XML-RPC INVOCATION OF A METHOD ON A REMOTE OBJECT

The default communication protocol for licas itself is XML-RPC, or XML Remote Procedure Calling. This is similar to Web Services, for example. Communication is carried out by invoking a method on some object using Reflection. This object can be local or remote and there are some main classes that are used to build the method call. These include `CallObject`, `MethodInfo`, `MethodFactory` and `MethodHandler`, all in the 'licas.call' package. These classes are used as follows:

CALLOBJECT CLASS

This class is used to make the method call. This class checks if the call is to a local or a remote object. If it is a local object, then the object itself is retrieved and the method is invoked using the Java Reflection package. If the call is remote, that is without a direct reference, then a remote communication object is created and used to make the method call. This would be an `RPC_MethodHandler` for licas internally, but could be a `Rest_MethodHandler` or a `Soap_MethodHandler` object just as easily, for any other call. This is determined by the MethodInfo `getCallType` reply.

METHODINFO CLASS

The MethodInfo class is used to describe what method you want to call. There are a number of variable values that need to be set to use this and there are two different ways in which it can be constructed. The easiest way to create the method information is to use the `MethodFactory.createMethodCall` method. This would look like:

methodInfo = MethodFactory.createMethodCall(methodName, returnType, serviceObj, parameters, passwordHandler);

In this case:

- 'methodName' is the name of the method that you want to call, for example 'toString'.
- 'returnType' is the fully qualified classtype of the return parameter, for example 'java.lang.String'.
- 'serviceObj' is the reference to the object that you want to call. This could be a direct reference to the object itself or an XML description of the URI for the service on some remote (or local) server. These example files show how this URI path is constructed.
- 'parameters' are a list of parameters that are required for the method interface description. This is a list of parameters stored in a 'Vector'. They are then added to 'ParamInfo' objects by the 'MethodInfo' object during the construction process.
- 'passwordHandler' is the password handler that stores the service passwords. The 'MethodInfo' object will automatically try to retrieve the 'server' and the 'service' passwords for the service that is being invoked on the server.

If you use this format, the following is automatically done for you:

- The service URI is cast from the serviceObj passed in. If the object is a direct reference, then there is no URI and the method is invoked on the object itself.
- The server and service passwords are automatically retrieved.
- The parameter list is automatically stored in 'ParamInfo' objects.

You can also construct this manually, which is also more reliable, by using the following sort of procedure:

```
//this example adds a new service and the parameters are for that specifically
//see the other example files for further details
methodInfo = new MethodInfo();
//set service method name
methodInfo.setName(MethodConst.ADDSERVICE);
//set the return type
methodInfo.setRtnType(TypeConst.BOOLEAN);
//set the service uri
methodInfo.setServiceURI(serviceUri);
//set the server password
methodInfo.setServerPassword(passwordHandler.getServicePassword(serverUri));
//set the service password
methodInfo.setPassword(passwordHandler.getServicePassword(theUuid));
//add the parameter list, first is the service password again
methodInfo.addParam(passwordHandler.getServicePassword(theUuid));
//second is the new service uuid
methodInfo.addParam(theUuid);
//third is the new service type
methodInfo.addParam(serviceType);
//additional jar files might need to be loaded for new classes, but
typically this list is empty
methodInfo.addParam(jarFiles);
//classtype of the new service to create and load
methodInfo.addParam(serviceClass);
//true if automatically start the new service's thread
methodInfo.addParam(false);
//constructor parameters for the new service to load
methodInfo.addParam(params);
```

The MethodInfo object also contains some other parameters that must be set manually if they are used. They are:

setCallTimeout(...) this can be used to change the time the client service will wait for a reply. The time is 10 seconds by default, but you can change it here by specifying a different time in milliseconds.

setPacketSize(...) this allows you to make a call in stages and pass only part of the message, up to the specified size, each time. There is an additional overhead for the wrapping MethodInfo

object itself, but that is constant, where the large size would come from a reply parameter. This however allows you to send a very large message in a smaller number of stages.

setCommunicationID(...) this allows you to tag the message call with a unique id that can be used for agent-like communication.

also setClientUri(...), if it is required and not passed to the constructor.

After you have constructed the method info, you can make the method call as follows:
CallObject callObject = new CallObject();
Object reply = (cast to type)callObject.call(methodInfo);

The reply is always an Object and so a primitive type might need to be cast from a similar Object type.

PARSERS

The XML-RPC mechanism parses or serializes the message into an XML and then String-based format, using parsers that you need to provide yourself. Most of the basic object and lists are covered now, but any of your own complex classes might need a new parser. Typically however, you make remote calls using known data objects that are already covered, but it is relatively easy to add a new one.

For the parser to be recognised, it needs to implement the 'ParserDef' interface from the text package. The parser is then added to the system by making the method call:

Parsers.addParser('classtype of object to be parsed', 'parser object');

`Parsers` is the main parsing class and can be found in 'org.licas.parsers'. So long as your parser exists and can be found, this class will automatically use it to parse your new classes when needed. For example, if the calling mechanism finds the object Obj_1 as a parameter and needs to parse it, it looks for a parser that is saved under the Obj_1 class type name. If one exists, then it is retrieved and used to serialize the object into XML. The server side then also needs the same parser to parse the serialized XML format back into the object again. There is also the possibility of serializing your object in a standard Java Serialization way, by implementing the `Serializable` interface on your object, which will also work automatically as part of the calling mechanism. See the user guide for more details on this.

SYNCHRONOUS OR ASYNCHRONOUS CALLING

If your method call has a return type, the client service is made to wait for a reply. If your method call does not have a return type (the return type is `TypeConst.VOID`), then it is treated as an asynchronous call and the calling service can be freed from the call before it is completed.