

Design

The design we chose for this lab is very similar to Java's original RMI design.

At least three actors are needed: a client, a server, and a registry.

To begin, the registry listens for incoming LOOKUP, LIST, BIND, and REBIND requests, all of which are wrapped in the RegistryRequestMessage object.

The requests are processed as follows:

- i The LIST command returns a list of all the registered objects that exist with that registry.
- ii The LOOKUP command returns the remoteObjectReference of the requested object.
- iii The BIND command binds the given remote object reference to the specified unused ID in the registry, allowing for future lookups.
- iv The REBIND command binds the given remote object reference to the specified ID, overwriting a previous reference if it exists.

All messages sent to the registry are encapsulated in request-message objects.

Similarly, all messages sent from the registry are encapsulated in return-message objects.

The server(s) registers its remote objects by going through a registry messenger, which tells the registry to register a remote object with the specified ID and stub (interface). Server methods invoked remotely can return other remote object references by either already having the remote reference in the class containing said method, or can look up the appropriate reference in the registry.

The server(s) uses a helper to listen to RMI messages and invoke the methods on the appropriate local (server) objects.

The client(s) does not query the registry directly either, but rather goes through the same messenger used by the server. Through the messenger, the client queries the registry for a list of remote object references available, and then requests one or more of them. The remote object reference returned has a method to return its stub, from which we can finally make remote method invocations.

The stubs construct an invocation message to send off to the server helper, which is actually the execution skeleton / dispatcher. This execution skeleton invokes the method and packages the return value (or exception) into a message back to the stub. The stub then finally extracts the return value from the message and returns this value if it is not an exception; if the value is an exception, it is thrown.

Major Design Decisions

Our framework has both clients and servers using the same message object, the `RegistryRequestMessage`, to communicate with the Registry. This allows for less distinction between the roles of a server and of a client. For example, two machines could both start as "servers" and register their own remote objects, but later these machines become "clients" by invoking the remote methods of the other machine.

For similar reasons, we encapsulate all messages from the Registry within a `ReturnMessage` object, which communicates whether a request to the registry was properly fulfilled or not. This message can contain either the return value, or an exception if one occurred.

In order to more clearly separate receiving RMI requests from actually handling them, the server helper (which represents the skeleton) has two components:

- 1) The helper itself only listens for incoming RMI requests, and spawns a new thread for each request.
- 2) The execution of the request occurs in an `ExecutionAgent`, which is also responsible for communicating the results back to the client. Each agent runs on its own thread, which was created by the helper.

The stubs are implemented as Proxy objects with our own custom `RemoteInvocationHandler`. This handler marshalls all method invocations into a bean called a `RemoteInvocationMessage`, which is then sent to the server helper / execution skeleton.

How to Run

Our example involves querying "databases" for the capitals of nations and states. These databases are stored in an archive, from the client requests specific databases.

The archive and all of its databases have remote object references on the registry.

The references to the databases can either be retrieved from the registry directly, or by a method call on the archive, which returns the appropriate remote object reference.

- 1) The first step towards getting our project up and running is to make some changes to the `properties.txt` file located in the project root.

Here, the parameters of interest are:

- i `serverPort`: If starting a server instance, this port will be used
- ii `clientPort`: If starting a client instance, this port will be used
- iii `registryPort`: If starting a registry instance, this port will be used
- iv `registryIPAddress`: Set to IP of the machine that the registry will run on.
Used by clients and servers to connect to registry, ensure that this is correct!

- 2) The next step is to run one or more of the following scripts:
 - i package.sh: compiles all java files and creates a jar
 - ii execute.sh <Registry / Server / Client>: starts the specified instance
 - iii clean.sh: deletes all generated class files and the jar file
 - iv run.sh <Registry / Server / Client>:
same as running: package.sh, followed by execute.sh <Registry / Server / Client>
- 3) For our example, please start the instances in the following order:
 - (a) Registry
 - (b) Server
 - (c) Client
- 4) The output should demonstrate our framework's capability to:
 - i Register (bind) remote object references on the registry
 - ii List remote object references currently on the registry
 - iii Lookup remote object references from registry
 - iv Get local stubs for remote object references
 - v Invoke method on stub, which can return, a value, an exception, or another remote object reference.
 - vi Propagate exceptions thrown from remote procedure call

Assumptions

1. The following file(s) are located at the execution directory of EVERY instance:
 - (a) properties.txt
2. The following file(s) are located at the execution directory of each SERVER instance:
 - (a) nations.txt
 - (b) states.txt
3. The ports specified in the configuration file are not being used by other processes.