Bilkent University

Computer Engineering Department

CS442: Distributed Systems

Project Report

RPC-Middleware System

Group 4

Group Members

Enes Akdoğan - 21200514

Mesut Gürlek - 21201157

Mehmet Çağrı Kaymak - 21202177

21.05.2017

System Design of RPC-Middleware Implementation

1. Registry

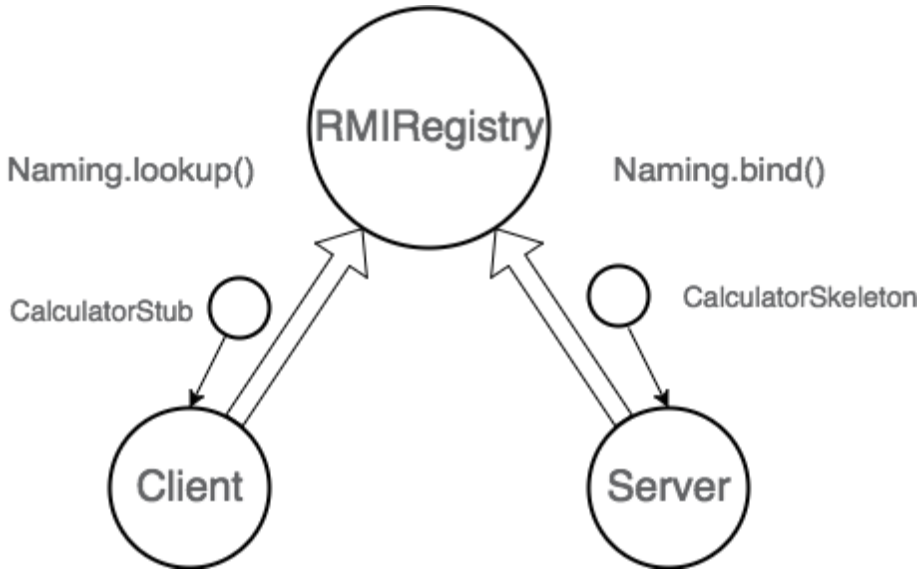

Figure 1: Simple visualization of registry system

For the registry system, a hashmap is used to store each entry. This systems maps remote object names to remote object references. When an object is binded to the registry, a remote object references is created with using port, address, name and class name of the object which is binded to the registry. After the binding is done, clients who want to use the binded objects could access the remote objects with using RMI system with simple lookup. So, with this system clients could access remote objects by just knowing the name of the entry and address of the registry.

Even it is not implemented in our current system, this registry table could be expanded to support multiple objects for the same name. So, when one of the objects fail, replicated objects could answer the incoming request. Registry system makes these replicas transparent to the user.

Clients and servers cannot reach registry directly. A static method is implemented to create a gateway between registry and other nodes. When a node wants to make lookup or bind, it uses Naming class. This class creates a specific

registry message to send to the registry server. So, naming decouples nodes from registry. This class is also responsible for creating stub and skeleton instances. Client will get stub and server will get a skeleton object. They will use these objects to communicate. This subject will be discussed in further details in the following sections.

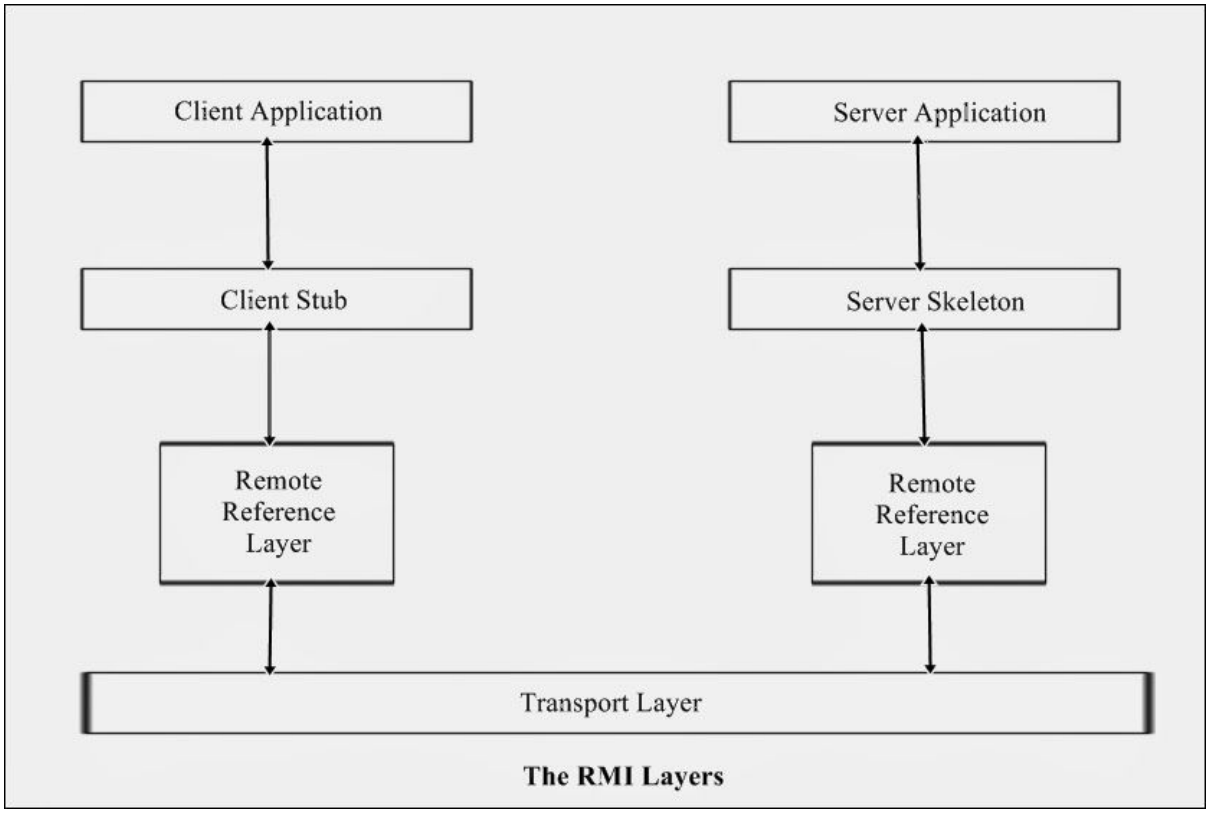

Figure 2: Overall design of RMI system

2. Server

In RPC system, server side has all the actual objects. It directs incoming requests to the responsible objects with using different components. The main components are server communication module, dispatcher, skeleton and an object reference hash table which is very similar to the registry table.

2.1. Server Communication Module

This component is responsible for the receiving messages from the clients and sending them to the responses generated by the actual objects. It uses two

different classes to handle this task. For each of skeletons a ServerCommunicationModule thread is created. This listens the port of a specific object which of the skeleton is related with. For the every incoming messages, it creates an instance of ServerCommunicationHandler thread. This thread receives the message from a specific client and it sends this message to the dispatcher and it waits a response so that it can send it back to the client.

2.2. Dispatcher

This component is responsible for the finding correct method of the skeleton. It checks the method name field of the message which is coming from the communication module. After that, it loops through all methods of the clients to find the correct method. It invokes that specific method of the skeleton with the arguments indicated in the message. Then, it waits a response from skeleton so that it can return this message to the communication module.

2.3. Skeleton

This component is responsible for unmarshalling the message arguments and calling correct method of actual object with this unmarshalled arguments. After that method is completed, skeleton takes the returned response and unmarshall it to send back to communication layer via dispatcher.

3. Client

Clients lookup the objects they want to use with using naming service. Naming class returns a stub instance which mimics the actual object. Thanks to this stub, client doesn't care the actual implementation of the RMI middleware. It uses this stub as if it is the real object. Communication with the server is completely transparent to the client. To handle client side Client Communication Module and Stubs are used.

3.1. Stub

After client calls lookup method of the Naming service, this service returns a stub for the actual object. Since this stub has the same methods with the actual

object, it can easily mimic the actual object. However, the underlying structure of each method is completely different. Even they share same interface with actual object, a stub creates a communication module instance and a message which includes arguments, method name and remote object reference. Then, this method just sends this message to the server communication module then wait for the response. After the response arrives, it returns the value inside the response message. Thanks to this logic, client couldn't understand the difference between stub and actual object. These stubs are generated with using stub generator and this generator will be discussed in the following “Stub and Skeleton Generator” section.

3.2. Client Communication Module

This module is almost identical to Server Communication Module. The only difference is that while server communication module uses multi threaded approach to handle incoming requests, this module just uses single thread approach since it is responsible for just one client. This module is used by stubs which are generated by stub and skeleton generator.

4. Stub and Skeleton Generator

This component is responsible for generation Stubs for Clients and Skeleton for Server automatically. Component only takes the Object Interfaces then generates both stub and skeleton parts using Java Reflections. Since, Java Reflections can provide method definitions and their parameters from the instance of an interface, we reach these methods, class definition and private, public variables the generate stub and skeletons. After taking modifiers we generate other parts as hard coded to simplify generation.

In stub generation, We first put library imports, class definition, variables and constructor codes in order. We get class methods from interface instance and create a message by adding method parameters into a vector. After that generates a message that includes method name, parameters and remote reference object. Finally we send message through Client Communication Module. Generated Stub method code as follows:

```
Vector<Object> vec = new Vector<Object>();
```

```
vec.add(param0);
vec.add(param1);
Message message = new Message();
message.setMethodName("match");
message.setArguments(vec);
message.setRemoteObjectReference(ror);
comm.sendMessage(message);
```

In Skeleton generation, we apply the same process until method generations. In method generation, we take the parameters from incoming message. Then calls the corresponding method onto remote object by passing the parameters.Then returned value put into the message's vector attribute and resend the message again. Generated code as the following:

```
Vector vec = message.getArguments();
java.lang.String returnValue;
returnValue = remoteObject.match((java.lang.String) vec.get(0), (Integer) vec.get(1));
Vector<Object> returnVector = new Vector<Object>();        // vector of args to pass back
returnVector.add(returnValue);
message.setArguments(returnVector);
return message;
```

5. Message Types

There are two types of messages in the system: Registry Message and Message.

5.1. Registry Message

Registry Message is the message send and received between Registry and Naming modules. It has following attributes: messageType, name, object and port.

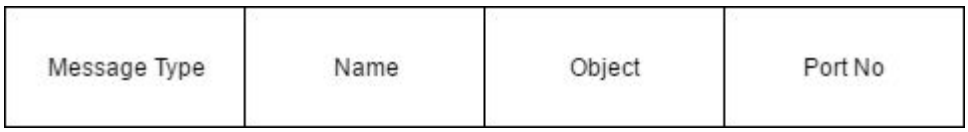

Figure 3: Registry Message Format

5.2. Message

Message is the main message type to communicate through client to server by sockets. Methods and its parameters are converted into message and send from

Client Stub to Server through communication modules. It has the following attributes: messageType, remoteObjectReference, methodName and arguments.

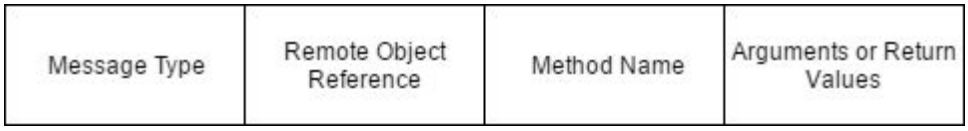

Figure 4: Message Format

6. Calculator Example with Using RMI Middleware

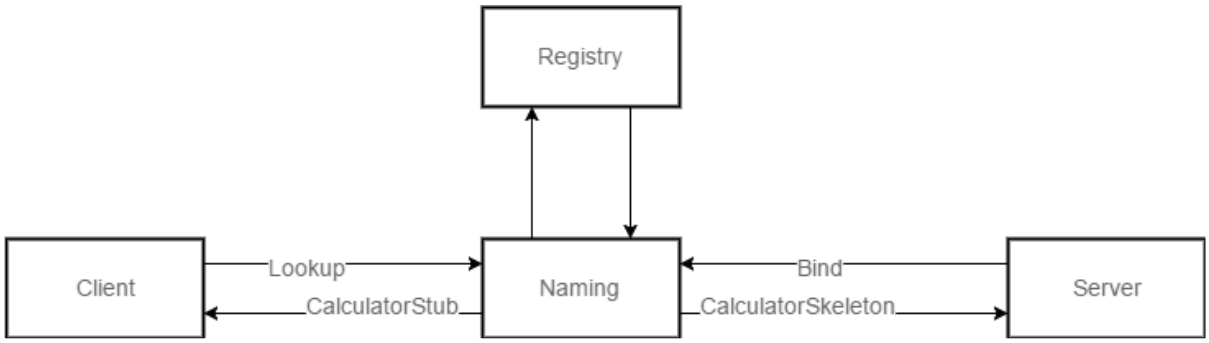

Figure 5: Registry phase

At this phase, server binds a calculator object to the registry with using Naming service and it receives an instance of Calculator Skeleton. Then, Client lookup the registry and it receives Calculator Stub.

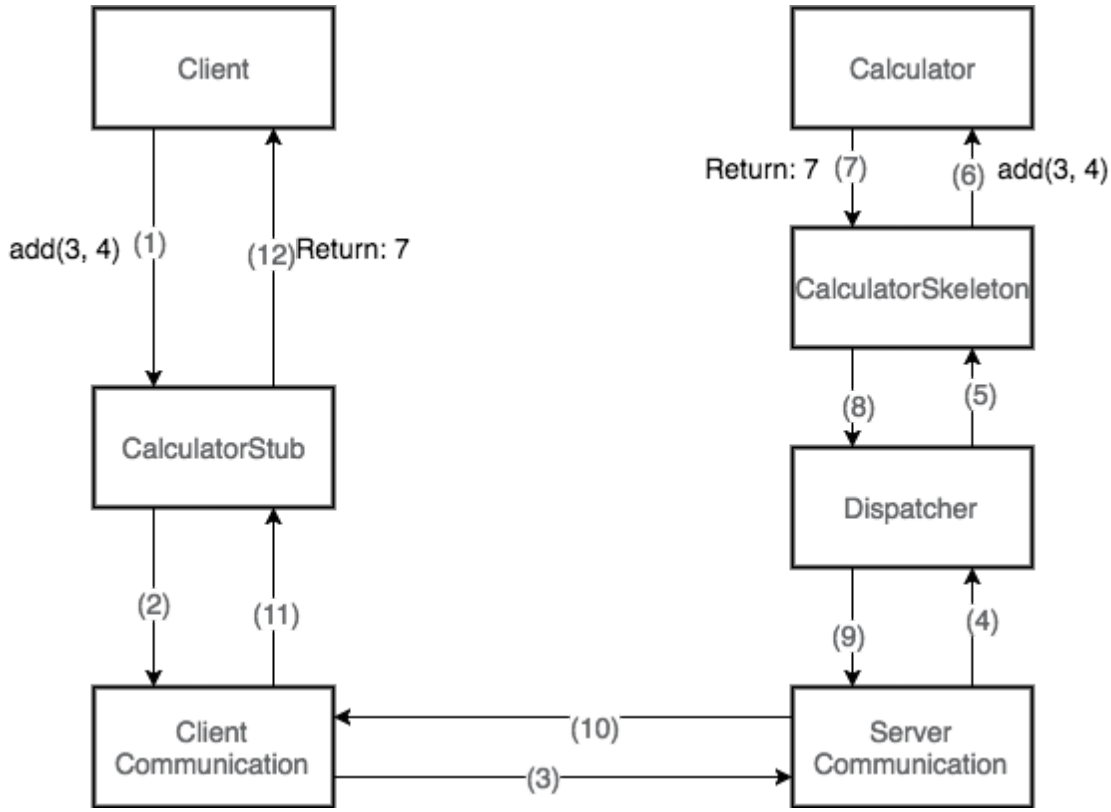

Figure 6: Flow Diagram of addition call via Calculator stub

At this phase, client calls add method of Calculator Stub. Then, even the client doesn't realize the underlying structure, a message is created at the stub and sent to Server Communication Module with using Client Communication Module. Then, server module propagates this message to the dispatcher. It checks the method name field and sees it is "add" method. Then, it calls the add method of the related skeleton with using this message as a parameter. Calculator Skeleton unmarshalls the arguments (3 and 4), calls the add method of actual Calculator object. Then, it calculates the result which is 7, and sends the return value back to the client with using same path.