConstraints(
        new SecurityConstraint[] {
            Integrity.YES,
    Confidentiality.YES,
    ServerAuthentication.YES,
    new ServerMinPrincipal(
        new KerberosPrincipal("test@DAVIS.JINI.SUN.COM")
            )
        }, null
    );

/* Method constraints for server proxy */
static clientConstraints =
    new BasicMethodConstraints(
new MethodDesc[] {
    new MethodDesc("getInfo", pubConstraints),
    new MethodDesc(privConstraints) /* default using private Cs */
        });
```

krb-server.config

```
/* Configuration source file for Kerberos server */

import com.sun.jini.temp.davis.jeri.security.*;
import com.sun.jini.temp.davis.security.BasicMethodConstraints.MethodDesc;
import com.sun.jini.temp.davis.jeri.security.jgss.*;
import com.sun.jini.temp.davis.security.*;
import broker.ServerPermission;
import javax.security.auth.kerberos.KerberosPrincipal;

private static pubServiceConstraints =
    new SecurityConstraints(
        new SecurityConstraint[] {
            Integrity.YES,
            ClientAuthentication.YES
        }, null
    );

private static privServiceConstraints =
    new SecurityConstraints(
        new SecurityConstraint[] {
            Integrity.YES,
    Confidentiality.YES,
    ClientAuthentication.YES,
    new ClientMinPrincipal(
        new KerberosPrincipal("xjiang@DAVIS.JINI.SUN.COM")
            )
        }, null
    );

/* Exporter for the server proxy */
serverExporter =
    /* Use secure exporter */
    new SecureJeriExporter(
/* Use Kerberos transport */
new KerberosServerEndpoint(),
/* Require ServerPermission */
new SecureUnicastRefFactory(
    ServerPermission.class, SecureDispatcher.MARSHAL_ASIS),
new BasicMethodConstraints(
    new MethodDesc[] {
                new MethodDesc("getInfo", null), /* no constraint */
                new MethodDesc("getPrice", pubServiceConstraints),
                new MethodDesc(privServiceConstraints)
            }
        )
    );
```