FADA tutorial

Revision history

Date	Version	Description	Author
30 June 2004	1.0	Initial	Javier Noguera

Table of contents

Foreword

...

We hope you find the information contained here complete and useful.

Enjoy it.

The FADA development team.

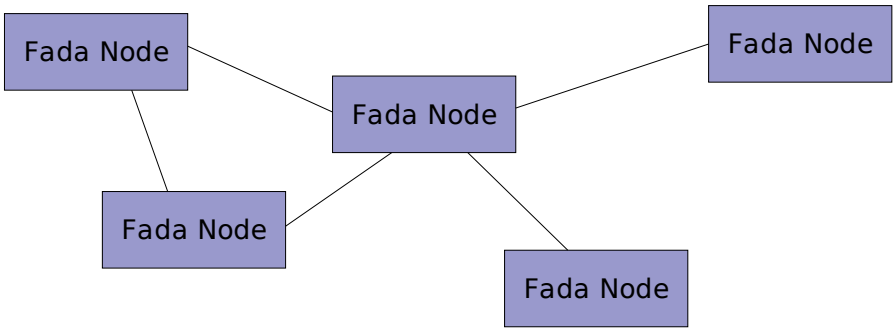
Introduction

This user manual tries to help programmers to work with FADA technology. It includes examples and explanations about the most important FADA scenarios.

In the first chapter there is a short explanation about FADA itself and about FADA from the point of view of the client and the server. If you need further detailed information about FADA architecture you will find it in the file named **Core FADA**. It explains in depth the bases of FADA, FADA nodes communication, etc.

In the second chapter we there is a discussion about the use of FADA and the cases in which FADA may be necessary.

In the third chapter there are some FADA scenarios: do-it-yourself, fadagen and fada J2EE. We will implement a solution in all three scenarios using FADA with the same example. You will find all the code for the three examples in the Annex A. It can be used as a reference.



FADA network architecture

What is FADA ?

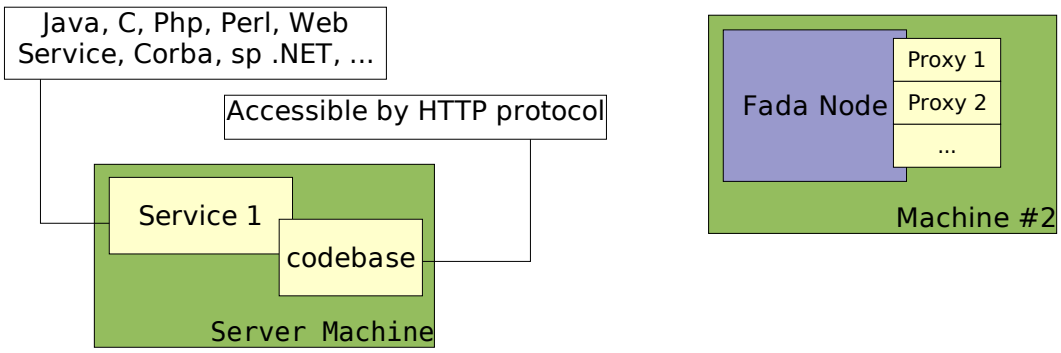
FADA stands for “Federated Advanced Directory Architecture”. It is a virtual Lookup Server in the sense that different Lookup Servers (FADA nodes) will work together to provide the LookupServer functionality from any entry point. Also, any of these Lookup Servers will cooperate with the rest to find implementations of services.

The FADA is a truly distributed system, in the sense that there is no central authority or common communication channel.

The FADA holds proxies for services. A proxy is a Java class that acts as a mediator between client and service provider performing communication with a real service, and that is downloaded at run-time by clients. Clients use the public methods on the proxies to access the services. These public methods are specified in Java interfaces that service proxies implement. A FADA node is a service that acts as an entry point and a container to the distributed database of services.

Service providers must implement Java classes that act as a gateway to use their services, which may be written in Java or not. In case they aren't, the Java class they provide to the FADA clients, the service proxy from now on, can use whatever method to communicate with the service. For example, the service could be written in C, or it could be accessed through http, and the service proxy could open sockets to the proper ports to communicate with the service.

Note that these proxy objects will be executed in the client's machine, so this fact must be taken in account when designing and implementing service proxies.



Situation after registration

How FADA works

Let's see now, more in detail, one FADA node and the steps used by the server to register a service and by the client to look for a service.

One server who wants to register a service in a FADA node needs:

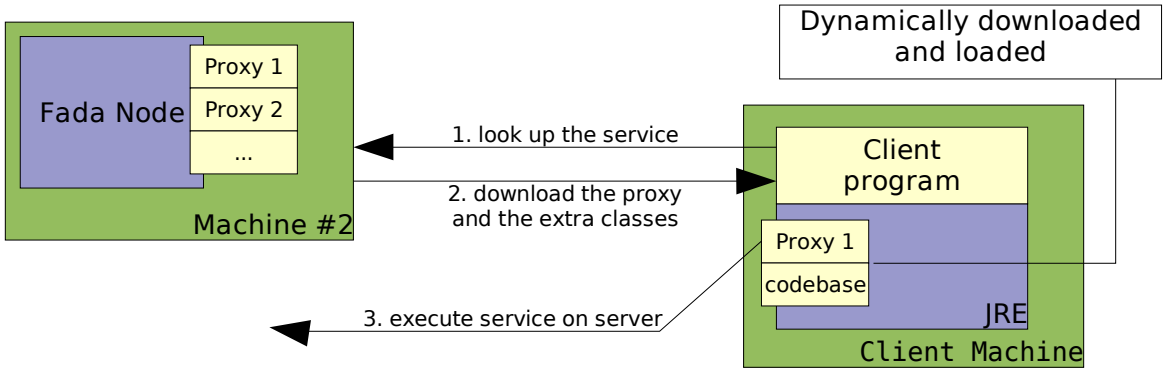
1. **To have a service running**. It is necessary to difference between *service* and *proxy*. The service must be running over any machine and can be implemented with any language or technology. The proxy is a little piece or code that knows how to communicate with the service.

2. **To define a service interface**. This interface will define the methods that the client can execute to communicate with the service. This interface is accessible to all clients and must be coded in Java language.

3. **To code a service proxy**. The proxy is the implementation of the *service interface* and it is the way in which the client executes the service on the server side. The proxy class will be executed on the client side, this has to be taken in account when implementing it. It has to be coded in Java language.

4. **To register the proxy in one FADA node**. When the proxy is uploaded to one FADA node it is available for any node within the FADA network. Clients now can discover and use the service.

5. **Make accessible a jar file with libraries**. Proxy implementation is not known by the clients so it has to be downloaded. The server needs to make accessible all extra classes in a jar file to be used by service proxy. The location of this jar file is named codebase.



Situation after discovering

One client who wants to use a service (previously registered in the FADA network) needs to follow the next steps:

1. **To have the service interface in the classpath**. Client does not need to know anything about service or proxy implementation but service interface must be accessible for compiling.

2. **Discover the service**. To discover the service, client needs to know the name of the service interface (and optionally some of its entries) and the address of one FADA node that is part of the FADA network.

3. **Download and execute the proxy**. After discover the service, FADA framework will automatically download and load the proxy in memory using the codebase. Proxy is now ready to be used by client therefore client will communicate directly with the service running on the server.

Why FADA

FADA allows the servers to publish one or more services and make them accessible to everyone. On the other hand, FADA allows to the clients to find any published service and use it.

Client does not need to know how the service is implemented neither where it is. Client only must find the service and, this is the good thing, FADA assures us that the service is working.

We can imagine, for example, that we usually use FADA to access to a printer service located over Internet (ie. A photo-shop store). One day the printer is moved (its IP and name have changed). This is not a problem, because a new proxy will be registered by the server. If the printer is changed and a new protocol is needed, it is not a problem either, because a new proxy will be implemented and re-registered in the FADA network. If the server is down and unaccessible the proxy will be unregistered automatically and no client will be able to use that printer.

Thinking of FADA we can list situations in which it can be used and in which it improve the development, the performance and the accessibility:

- If you need a distributed system over Internet (WAN) where many servers offer many services for clients.

- If the service nature is noncontinuous. In this scenario, clients cannot access to the service at any moment. FADA assures the client will not find the service if it is down.

- If the service implementation can often change. The proxy will code the new implementation without client knowledge.

- If you have a distributed system with different Operating Systems and/or different applications languages and you want to access in the same way them all... of course, in Java.

- If you want to connect two or more different applications. To publish their service interfaces is the easiest way to integrate them.

FADA Examples

We will discuss now some scenarios in the use of FADA. This chapter is a technical reference for programmers where they can see how to use FADA API.

The sample code can be found at the end of the document (Annex A)

You can access directly to the most suitable scenario:

- **Do-it-yourself**. The easiest way to understand FADA. There is a standalone service with and a customized protocol and it is necessary to code and register the proxy.
- **Fadagen**. The easiest way to create a service. There is no service nor proxy but FADA will code them both for us.
- **J2EE**. We can code the service like a servlet in an Application Server and register it at start up.

We will use the same example in all three scenarios: “There is a server where users can be added or removed. A client looks for the service (discovers it over FADA network) and add his/her username. Clients can also get a list of usernames from the server”.

Do-it-yourself scenario

In the first case we begin with a server running and waiting for clients. It can be coded in any language and it can implement any protocol (http, simple xml, soap...). To register this service in the FADA network we only need to define the service interface and to code the proxy.

Server

In our example the server implements it own communication protocol: One character that indicates the action and one argument if needed. There are four possible actions:

- Add a user: “A <user_name>”
- Remove a user: “R <user_name>”
- List usernames: “L”. Returns a list of usernames separated by “#” token.

- Quit: “Q”. This stops the server.

We can run the service before registering the service. We will leave it listening on port 2727.

```
java net.fada.examples.uptoyou.Userlist 2727
```

Users who do not want to use FADA can access to the server now. We will still wait until the service will be registered.

Service interface

Now we have to define the server interface. This interface is used by clients to access to the server, so, we need to define the methods to be used by clients. In this example we will authorize the clients to add, remove and get a list of usernames, but they will not stop the server.

```
public interface RemoteUserList {
    public void register (String name);
    public void unregister (String name);
    public String getRoomName ();
    public String [] getList ();
}
```

We can notice three important aspects of the interface definition: First, we have not define a quit method, so we will not allow users to stop the server. Second, the `getList` method will return an array of usernames instead of a usernames list separated by “#” token. And third, we have added an extra method called `getRoomName` not known by the server but, in this case, known by the proxy.

Client must know the interface before compiling so it has to be well done and preview future changes. Changes in the implementation (proxy) will not affect to the client, changes in the interface will do.

Now we have the definition of the service, let's implement it with the proxy class.

Proxy

The proxy class implements the `RemoteUserList` interface and, of course, `java.io.Serializable`. If the proxy does not implement `java.io.Serializable` it will not be able to be register in the FADA node.

In our example, the proxy class will open a socket to the server for each action (except `getRoomName` method) . As you can see, to implement the proxy it is necessary to know how to communicate with the server. It is also necessary to know the address and port

where server is listening. This information will be st in the proxy in the constructor along with the `roomName`.

An empty constructor is also necessary for serialization proposes.

```
public class UserListProxy implements RemoteUserList, java.io.Serializable
{
    private String roomName;
    private String ip;
    private int port;

    /**
     * Empty constructor necessary for serialization
     *
     */
    public UserListProxy ()
    {
        /* nop */
    }

    /**
     * Creates a new instance specifying address, name and an extra parameter
     * called roomName defined in register time.
     */
    public UserListProxy (String roomName, String address, int port)
    {
        this.roomName = roomName;
        this.ip = address;
        this.port = port;
    }
}
```

There are two simple methods without return value: `addUser` and `removeUser`.

```
    public void addUser(String name)
    {
        send ("A " + name);
    }

    public void removeUser(String name)
    {
        send ("R " + name);
    }
```

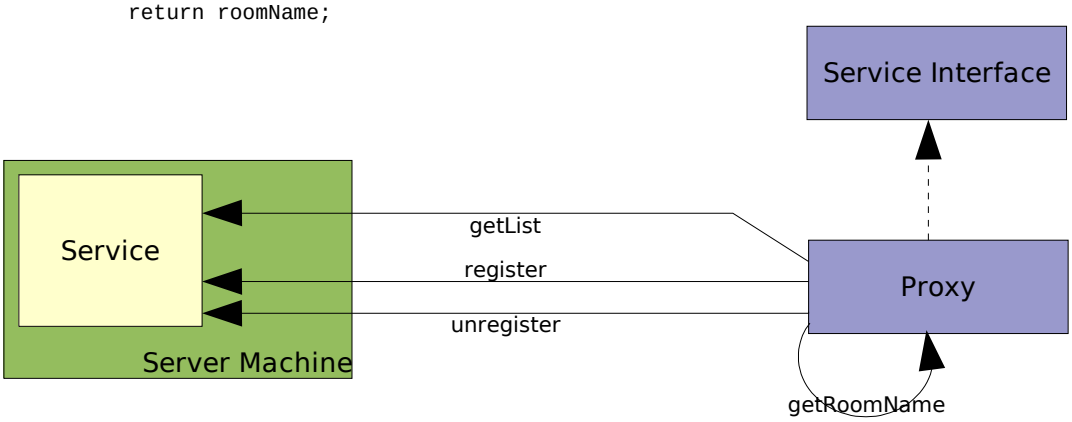
The `getList` method has to transform the server output to adapt it to the interface definition. The server will return a separated by “#” String with usernames. We will change this return value in a string array.

```
    public String[] getList()
    {
        String[] response = null;
        int i = 0;

        StringTokenizer namelist = new StringTokenizer(send ("L"), "#", false);

        // If there are results
        if (namelist != null) {
```

```
            // Allocate array size for response and fill it
            response = new String [namelist.countTokens()];
            while (namelist.hasMoreTokens()) {
                response[i++] = namelist.nextToken();
            }
        }

        return response;
    }
```

Finally we have a method called `getRoomName` that have no relation with the server. This is an extra known information when the server registers the proxy in FADA network. We will get this information stored in the server proxy and no communication with the server will be needed.

```
public String getRoomName()
{
    return roomName;
```



Proxy implementation

```
}
```

Registering the proxy

Finally the server must register the proxy in the FADA network to make it accessible by the clients. The easy way to do this is to use the `net.fada.toolkit.FadaHelper` class.

First of all we need to create a proxy instance. This instance will be sent to the FADA network and used by clients to access directly to the server. We must specify the address and the port to the server.

```
String roomName = “Example do-it-yourself”;
String serverAddress = “my.machine”;
int serverPort = 2727;
UserListProxy proxy = new UserListProxy (roomName, serverAddress, serverPort);
```

To register a service we will use the `register` method, located in `FadaHelper` class. Let's see it:

```
public FadaServiceID register(FadaInterface fp,
                              java.io.Serializable item,
```

```
                FadaServiceID id,
                java.lang.String[] entries,
                long leasePeriod,
                SecurityWrapper wrapper,
                java.lang.String annotation,
                RenewalEventListener listener)
        throws FadaException,
                java.io.IOException,
                java.lang.NullPointerException,
                java.security.InvalidKeyException
```

- **FadaInterface fp**. This is the FADA node proxy in which service will be registered. When we are using FADA through a WAN it is necessary to know the address and the port in which the FADA node is listening. If we are using FADA through a LAN there are multicast methods to find any of the FADA nodes in it (we will see it ahead).
- **java.io.Serializable item**. This is the proxy, the serializable object to be registered in the node. This is the piece of code used by clients.
- **FadaServiceID id**. This is the identifier of the service. The `register` method returns a FadaServiceID, we can use this id here for re-registration purposes. The first time we need to register a service this parameter can be null.
- **java.lang.String[] entries**. We can register the service with some entries to be found by the client. When a client is searching for a service he can specify the service interface and/or any of the entries it has been registered with. This is an easy way to distinguish between different implementations or different providers of the same interface.
- **long leasePeriod**. Periodically and automatically the registration class sends signals to the FADA node which stores the service to renew it. The *leasePeriod* is the maximum time passed between two signals. If signal is not send, the service will be automatically unregistered from FADA node. It is expressed in milliseconds.
- **SecurityWrapper wrapper**. We can sign the object with the security wrapper (we will see it ahead). If no sign is needed null is possible.
- **java.lang.String annotation** (a.k.a. codebase). We know that the proxy class will be registered in a FADA node and later downloaded by client to be executed, but proxy class and other classes used by the proxy are not in the client classpath. Server must publish the codebase (a jar file with the extra classes) usually

over a Web Server. Annotation is the full URL to locate the extra classes.

- **RenewalEventListener listener**. If the lease can not be renewed, this class is called to do something, for example, to re-register it, send an email to the boss, etc. We will see it ahead.

The first thing we need to do is a `FadaInterface`, with the node address.

This code will register the proxy interface created before in the FADA node located on www.fadanet.org:2002. In the code you can see how we create a new instance of `FadaHelper` class with a `FadaLeaseRenewer` object as a parameter. This parameter will control when to send the renew signal. One good practice is not to try to change this class.

```
String [] entries = new String [] {"userlist_yourself", "do-it-yourself"};
String codebase = “http://my.server/classes.jar”;
try {
    // Prepare the FadaHelper instance
    FadaHelper helper = new FadaHelper( new FadaLeaseRenewer() );

    // register service in a FADA node.
    FadaServiceID id = helper.register(
        new FadaLookupLocator (“www.fadanet.org:2002”).getRegistrar(),
        proxy,
        null,
        entries,
        10000L,
        null,
        codebase,
        null);
}
catch (Exception e) {
    // TODO: Something
}
```

Codebase parameter is needed because the proxy uses `UserListProxy` class and the client needs to find this class and load it. Codebase points to a .jar file that contains, at least, `UserListProxy` class.

The service is registered in the FADA network but... what happens with the renewal? In this example we have launched the server before registering the service inside the FADA node. This is good for to understand the different parts of a FADA architecture but there is a problem. When register class finish its execution, the lease will not be sent and and the service will be unregistered.

If it is possible, the easy way is to register the service at the same time we start up the server. If it is not possible we can do something

else. The register thread must not finish while the server is running. In our example we know were the server is listening so we can see if it is running.

```
// We will wait till server is up. If server is down, a exception is thrown
while (true) {
    Socket s = new Socket (serverAddress, serverPort);
    s.close();
    try {
        Thread.sleep(10000);
    }
    catch (InterruptedException e) { /* nop */ }
}
```

Client

Once the proxy has been published, any client can use it. Client only needs to look for the proxy along the FADA network.

There are some things client needs to know: the location of one FADA node address, and the name of the interface he wants to use. Maybe it is necessary to know some *entries* that the server used to register the service, specially if there are many proxies implementing the same interface registered in the FADA network.

To look for a service we will use the static `lookup` method, located in `FadaHelper` class. Let's see it.

```
public static java.lang.Object[] lookup(FadaInterface fp,
                                        java.lang.String[] entries,
                                        FadaServiceID id,
                                        java.lang.String[] servTypes,
                                        int maxMatches,
                                        long timeout,
                                        java.security.cert.X509Certificate cert)
                                 throws FadaException,
                                        java.io.IOException,
                                        java.lang.ClassNotFoundException,
                                        java.lang.NullPointerException,
                                        java.security.InvalidKeyException
```

- **FadaInterface fp**. Like in the `register` method.
- **java.lang.String[] entries**. Array of entries that we want to use in the search. Each interface returned will contain all these entries (AND operation).
- **FadaServiceID id**. Like in the `register` method.
- **java.lang.String[] servTypes**. The name of any interface the proxy can implement. Each interface returned will implement all these entries (AND operation).

- **int maxMatches**. The return value of the lookup method is an array of proxies. We can specify a maximum of results to be returned.

- **long timeout**. As the FADA network can be very wide, we must indicate the maximum time we want to wait while FADA makes the search. It is expressed in milliseconds.

- **java.security.cert.X509Certificate cert**. When a server registers a service, can attach a certificate. User can perform a search over the FADA network indicating the certificate trusted by him. This is the way clients can trust they are executing secure code.

In our example the interface is named `net.fada.examples.uptoyou.RemoteUserList`. We will search for ten seconds as a maximum and we will not specify any entry. After obtaining the result array we will take the first ones.

```
Object[] proxies = FadaHelper.lookup(
                        fadaUrl,
                        null,
                        null,
                        new String[] { "info.techideas.uptoyou.RemoteUserList" },
                        1,
                        10000L);
```

We can now execute any of the interface methods:

```
// Cast is needed
RemoteUserList service = (RemoteUserList) proxies[0];

service.register("bob");
String [] list = service.getList();
...
```

Fadagen scenario

In this second example we have no server nor protocol defined yet. We will use fadagen to automatically create the server and the proxy. Once the process will be finished, we will have a server listening on a port and a proxy that communicates with it using serialization.

It will be more difficult to integrate our server with other clients but it will be the easiest way for us.

This example implies changes in the server and in the client side. At the end of the section we will explain how to do this changes without affecting the client side.

Service interface

The server interface suffers few changes: it has to extend `net.fada.remote.Remote` interface and each method needs to throw a `net.fada.remote.RemoteException` exception.

```
public interface RemoteUserListWithStub extends net.fada.remote.Remote
{
    public void addUser (String name) throws net.fada.remote.RemoteException;
    public void removeUser (String name) throws net.fada.remote.RemoteException;
    public String getRoomName () throws net.fada.remote.RemoteException;
    public String [] getList () throws net.fada.remote.RemoteException;
}
```

Server

To code the server we only need to implement the interface without thinking of socket or communication. Later we will create the stub and skeleton classes that will perform remote communication.

Before continuing some words about the *stub* and the *skeleton* classes are necessary. These classes are used by Java to perform a Remote Method Invocation (RMI). The stub and the skeleton classes mask the communication protocol between two classes located in different machines over the net. Both classes are creates automatically by helper programs. You will find further information at http://java.sun.com/products/jdk/rmi/

Our server needs to implement the interface defined before and to extend `net.fada.remote.RemoteObject` class.

```
import net.fada.remote.RemoteObject;

public class UserList extends RemoteObject implements RemoteUserListWithStub
{
    private String roomName;
    private Collection names = new ArrayList ();
}
```

Implementation is now pretty simple. We only need to implement a simple java class. Notice that we have decided to include *roomName* as part of the server.

```
public void addUser(String name) throws RemoteException
{
        if (!names.contains(name)) {
            names.add(name);
        }
}


public void removeUser(String name) throws RemoteException
{
    names.remove(name);
}
```

```
public String getRoomName() throws RemoteException
{
    return roomName;
}

public String[] getList() throws RemoteException
{
    String [] result = new String [names.size()];
    int i = 0;

    for (Iterator it = names.iterator(); it.hasNext();) {
        result [i++] = (String) it.next();
    }

    return result;
}
```

A constructor is also necessary, it will receive *roomName* as a parameter.

```
public UserList (String roomName)
{
    this.roomName = roomName;
}
```

Once the server has been coded we will generate stub and skeleton classes. We will use `fadagen.jar` library included in FADA distribution. `fadagen.jar` library contains a main class that receives as a parameter the class from which we want to obtain the skeleton and stub classes. The classes obtained are source files (.java) located in the same directory where original class file was. These classes have to be moved, if necessary, and compiled. `fada-toolkit.jar` is also necessary to be in the classpath.

```
java -jar ../lib/fadagen.jar \
          -classpath ../lib/fada-toolkit.jar:. \
          info.techideas.fadagen.UserList
```

`UserList_Skel.java` and `UserList_Stub.java` have been created.

Proxy

In the first and easy example of fadagen we will not use a real proxy. Instead we will use the `UserList_Stub` class as a proxy. At the end of this section we will explain how to implement a real proxy that implies no change in client side.

Registering the proxy

Before registering the server we need to run it. Because we have extended `net.fada.remote.RemoteObject` we have an `export` method. This method takes an instance of two classes, which are implementations of the interfaces ServerTransport and

ClientTransport. The FADA software bundle offers a default implementation for each interface, using HTTP as the transport layer. The implementation of the transport layer is open, and can be freely modified by providing a different implementation of the ServerTransport and ClientTransport interfaces.

ServerTransport receives two parameters: the port to be used by the server and the endpoint of the service.

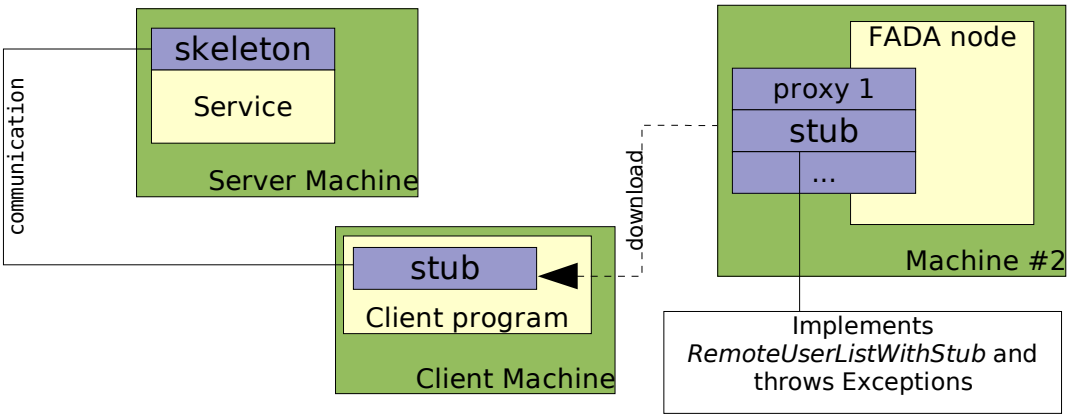
Client Transport receives only one parameter, the full URL in which the server will be listening.

```
// Some harcoded parameters
String roomName = “Example fadagen”;
String address = “my.machine”;      // server Address
int port = 2727;                    // port Address
String endpoint = “/example/fadagen”;

// Create a server instance
UserList myRegister = new UserList (roomName);

// Create transport classes
ServerTransport st = new ServerTransportImpl (port, address);
ClientTransport ct = new ClientTransportImpl ("http://" + address + “:” +
                                                port + endpoint);

// Return the stub (our proxy)
RemoteStub stub = myRegister.export(st, ct);
```

At this moment the server is running and we only need to register the proxy in the FADA network. We will register the proxy with one entry named “userlist_fadagen”.

```
// Prepare the FadaHelper instance
FadaHelper helper = new FadaHelper (new FadaLeaseRenewer());

// register service in a FADA node.
FadaServiceID id = helper.register(
new FadaLookupLocator (fadaAddress).getRegistrar(), // where to find a node
          (Serializable) stub, // proxy (in this case created with fadagen)
          null,                //FadaServiceId (if it was register before for example)
          entries,             // entries that help the user to find the service
          10000L,              // Renew lease period (in millis)
          null,                // SecurityWrapper
          codebase,            // codebase where client cant find libraries
          null);               // What to do if FADA cannot renew the proxy
```

Server is now running and the service is registered.



Registering the stub

Codebase parameter is needed because the proxy uses `UserList_Stub` class and the client needs to find this class and load it. Codebase points to a .jar file that contains, at least, `UserList_Stub` class.

Client

Client is basically the same we explained in the previous example. There are only few changes to be made:

- The service interface has change so names have to be updated.
- The new interface methods now throw exceptions that can be caught.

Coding a real proxy

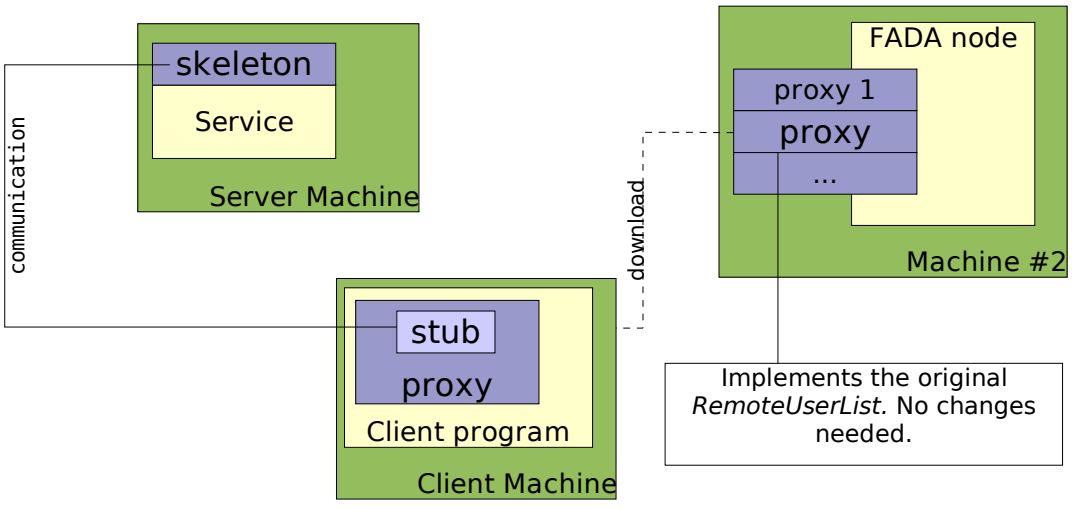
In the previous example we have registered directly the stub class. This is good enough but, this way, the client have to change his code because the *remote interface* has change.

The correct way to do it is to create a proxy class that implements the original interface. This class receives the stub class as a parameter and call its methods when necessary. Another advantage is the possibility of make changes or transformations with the input parameters or with the return value.

In this new scenario you can see that the *RemoteUserListWithStub* interface and the original *RemoteUserList* interface don't need to declare the same methods. Proxy class will manage to agree the

methods using the stub interface. The new proxy implementation will look like this:

```
public class UserListProxy
          implements net.fada.examples.RemoteUserList, java.io.Serializable
{
    private RemoteUserListWithStub stub;
    private String roomName;

    public UserListProxy () {
    }

    public UserListProxy (RemoteUserListWithStub stub, String roomName) {
        this.stub = stub;
        this.roomName = roomName;
    }

    public void addUser(String name) {
        try {
            stub.addUser(name);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    ...
}
```



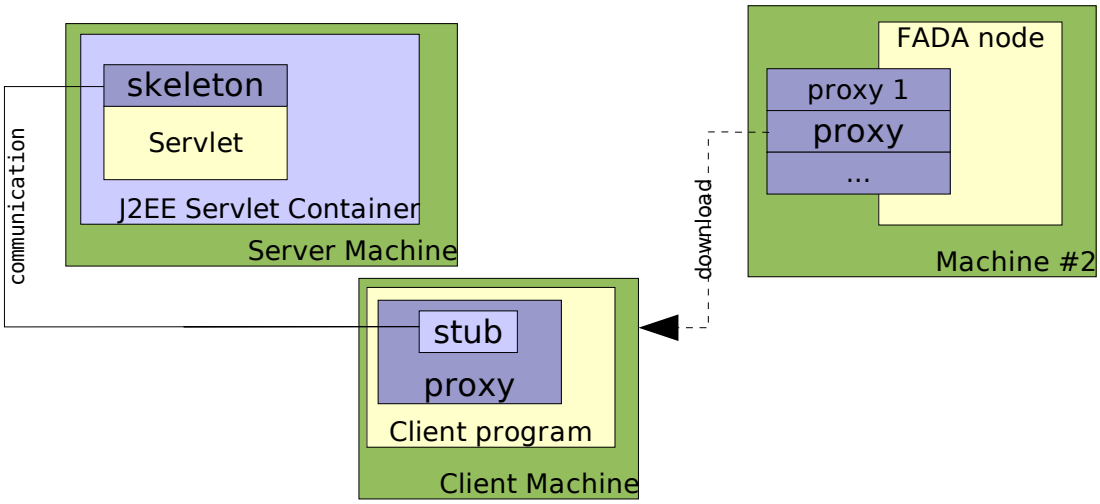
Registering a new proxy that contains the stub class

J2ee scenario

In this third example we will place our service inside an Application Server. We will take advantage of the communication and configuration improvements that the application server have.

We will do this coding our service in a servlet. This servlet will also be the service register so we need assure the service will be registered when the application server will start. The suitable way to do it will be using the *init* method to register the service and modifying the *web.xml* file to indicate to the Application Server that we want to load the servlet at the beginning.

As in the previous case the stub and the skeleton classes will be needed and, also like in the previous case, we can register directly the stub class implementing `RemoteUserListWithStub` interface, or wrap it inside the proxy class than implements the original `RemoteUserList` interface.



Registering a new proxy that contains the stub class

Service interface

The server interface do not suffer any change. It still has to extend `net.fada.remote.Remote` interface and each method needs to throw a `net.fada.remote.RemoteException` exception.

```
public interface RemoteUserListWithStub extends net.fada.remote.Remote
{
    public void register (String name) throws net.fada.remote.RemoteException;
    public void unregister (String name) throws net.fada.remote.RemoteException;
    public String getRoomName () throws net.fada.remote.RemoteException;
    public String [] getList () throws net.fada.remote.RemoteException;
}
```

Server

The service in now a servlet and needs to extends `FadaHttpServlet`. This class extends directly the `HttpServlet` class, needed for servlets. It also have some code necessary to perform the communication between client and server using the HTTP protocol through the POST method.

POST method is used by FADA so our servlet can only use the GET method for our proposes and never overwrite the original *doPost* method (neither the *doService,* which affect directly to the *doPost* method).

Usually servlets will do nothing when *doGet* method is executed and will only attend FADA clients (no Web Server clients). In our example, nevertheless, when the *doGet* method is executed, a list of users is showed in the browser.

```
public class UserList extends FadaHTTPServlet implements RemoteUserListWithStub
{
    private String roomName;
    private Collection names = new ArrayList ();

    public void init(ServletConfig config) throws ServletException
    {
        ...
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Header
        out.println ("<html>Userlist:<br><hr>");

        // Print all names
        for (Iterator it = names.iterator(); it.hasNext();) {
            out.println (" - " + it.next() + "<br>");
        }

        // Footer
        out.println("<hr></html>");
    }


    public void addUser(String name) throws RemoteException
    {
        // Si NO existe, lo guardamos
        if (!names.contains(name)) {
            names.add(name);
        }
    }

    ...
}
```

Proxy

We will code our proxy wrapping the `UserList_Stub` class. First thing is to create `UserList_Stub` and `UserList_Skel` classes using the `fadagen.jar` library.

```
java -jar ../lib/fadagen.jar \
          -classpath ../lib/fada-toolkit.jar:. \
          info.techideas.j2ee.UserList
```

Once created we can code our proxy class as in the previous example:

```
public class UserListProxy
          implements net.fada.examples.RemoteUserList, java.io.Serializable
{
    private RemoteUserListWithStub stub;
    private String roomName;

    public UserListProxy ()
    {
        /* nop */
    }

    public UserListProxy (RemoteUserListWithStub stub, String roomName)
    {
        this.stub = stub;
        this.roomName = roomName;
    }

    public void addUser(String name)
    {
        try {
            stub.addUser(name);
        }
        catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    ...

}
```

Registering the proxy

The best place for registering the service is the *init* method located in the servlet. We can access to the configuration variables defined in the *web.xml* file. Also in this example, the Fadagen transport will be used.

```
public void init(ServletConfig config) throws ServletException
{
    super.init(config);
    String [] entries = new String [] {"userlist_j2ee", "userlist_servlet"};
```

```
        // Getting parameters
        String roomName = config.getInitParameter("room_name");
        String fadaAddress = config.getInitParameter("fada_address");
        String servletUrl = config.getInitParameter("end_point");
        String codebase = config.getInitParameter("codebase");

        try {
            ClientTransportImpl trans = new ClientTransportImpl(servletUrl);

            // Create the stub and the proxy
            RemoteUserListWithStub stub =
                (RemoteUserListWithStub) this.export (this, trans);
            UserListProxy proxy = new UserListProxy (stub, roomName);

            // Prepare the FadaHelper instance
            FadaHelper helper = new FadaHelper( new FadaLeaseRenewer() );

            // register service into a FADA node.
            FadaServiceID id = helper.register(
              new FadaLookupLocator (fadaAddress).getRegistrar(), // where to find a node
              proxy,           // proxy (in this case created with fadagen)
              null,            // FadaServiceId (if it was register before for example)
              entries,         // entries that help the user to find the service
              10000L,          // Renew lease period (in millis)
              null,            // SecurityWrapper
              codebase,        // codebase where cliend cand find libraries
              null);           // What to do if FADA cannot renew the proxy

            System.out.println("proxy was registered into the FADA node");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Some changes have to be done in the *web.xml* file. We will specify the servlet name and the init parameters.

```
<servlet>
    <servlet-name>UserList</servlet-name>
    <servlet-class>net.fada.examples.j2ee.UserList</servlet-class>
    <init-param>
        <param-name>room_name</param-name>
        <param-value>ROOM userlist J2EE</param-value>
    </init-param>
    <init-param>
        <param-name>fada_address</param-name>
        <param-value>localhost:2002</param-value>
    </init-param>
    <init-param>
        <param-name>end_point</param-name>
        <param-value>http://localhost:8080/tutorial/userlist</param-value>
    </init-param>
    <init-param>
        <param-name>codebase</param-name>
        <param-value>http://localhost:8080/tutorial/tutorial_classes.jar</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
    <servlet-name>UserList</servlet-name>
    <url-pattern>/userlist</url-pattern>
</servlet-mapping>
```

Client

Client is the same of the first example because we used a proxy that implements the original service interface.

After a few tests we can check the service with a browser calling the servlet without parameters.

FADA nodes over a LAN

As we have seen, clients who are searching for a service in the FADA network need, to know the address of, at least, one FADA node and the port in with it is running.

If our FADA network is running in a LAN (or it is running in a WAN but we know one of the nodes is in our LAN network), we can use multi-cast technology to discover it.

There are two helper classes that will help us to do the discovering: `DiscoveryListener` and `FadaDiscovery`, both located in the `net.fada.toolkit` package.

`DiscoveryListener` is an interface with one only method to be implemented named *discovered*. This method is called when a new node is found.

`FadaDiscovery` is a helper class to which so many `DiscoveryListener` implementation can be added. When a new `DiscoveryListener` implementation is added, a new multi-cast signal is sent over the net.

Notice that you can specify in the FADA node configuration file if it must be listening for multi-cast signal or not. Nodes not listening will not be discovered at all.

Once the node has been discovered the process continues in the habitual way.

```
public class ClientLan implements DiscoveryListener
{
    static int nodes = 0;

    public static void main(String[] args)
    {
        // New FadaDiscovery class
        FadaDiscovery fadaLan = new FadaDiscovery ();

        try {
            // New DiscoveryListener Implementation
            ClientLan me = new ClientLan ();

            // Send a multi-cast signal
            fadaLan.addDiscoveryListener(me);

            // Waiting...
            System.out.println("waiting 10 seconds...");
            Thread.sleep(10000);

            System.out.println("finished. " + nodes + " nodes was/were found");
        }
        catch (IOException e) {
            e.printStackTrace();
```

```
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }


        /**
         * @see net.fada.toolkit.DiscoveryListener#discovered(FadaLookupLocator)
         */
        public void discovered(FadaLookupLocator locator)
        {
            nodes++;
            System.out.println("FADA node found: " +
                locator.getHost() + ":" + locator.getPort());
        }
    }
```

As you can see, *discover* method is called with a FadaLookupLocator as a parameter. This parameter contains all information about the node.

Renewal Event

When a proxy is registered in a FADA node you can specify what to do if the renewal fails. Usually you need to find another FADA node or try to register in the same FADA node again. Sometimes we will not want to re-register the node but write the failure in a log file, send an email or write a entry in the Database.

`RenewalEventListener` interface have to be implemented and passed as a parameter to the *register* method. It is located in the `net.fada.directory.tool` package.

We have change the Fadagen Register class to catch the renewal event and try to re-register the proxy if it fails. First of all, we need to implement `RenewalEventListener` and to keep some variables as a member variables. The static *main* method now create a class instance. Some code has been replaced by "..." characters.

```
public class RegisterFadagen implements RenewalEventListener
{
    // Variables needed
    private FadaServiceID id;
    private UserListProxy proxy;
    private String codebase;
    private FadaHelper helper;
    private String fadaAddress;


    public static void main(String[] arg) {
        RegisterFadagen register = new RegisterFadagen ();
        cp.register(arg);
    }

    public void register (String[] arg) {
        ...
        // Prepare the FadaHelper instance
        helper = new FadaHelper( new FadaLeaseRenewer() );

        // register service into a FADA node.
        id = helper.register(
          new FadaLookupLocator (fadaAddress).getRegistrar(), // where to find a node
          proxy, // proxy (in this case created with fadagen)
          null,           // FadaServiceId (if it was register before for example)
          entries,        // entries that help the user to find the service
          10000L,         // Renew lease period (in millis)
          null,           // SecurityWrapper
          codebase        // codebase where cliend cand find libraries
          this);          // What to do if FADA cannot renew the proxy
        ...
    }


    /**
     * @see net.fada.directory.tool.RenewalEventListener#eventOccured(RenewalEvent)
```

```
     */
    public void eventOccured(RenewalEvent event)
    {
        System.out.println("reregistering... ");
        try {
            id = helper.register(
                new FadaLookupLocator (fadaAddress).getRegistrar(),
                proxy, // proxy (in this case created with fadagen)
                id,           // FadaServiceId (if it was register before for example)
                new String [] {"re-registered"},             // entries
                10000L,              // Renew lease period (in millis)
                null,                // SecurityWrapper
                codebase,            // codebase where cliend cand find libraries
                this);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

We store the FadaServiceID returned in the first register to use it if a second (or third, or fourth,...) register is necessary. We are doing this to maintain registration consistency. This is a good practice even though it is not necessary.

Security Wrapper

Security can be added to our registered proxies. X509 certificates can be registered within the proxy in the FADA node. These certificates indicate who registered the proxy and who is the certification authority (CA). Clients can use this information to retrieve only the services they trust.

When the client performs the lookup operation, he can specify any `X509Certificate` to find with. If no Certificate is send, all proxies that match with *interface* and *entries* are retrieved. If a Certificate is used, only proxies with the appropriate certificate are retrieved.

Usually two certificates are necessary at registration time: the target certificate and the certification authority one. The target `PrivateKey` is also necessary to create a *hash value* that is attached to the proxy and later checked by the client.

We can modify our Fadagen example to create a SecurityWrapper. Keys and certificates are stored in the */tmp* folder.

```
// Read the private key from a file
byte [] buffer = new byte[2048];
FileInputStream fis = new FileInputStream ("/tmp/cert.key");
int len = fis.read(buffer);
BASE64Decoder dec = new BASE64Decoder ();
byte [] bkey = dec.decodeBuffer(new String (buffer, 0, len));

SecurityWrapper sw = null;
try {
    // Gets the PrivateKey class
    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    PrivateKey priKey = keyFactory.generatePrivate(new PKCS8EncodedKeySpec (bkey));

    // Get both certificates
    Certificate targetCert = new X509CertImpl (new FileInputStream ("/tmp/cert.pem"));
    Certificate caCert = new X509CertImpl (new FileInputStream ("/tmp/cacert.pem"));

    // Create the SecurityCertificate
    sw = new SecurityWrapper (priKey, new Certificate [] { targetCert, caCert});

} catch (Exception e) {
    e1.printStackTrace();
}
...
    // register service in a FADA node.
    FadaServiceID id = helper.register(
        new FadaLookupLocator (“www.fadanet.org:2002”).getRegistrar(),
        proxy,
        null,
        entries,
        10000L,
        null,
        codebase,
```

```
        sw); // The security wrapper
...
```

Similar code is executed by the client to read the certificate and later it is used in the lookup. In this case the certificate is explicitly a X509Certificate.

```
X509Certificate cert = null;
try {
    // Read the X509Certificate
    cert = new X509CertImpl (new FileInputStream ("/tmp/cacert.pem");
} catch (Exception e1) {
    e1.printStackTrace();
}
```

Annex A

Client and Service interface

Client

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples;

import java.io.IOException;
import java.security.InvalidKeyException;

import net.fada.FadaException;
import net.fada.directory.FadaLookupLocator;
import net.fada.toolkit.FadaHelper;

/**
 * Generic client to execute any proxy that implements net.fada.examples.RemoteUserList
 *
 * @author bob (javier.noguera@techideas.info)
 */
public class Client
{
    public static void main(String[] arg)
    {
        // check if we have enough args
        if (arg.length < 2) {
            help ();
            return;
        }

        // getting arguments
        String fadaAddress = arg[0];
        String action = arg[1];
        String parameter = null;

        // Try to get optional parámeters
        if (arg.length > 2) {
            parameter = arg[2];
        }

        // Prepare the lookup request parameters
        String[] interfaces = new String[] { "net.fada.examples.RemoteUserList" };

        // Perform the lookup procedure
        Object[] proxies;
        try {
            proxies = FadaHelper.lookup(
                    new FadaLookupLocator (fadaAddress).getRegistrar(), // where to find a node
                    null,               // entries to search for
                    null,               // FadaServiceId
                    interfaces,         // interfaces to search for
                    1,                  // Max results we want to search for
                    10000L,             // Search expiration times (in millis)
                    null                // Certificate (if needed)
                    );

            // Take one of the returned service proxies
            if (proxies.length == 0) {
                System.out.println (" Proxy do not found :(");
                return;
            }
```

```
            // Declare a variable to cast the service proxy onto
            net.fada.examples.RemoteUserList service = (net.fada.examples.RemoteUserList) proxies[0];

            // Write the command to be executed
            System.out.println(" Executing action:  " + action + " " + parameter);

            if ("add".equals (action)) {
                service.addUser (parameter);
            }
            else if ("remove".equals(action)) {
                service.removeUser(parameter);
            }
            else if ("list".equals(action)) {
                String [] list = service.getList();
                for (int i = 0; i < list.length; i++) {
                    System.out.println("\t" + list[i]);
                }
            }
            else if ("name".equals(action)) {
                System.out.println(service.getRoomName());
            }
        }
        catch (FadaException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (NullPointerException e) {
            e.printStackTrace();
        }
        catch (InvalidKeyException e) {
            e.printStackTrace();
        }
    }

    /**
     * Write help to the standard output
     *
     */
    private static void help ()
    {
        System.out.println("FADA client");
        System.out.println("Usage");
        System.out.println("java net.fada.examples.Client fada_url action [parameter]");
        System.out.println("\tfada_address: address and port to the fada node");
        System.out.println("\taction: action to do (add, remove, list or name)");
        System.out.println("\tparameter: parameter needed by the action (if applicable)");
        System.out.println("\nie. java net.fada.examples.j2ee.Client localhost:2002 add bob");
        System.out.println("    java net.fada.examples.j2ee.Client localhost2002 remove bob");
        System.out.println("    java net.fada.examples.j2ee.Client localhost2002 list");
        System.out.println("    java net.fada.examples.j2ee.Client localhost2002 name");
    }
}
```

RemoteUserList

```
/*
 * Created on 14-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples;


/**
 * Defines the methods that have to be implemented by the proxy to acces to the server
 * service.
```

```
 *
 * This Interface have to be into the client CLASSPATH. The proxy that implements this
 * interface will be downloaded from a FADA node by the client.
 *
 * @author bob (javier.noguera@techideas.info)
 */
public interface RemoteUserList
{
    /**
     * Registers one <code>name</code> into the server. If the username is already registered,
     * do nothing
     *
     * @param name username
     */
             public void addUser (String name);

    /**
     * Unregister one <code>name</code> into the server. If the username is not registered,
     * do nothing.
     *
     * @param name username
     */
             public void removeUser (String name);

    /**
     * Returns the <code>roomName</code> of the proxy. Nothing to do with the server.
     *
     * @param name username
     */
             public String getRoomName ();

    /**
     * Returns the registered user names array located in the server.
     *
     * @return usernames array
     */
             public String [] getList ();
}
```

Do-it-yourself

Userlist

```
/*
 * Created on 14-jun-2004
 *
 * This file is part of the FADA tutorial
 */
package net.fada.examples.uptoyou;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

/**
 * Implements a simple server with a simple communication protocol.
 *
 * This server allow external programs to add or remove users from an internal list.
 * It also prints a list of registered users to the standard output.
 *
 * All input messages begin with one character: (A)dd, (R)emove, (L)ist or (Q)uit and
 * their length has to be, as a maximun, 20 characters long.
 *
 * Bellow you can see the list of all messages that can be processed by the server:
 *  A username: add the username to the list if it does not exist yet.
 *  R username: remove the username form the list if it exists.
 *  L         : prints a list with registered usernames to standard output.
 *  Q         : quit.
 *
 * To run the server you only need  to run this class. If no parameter is passed the server will listen
 * on the default port (2727).
 *
 * @author bob
 */
public class UserList extends Thread
{
    private static final int DEFAULT_PORT = 2727;
    private int port;
    private Collection names = new ArrayList ();


             /**
              * Creates a server instance that will listen on <code>port</code>
              *
              * @param port listenig port
              */
             public UserList (int port)
             {
        this.port = port;
             }

    /**
     * @see java.lang.Thread#run()
     *
     */
    public void run ()
    {

        System.out.println ("UserList server started...");

        try {
            // Initialize ServerSocket
            ServerSocket ssocket = new ServerSocket (port);

            boolean end = false; // Do not end yet.
            byte [] command = new byte [20];

            while (!end) {
```

```
            // Wait for someone
            Socket client = ssocket.accept();

            // Read command
            int len = client.getInputStream().read(command);
            if (len < 0) {
                client.getOutputStream().close ();
                continue;
            }

            System.out.println("command: " + new String (command, 0, len));
            // Get the argument (if exists)
            String argument = new String (command, 1, len -1).trim();

            // Execute the apropiate action
            switch ((int) command[0]) {
                case 'A':
                    // Add the username to the Collection
                    // Only if name do not exists yet
                    if ((argument != null) && (!names.contains(argument))) {
                        names.add(argument);
                    }
                    break;

                case 'R':
                    // Remove the username form de Collection
                    names.remove(new String (command, 1, len -1).trim());
                    break;

                case 'L':
                    // List the registered user list
                    StringBuffer response = new StringBuffer ();

                    // Simbol '#' is the separator token
                    for (Iterator it = names.iterator(); it.hasNext();) {
                        response.append(it.next() + "#");
                    }

                    // Write the list and send results
                    System.out.println("USER LIST: " + response);
                    client.getOutputStream().write(response.toString().getBytes());
                    break;

                case 'Q':
                    // Stop the server
                    end = true;
                    break;
            }

            // Close client socket.
            client.getOutputStream().close ();
        }

    }
    catch (IOException e) {
        e.printStackTrace();
    }

    System.out.println ("UserList server finished!");
}


/**
 * Main method with parameters needed.
 * Parameters:
 *  - arg[0]: OPTIONAL. listening port (default 2727)
 *
 * @param arg arguments
 */
public static void main (String[] arg)
{
    // Allways print help
    help ();
```

```
        // Get port if param exists
        int port = DEFAULT_PORT;
        if (arg.length > 0) {
            try {
                port = Integer.parseInt(arg[0]);
            }
            catch (NumberFormatException e) { /* nop */ }
        }

        // Start the server
        UserList myRegister = new UserList (DEFAULT_PORT);
        myRegister.start();

        // Do nothing untill someone send 'Q' throw socket or CTRL+C key is pressed.
    }

    /**
     * Writes help to the standard output
     *
     */
    private static void help ()
    {
        System.out.println("Register a sevice into a FADA node (J2EE version)");
        System.out.println("Usage");
        System.out.println("java net.fada.examples.j2ee.Register [port]");
        System.out.println("\tport: listening port (default port is 2727)");
        System.out.println("\nie. java net.fada.examples.j2ee.Register 2727\n");
    }
}
```

UserListProxy

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 */
package net.fada.examples.uptoyou;


import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.StringTokenizer;

import net.fada.examples.RemoteUserList;

/**
 * This proxy will be registered into a FADA node and will enable the comunication
 * between the client an the server (UserList in this case).
 *
 * This proxy knows how to "speak" with the server, where it is and where it is listening.
 * This class implements the apropiate protocol to send commands and recive information
 * from the server. The client will download this proxy form a FADA node and automatically
 * can communicate with the server not knowing where the server is or what the communication
 * protocol is.
 *
 * In this case the <code>UserListProxy</code> class will communicate with the server via socket.
 *
 * @author bob
 */
public class UserListProxy implements RemoteUserList, java.io.Serializable
{
    private String roomName;
    private String ip;
    private int port;

    /**
     * Empty constructor necessary for serialization
     *
     */
    public UserListProxy ()
    {
```

```
    /* nop */
}

/**
 * Creates a new instance to be registered into a FADA node, specifying the
 * <code>address</code> and the <code>port</code> where the server will be waiting.
 * <code>roomName</code> is an internal identifier not used in the communication
 * process.
 *
 * @param roomName
 * @param address
 * @param port
 */
public UserListProxy (String roomName, String address, int port)
{
    this.roomName = roomName;
    this.ip = address;
    this.port = port;
}

/**
 * Communicates directly with the server and sends the appropiate command
 * to register the <code>name</code>
 *
 * @param name username
 */
public void addUser(String name)
{
    send ("A " + name);
}

/**
 * Communicates directly with the server and sends the appropiate command
 * to unregister the <code>name</code>
 *
 * @param name username
 */
public void removeUser(String name)
{
    send ("R " + name);
}

/**
 * Gets the <code>roomName</code>. This method does not need a communication
 * whith the server.
 *
 * @param name username
 */
public String getRoomName()
{
    return roomName;
}

/**
 * Communicates directly with the server and sends the appropiate command
 * to get the registered user list
 *
 * @return usernames array.
 */
public String[] getList()
{
    String[] response = null;
    int i = 0;

    StringTokenizer namelist = new StringTokenizer(send ("L"), "#", false);

    // If there are results
    if (namelist != null) {
        // Allocate array size for response and fill it
        response = new String [namelist.countTokens()];
        while (namelist.hasMoreTokens()) {
            response[i++] = namelist.nextToken();
        }
    }
```

```
        return response;
    }

    /**
     * Sends a command to the server via socket.
     *
     * @param text command to send
     * @return server response or null if there are no response.
     */
    private String  send (String text)
    {
        StringBuffer response = new StringBuffer ();

        try {
            // Conectamos y escribimos
            Socket client = new Socket (ip, port);
            client.getOutputStream().write(text.getBytes());

            // Esperamos respuesta si la hay
            int c = 0;
            while ((c = client.getInputStream().read()) > -1) {
                response.append((char) c);
            }

            client.close();
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        return response.toString();
    }

    public static void main (String [] arg)
    {
        UserListProxy p = new UserListProxy ("hola", "B", 3);

        p.getRoomName();
    }
}
```

Register

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 */
package net.fada.examples.uptoyou;

import java.io.IOException;
import java.net.Socket;
import java.security.InvalidKeyException;

import net.fada.FadaException;
import net.fada.directory.FadaLookupLocator;
import net.fada.directory.tool.FadaLeaseRenewer;
import net.fada.directory.tool.FadaServiceID;
import net.fada.toolkit.FadaHelper;


/**
 * Regiters a proxy (net.fada.examples.uptoyou.UserListProxy) into a FADA node using FadaHelper
 * classes.
 *
 * The proxy will be registered with the entries "userlist_yourself" and "make-it-yourself" that
 * and with the roomname "ROOM UserList make-it-yourself"
 *
 * @author bob
```

```
 */
public class Register
{
    /**
     * Main method with parameters needed.
     * Parameters:
     *  - arg[0]: fada node address "ip:port" (ie. "127.0.0.1:2002")
     *  - arg[1]: server addresss. The server have to be running (ie. "my.machine")
     *  - arg[2]: server listener port. Port in which the server is listening (ie. 2727)
     *  - arg[3]: OPTIONAL. Full url to the codebase (ie. "http://my.machine/example/classes.jar")
     *
     * if no arguments are supplied, help is send to the standard output.
     *
     * @param arg arguments
     */
    public static void main(String[] arg)
    {

        String roomName = "ROOM UserList make-it-yourself";
        String [] entries = new String [] {"userlist_yourself", "make-it-yourself"};

        // check if there are have enough args
        if (arg.length < 3) {
            help ();
            return;
        }

        // getting arguments
        String fadaAddress = arg[0];
        String serverAddress = arg[1];
        int serverPort = 0;
        try {
            serverPort = Integer.parseInt(arg[2]);
        }
        catch (NumberFormatException e) {
            System.out.println ("server_port have to be a number!!");
            System.out.println (arg[2] + "is not a number");
            help ();
            return;
        }
        String codebase = null;
        if (arg.length > 3) {
            codebase = arg[3];
        }

        // Create the proxy instance that will be registered into the
        UserListProxy proxy = new UserListProxy (roomName, serverAddress, serverPort);
        try {
            // Prepare the FadaHelper instance
            FadaHelper helper = new FadaHelper( new FadaLeaseRenewer() );
            // register service into a FADA node.
            FadaServiceID id = helper.register(
                    new FadaLookupLocator (fadaAddress).getRegistrar(), // where to find a node
                    proxy, // proxy (in this case created with fadagen)
                    null,               // FadaServiceId (if it was register before for example)
                    entries,            // entries that help the user to find the service
                    10000L,             // Renew lease period (in millis)
                    null,               // SecurityWrapper
                    codebase,               // codebase where cliend cand find libraries
                    null);              // What to do if FADA cannot renew the proxy

            System.out.println("proxy " + proxy.getClass() + " was registered into the FADA node");

            // We will wait till server is up
            int k = 0;
            while (true) {
                Socket s = new Socket (serverAddress, serverPort);
                //System.out.println(k++);
                s.close();
                try {
                    Thread.sleep(10000);
                }
                catch (InterruptedException e1) {
                    e1.printStackTrace();
```

```
                }
            }
        }
        catch (FadaException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        catch (NullPointerException e) {
            e.printStackTrace();
        }
        catch (InvalidKeyException e) {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * Write help to the standard output
     *
     */
    private static void help ()
    {
        System.out.println("Register a sevice into a FADA node (MAKE-IT-YOURSELF version)");
        System.out.println("Usage");
        System.out.println("java net.fada.examples.j2ee.Register fada_url server_address server_port [codebase]");
        System.out.println("\tfada_address: address and port to the fada node in the \"ip:port\" format");
        System.out.println("\tserver_address: name or ip to the server");
        System.out.println("\tserver_port: port to the server");
        System.out.println("\tcodebase: full url to the codebase");
                System.out.println("\nie.  java  net.fada.examples.j2ee.Register  127.0.0.1:2002  my.machine  2727
http://my.machine/example/classes.jar\n");
    }
}
```

Fadagen

UserRemoteListWithStub

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples.fadagen;

/**
 * Defines the methods to be implemented by fadagen library for communication.
 * This interface will be used by fadagen library to create stub and skeleton clases.
 *
 * There is one big diference between this class and <code>RemoteUserList<code> class.
 * This difference is that this interface defines the service (in the server side), not the
 * proxy. This is why <code>getRoomName</code> method is not defined, because roomName
 * is not a server variable (it is a proxy variable).
 *
 * @author bob (javier.noguera@techideas.info)
 */
public interface RemoteUserListWithStub extends net.fada.remote.Remote
{
    /**
     * Registers one <code>name</code> into the server. If the username is already registered,
     * do nothing
     *
     * @param name username
     */
    public void addUser (String name) throws net.fada.remote.RemoteException;

    /**
     * Unregister one <code>name</code> into the server. If the username is not registered,
     * do nothing.
     *
     * @param name username
     */
    public void removeUser (String name) throws net.fada.remote.RemoteException;

    /**
     * Returns the registered user names array located in the server.
     *
     * @return usernames array
     */
    public String [] getList () throws net.fada.remote.RemoteException;
}
```

UserList

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples.fadagen;

import net.fada.remote.RemoteObject;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import net.fada.remote.RemoteException;

/**
 * Implements the server interface.
 *
 * This class will run as server. It is needed to tun fadagen to generate a stub and
```

```
 * a skeleton for it.
 *
 * Listening port will be supplied just before trying to register it in the FADA node.
 *
 * @author bob (javier.noguera@techideas.info)
 */
public class UserList extends RemoteObject implements RemoteUserListWithStub
{
    private Collection names = new ArrayList ();

    /**
     * Adds <code>name</code> to the registered users list.
     *
     * @param name any name
     */
    public void addUser(String name) throws RemoteException
    {
        // Only if it does not exists
        if (!names.contains(name)) {
            names.add(name);
        }    }

    /**
     * Removes <code>name</code> from the registered user list.
     *
     * @param name any name
     */
    public void removeUser(String name) throws RemoteException
    {
        names.remove(name);
    }


    /**
     * Gets a registered user names array.
     *
     * @return usernames array
     */
    public String[] getList() throws RemoteException
    {
        // Allcate the array size
        String [] result = new String [names.size()];
        int i = 0;

        // Get all names in the Collection
        for (Iterator it = names.iterator(); it.hasNext();) {
            result [i++] = (String) it.next();
        }

        return result;
    }
}
```

UserListProxy

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples.fadagen;

import net.fada.remote.RemoteException;

/**
 * This proxy will be registered into a FADA node and will enable the comunication
 * between the client an the server (UserList in this case).
 *
 * This proxy knows how to "speak" with the server, where it is and where it is listening.
 * This class implements the apropiate protocol to send commands and recive information
 * from the server. The client will download this proxy form a FADA node and automatically
```

```
 * can communicate with the server not knowing where the server is or what the communication
 * protocol is.
 *
 * In this case the <code>UserListProxy</code> class will communicate with the server via socket.
 *
 * @author bob (javier.noguera@techideas.info)
 */
public class UserListProxy implements net.fada.examples.RemoteUserList, java.io.Serializable
{
    private RemoteUserListWithStub stub;
    private String roomName;

    /**
     * Empty constructor necessary for serialization
     *
     */
    public UserListProxy ()
    {
        /* nop */
    }

    /**
     * Creates a new instance to be registered into a FADA node, specifying the
     * <code>address</code> and the <code>port</code> where the server will be waiting.
     * <code>roomName</code> is an internal identifier not used in the communication
     * process.
     *
     * @param roomName
     * @param address
     * @param port
     */
    public UserListProxy (RemoteUserListWithStub stub, String roomName)
    {
        this.stub = stub;
        this.roomName = roomName;
    }

    /**
     * Communicates directly with the server and sends the appropiate command
     * to register the <code>name</code>
     *
     * @param name username
     */
    public void addUser(String name)
    {
        try {
            stub.addUser(name);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    /**
     * Communicates directly with the server and sends the appropiate command
     * to unregister the <code>name</code>
     *
     * @param name username
     */
    public void removeUser(String name)
    {
        try {
            stub.removeUser (name);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    /**
     * Gets the <code>roomName</code>. This method does not need a communication
     * whith the server.
     *
     * @param name username
     */
    public String getRoomName()
```

```
    {
        return roomName;
    }

    /**
     * Communicates directly with the server and sends the appropiate command
     * to get the registered user list
     *
     * @return usernames array.
     */
    public String[] getList()
    {
        String[] response = null;
        try {
            response = stub.getList();
        } catch (RemoteException e) {
            e.printStackTrace();
        }

        return response;
    }
}
```

RegisterFadagen

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples.fadagen;

import java.io.IOException;
import java.security.InvalidKeyException;

import net.fada.FadaException;
import net.fada.directory.FadaLookupLocator;
import net.fada.directory.tool.FadaLeaseRenewer;
import net.fada.directory.tool.FadaServiceID;
import net.fada.toolkit.FadaHelper;
import net.fada.transport.ClientTransport;
import net.fada.transport.ClientTransportImpl;
import net.fada.transport.ServerTransport;
import net.fada.transport.ServerTransportImpl;


/**
 * Register a service into a FADA node.
 *
 * This code is part of an example in which:
 *  - The server is a stand-alone class converted into server by <code>fadagen libraries</code>
 *  - The communication with the servlet is done with <i>fadagen</i>
 *
 * @author bob (javier.noguera@techideas.info)
 */
public class RegisterFadagen
{
    /**
     * Main method with parameters needed.
     * Parameters:
     *  - arg[0]: fada node address "ip:port" (ie. "127.0.0.1:2002")
     *  - arg[1]: full url to the servlet will be listening (ie. "http://my.machine/example/j2ee")
     *  - arg[2]: OPTIONAL. Full url to the codebase (ie. "http://my.machine/example/classes.jar")
     *  - arg[3]: OPTIONAL. Endpoint of the server. Default value is /fadagen
     *  - arg[4]: OPTIONAL. Server port. Default value is 2727
     *
     * if no arguments are supplied, help is send to the standard output.
     *
     * @param arg arguments
     */
    public static void main(String[] arg)
    {
```

```
String roomName = "ROOM UserList fadagen";
String [] entries = new String [] {"userlist_fadagen", "fadagen_userlist"};
String endpoint = "/fadagen/register";
int port = 2729;

// check if we have enough args
if (arg.length < 2) {
    help ();
    return;
}

// getting arguments
String fadaAddress = arg[0];
String serverIp = arg[1];
String codebase = null;

// Try to get optional parámeters
if (arg.length > 2) {
    codebase = arg[2];
}
if (arg.length > 3) {
    endpoint = arg[3];
}
if (arg.length > 4) {
    try {
        port = Integer.parseInt(arg[4]);
    }
    catch (NumberFormatException e) { /* nop */}
}

try {
    // Ejecutamos el servidor
    UserList myRegister = new UserList ();

    // Transporte
    ServerTransport st = new ServerTransportImpl (port, endpoint);
    ClientTransport ct = new ClientTransportImpl ("http://" + serverIp + ":" + port + endpoint);

    // Create the stub and the proxy
    RemoteUserListWithStub stub = (RemoteUserListWithStub) myRegister.export (st, ct);
    UserListProxy proxy = new UserListProxy (stub, roomName);

    // Prepare the FadaHelper instance
    FadaHelper helper = new FadaHelper( new FadaLeaseRenewer() );

    // register service into a FADA node.
    FadaServiceID id = helper.register(
            new FadaLookupLocator (fadaAddress).getRegistrar(), // where to find a node
            proxy, // proxy (in this case created with fadagen)
            null,               // FadaServiceId (if it was register before for example)
            entries,            // entries that help the user to find the service
            10000L,             // Renew lease period (in millis)
            null,               // SecurityWrapper
            codebase,           // codebase where cliend cand find libraries
            null);              // What to do if FADA cannot renew the proxy

    System.out.println("proxy " + stub.getClass() + " was registered into the FADA node");
}
catch (FadaException e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
catch (NullPointerException e) {
    e.printStackTrace();
}
catch (ClassNotFoundException e) {
    e.printStackTrace();
}
catch (InstantiationException e) {
    e.printStackTrace();
}
catch (IllegalAccessException e) {
```

```
                e.printStackTrace();
            }
            catch (InvalidKeyException e) {
                e.printStackTrace();
            }
        }

        /**
         * Write help to the standard output
         *
         */
        private static void help ()
        {
            System.out.println("Register a sevice into a FADA node (J2EE version)");
            System.out.println("Usage");
             System.out.println("java net.fada.examples.j2ee.Register fada_url server_ip [codebase] [endpoint [port]]
");
            System.out.println("\tfada_address: address and port to the fada node");
              System.out.println("\tserver_address: address of the server to be found by others computers (without
port)");
            System.out.println("\tcodebase: full url to the codebase");
            System.out.println("\tendpoint: default endpoint is /fadagen");
            System.out.println("\tport: default port is 2727");
                        System.out.println("\nie.   net.fada.examples.j2ee.Register   127.0.0.1:2002   81.91.102.203
http://my.machine/example/classes.jar\n");
        }
    }
```

j2ee

UserRemoteListWithStub

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples.j2ee;

/**
 * Defines the methods to be implemented by fadagen library for communication.
 * This interface will be used by fadagen library to create stub and skeleton clases.
 *
 * @author bob (javier.noguera@techideas.info)
 */
public interface RemoteUserListWithStub extends net.fada.remote.Remote
{
    /**
     * Registers one <code>name</code> into the server. If the username is already registered,
     * do nothing
     *
     * @param name username
     */
    public void addUser (String name) throws net.fada.remote.RemoteException;

    /**
     * Unregister one <code>name</code> into the server. If the username is not registered,
     * do nothing.
     *
     * @param name username
     */
    public void removeUser (String name) throws net.fada.remote.RemoteException;

    /**
     * Returns the registered user names array located in the server.
     *
     * @return usernames array
     */
    public String [] getList () throws net.fada.remote.RemoteException;
}
```

UserList

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples.j2ee;

import java.io.IOException;
import java.io.PrintWriter;
import java.security.InvalidKeyException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import net.fada.FadaException;
import net.fada.directory.FadaLookupLocator;
import net.fada.directory.tool.FadaLeaseRenewer;
import net.fada.directory.tool.FadaServiceID;
import net.fada.remote.RemoteException;
import net.fada.toolkit.FadaHelper;
import net.fada.transport.ClientTransportImpl;
import net.fada.transport.FadaHTTPServlet;

/**
 * Servlet used as a service. We can take advantage of the Application Server
 * architecture for placing our service.
 * It is also useful if we want to use the servlet interface to configure
 * servlet parameters in runtime or to analyze server progress.
 *
 * <code>init</code> method is used to register the service.
 *
 * @author bob (javier.noguera@techideas.info)
 */
public class UserList extends FadaHTTPServlet implements RemoteUserListWithStub
{
    private String roomName;
    private Collection names = new ArrayList ();

    /**
     * Registers the service in the FADA node. Some init parameters are
     * necessary:
     *  - room_name
     *  - fada_address
     *  - end_point
     *  - codebase
     * @see javax.servlet.Servlet#init(javax.servlet.ServletConfig)
     */
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);

        // Inicializamos el servicio en FADA
        String roomName = config.getInitParameter("room_name");
        String [] entries = new String [] {"userlist_j2ee", "userlist_servlet"};

        // getting arguments
        String fadaAddress = config.getInitParameter("fada_address");
        String servletUrl = config.getInitParameter("end_point");
        String codebase = config.getInitParameter("codebase"); //http://192.168.0.121/registerFada.jar";

        try {
            ClientTransportImpl trans = new ClientTransportImpl(servletUrl);
```

```
            // Create the stub and the proxy
            RemoteUserListWithStub stub = (RemoteUserListWithStub) this.export (this, trans);
            UserListProxy proxy = new UserListProxy (stub, roomName);

            // Prepare the FadaHelper instance
            FadaHelper helper = new FadaHelper( new FadaLeaseRenewer() );

            // register service into a FADA node.
            FadaServiceID id = helper.register(
                    new FadaLookupLocator (fadaAddress).getRegistrar(), // where to find a node
                    proxy,               // proxy (in this case created with fadagen)
                    null,                // FadaServiceId (if it was register before for example)
                    entries,             // entries that help the user to find the service
                    10000L,              // Renew lease period (in millis)
                    null,                // SecurityWrapper
                    codebase,            // codebase where cliend cand find libraries
                    null);               // What to do if FADA cannot renew the proxy

            System.out.println("proxy " + stub.getClass() + " was registered into the FADA node");
        }
        catch (FadaException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        catch (NullPointerException e) {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (InstantiationException e) {
            e.printStackTrace();
        }
        catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        catch (InvalidKeyException e) {
            e.printStackTrace();
        }
    }


    /**
                         *     @see     javax.servlet.http.HttpServlet#doGet(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
     */
       public  void  doGet(HttpServletRequest  request,  HttpServletResponse  response)  throws  ServletException,
IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Header
        out.println ("<html>");
        out.println ("Userlist:<br><hr>");

        // Print all names
        for (Iterator it = names.iterator(); it.hasNext();) {
            out.println (" - " + it.next() + "<br>");
        }

        // Footer
        out.println("<hr>");
        out.println ("</html>");
    }


    /**
     * @see fadagen.RemoteUserListWithStub#register(java.lang.String)
     */
    public void addUser(String name) throws RemoteException
```

```
    {
        // Si NO existe, lo guardamos
        if (!names.contains(name)) {
            names.add(name);
        }
    }

    /**
     * @see fadagen.RemoteUserListWithStub#unregister(java.lang.String)
     */
    public void removeUser(String name) throws RemoteException
    {
        names.remove(name);
    }

    /**
     * @see fadagen.RemoteUserListWithStub#getList()
     */
    public String[] getList() throws RemoteException
    {
        // Creamos el array de salida
        String [] result = new String [names.size()];
        int i = 0;

        // Recorremos todos los nombres
        for (Iterator it = names.iterator(); it.hasNext();) {
            result [i++] = (String) it.next();
        }

        return result;
    }

}
```

UserListProxy

```
/*
 * Created on 15-jun-2004
 *
 * This file is part of the FADA tutorial
 * www.fadanet.org
 */
package net.fada.examples.j2ee;

import net.fada.remote.RemoteException;

/**
 * This proxy will be registered into a FADA node and will enable the comunication
 * between the client an the server (UserList in this case).
 *
 * This proxy knows how to "speak" with the server, where it is and where it is listening.
 * This class implements the apropiate protocol to send commands and recive information
 * from the server. The client will download this proxy form a FADA node and automatically
 * can communicate with the server not knowing where the server is or what the communication
 * protocol is.
 *
 * In this case the <code>UserListProxy</code> class will communicate with the server via socket.
 *
 * @author bob (javier.noguera@techideas.info)
 */
public class UserListProxy implements net.fada.examples.RemoteUserList, java.io.Serializable
{
    private RemoteUserListWithStub stub;
    private String roomName;

    /**
     * Empty constructor necessary for serialization
     *
     */
    public UserListProxy ()
    {
        /* nop */
```

```
}

/**
 * Creates a new instance to be registered into a FADA node, specifying the
 * <code>address</code> and the <code>port</code> where the server will be waiting.
 * <code>roomName</code> is an internal identifier not used in the communication
 * process.
 *
 * @param roomName
 * @param address
 * @param port
 */
public UserListProxy (RemoteUserListWithStub stub, String roomName)
{
    this.stub = stub;
    this.roomName = roomName;
}

/**
 * Communicates directly with the server and sends the appropiate command
 * to register the <code>name</code>
 *
 * @param name username
 */
public void addUser(String name)
{
    try {
        stub.addUser(name);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}

/**
 * Communicates directly with the server and sends the appropiate command
 * to unregister the <code>name</code>
 *
 * @param name username
 */
public void removeUser(String name)
{
    try {
        stub.removeUser (name);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}

/**
 * Gets the <code>roomName</code>. This method does not need a communication
 * whith the server.
 *
 * @param name username
 */
public String getRoomName()
{
    return roomName;
}

/**
 * Communicates directly with the server and sends the appropiate command
 * to get the registered user list
 *
 * @return usernames array.
 */
public String[] getList()
{
    String[] response = null;
    try {
        response = stub.getList();
    } catch (RemoteException e) {
        e.printStackTrace();
    }
```

```
            return response;
        }
    }
```