CMPSCI 677 Operating Systems **Spring 2022**

Lecture 4: February 7

Lecturer: Prashant Shenoy *Scribe: Ge Shi (2019), Shubham Shetty (2022)*

4.1 Overview

In the previous lecture, we learned about remote procedure call, socket programming, and communication abstractions for distributed systems. This lecture will cover the following topics:

- Alternate RPC Models
- Remote Method Invocations (RMI)
- RMI & RPC Implementations and Examples

4.2 Alternate RPC Models

4.2.1 Lightweight RPC (LRPC)

Many RPCs occur between client and server on same machine. Lightweight RPCs are the special case of RPCs which are optimised to handle cases where calling process and the called process are on the same machine.

Remember the two forms of communication of a distributed system – explicit (passing data) and implicit (sharing memory). You can think of not using an RPC system for the special case that both processes are on the same machine but using a shared piece memory. The optimization is to construct the message as a buffer and simply write to the shared memory region. This avoids the TCP/IP overheads associated with normal RPC calls.

When client and server both are two processes on the same machine and you make RPC calls between two components on the same machine, following are the things which can make it better over the traditional RPC:

1. No need for the marshalling here.
2. We can get rid of explicit message passing completely. Rather shared memory is used as a way of communication.
3. Stub can use the run-time flag can be used to decide whether to use TCP/IP (Normal RPC) or shared memory (LRPC).
4. No XDR is required.

Steps of execution of LRPC:

1. Arguments of the calling process are pushed on the stack,

2. Trap to kernel is send,

3. After sending trap to kernel, it either constructs an explicit shared memory region and put the arguments there or take the page from stack and simply turn it into shared page,

4. Client thread executed procedure (OS upcall),

5. Then the thread traps to the kernel upon completion of the work,

6. Kernel again changes back the address space and returns control to client,

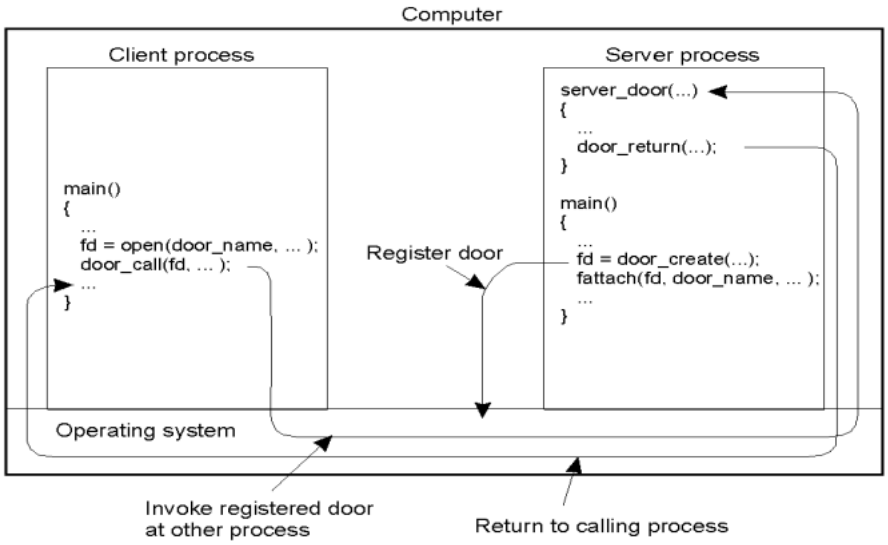

Figure 4.1: Lightweight RPC

Q: If the two processes are on the same machine, why do we need to use a networks stack because you are not communicating over network.

A: If you are using standard RPCs, your package will go down OS and the IP layer will realize they are the process on the same machine. It will come back up and call another process.

Note: RPCs are called "doors" in SUN-OS (Solaris).

4.3 Other RPC models

Traditional RPC uses Synchronous/blocked RPC, where the client gets blocked making an RPC call and gets resumed only after getting result from the called process. There are three other RPC models described below:

4.3.1 Asynchronous RPC

In Asynchronous or non-blocking RPC call, the client is not blocked after making an RPC call. Rather, client sends the request to the server and waits for an acknowledgement from the called process. Server can reply as soon as request is received and execute the procedure later. After getting the acceptance, client resumes the execution.

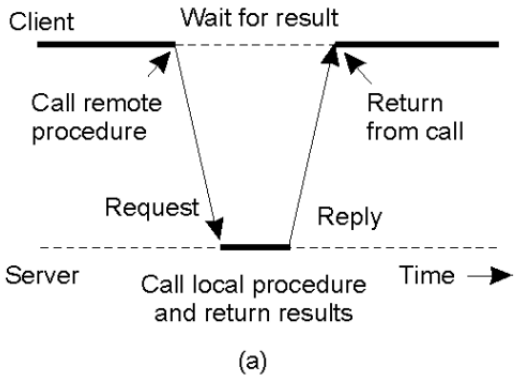

Figure 4.2: Traditional (Synchronous) RPC

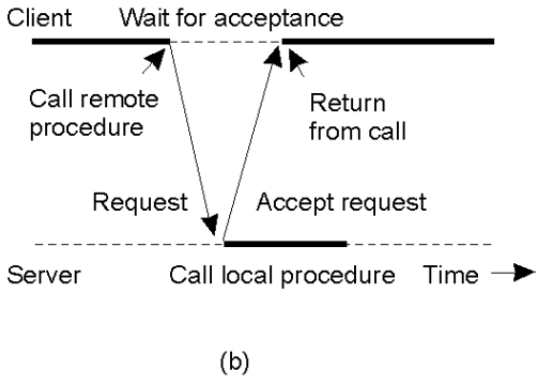

Figure 4.3: Asynchronous RPC

4.3.2 Deferred synchronous RPC

This is just a variant of non-blocking RPC. Client and server interact through two asynchronous RPCs. As compared to Asynchronous RPC, here the client needs a response for server but cannot wait for it, hence the server responds via its own asynchronous RPC call after completing the processing.

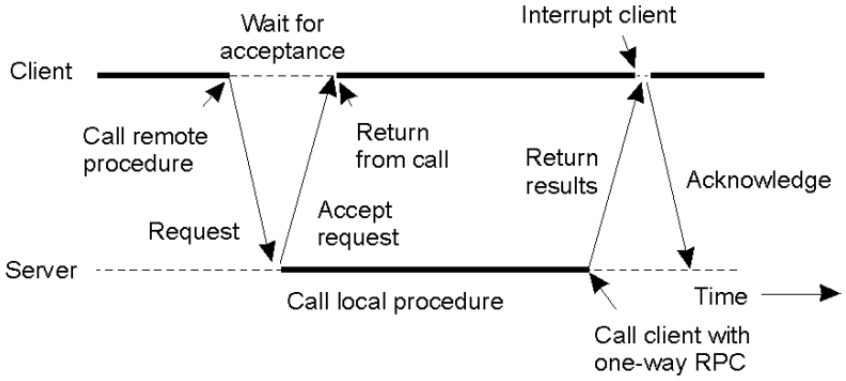

Figure 4.4: Deferred Synchronous RPC

4.3.3 One-way RPC

It is also a form of asynchronous RPC where the client does not even wait for an acknowledgment from the server. Client continues with its own execution after sending RPC call. This model has one disadvantage that it doesn't guarantee the reliability as the client doesn't know whether the request reaches the server or not.

4.4 Remote Method Invocation (RMI)

RMIs are RPCs in Object Oriented Programming Mode i.e., they can call the methods of the objects (instances of a class) which are residing on a remote machine. Here the objects hide the fact that they are remote. The function is called just like it is called on a local machine. For eg: obj.foo() , where *obj* is the object and *foo* is its public function.

Some important facts about RMIs:

1. There is separation between interface and the implementation as the interface is residing on the client machine whereas the implementation is on server machine.

2. It supports system-wide object references i.e parameters can be passed as object references here (which is not possible in normal RPC)

Figure 4.5 is showing an RMI call between the distributed objects. Just like a normal RPC, here also there is no need to setup socket connections separately by the programmer. Client stub is called the *proxy* and the server stub is called the *skeleton* and the instantiated object is one which is grayed in figure.

Now, when the client invokes the remote method, the RMI call comes to the *stub* (Proxy), it realizes that the object is on the remote machine. So it sets up the TCP/IP connection and the marshalled invocation is sent across the network. Then the server unpacks the message, perform the actions and send the marshalled reply back to the client.

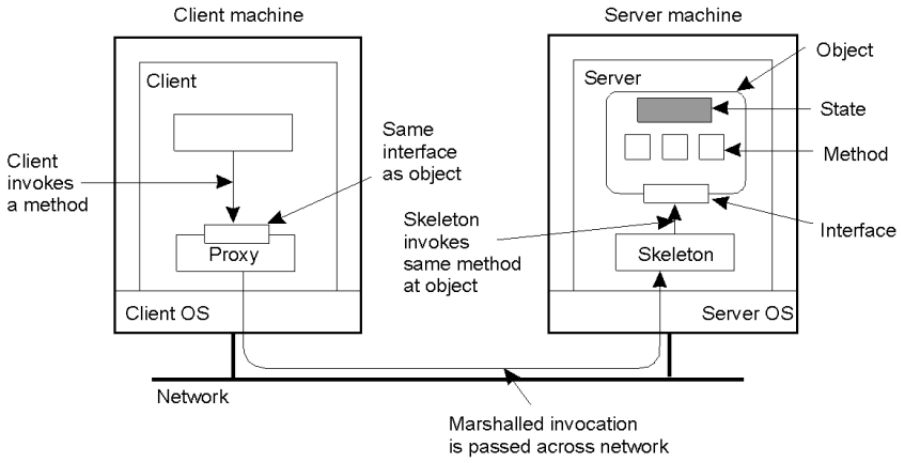

Figure 4.5: Distributed Objects

4.4.1 Proxies and Skeletons : Client and Server Stub

- Working of Proxy : Client stub

  1. Maintains server ID, endpoints, object ID.
  2. Sets up and tears down connection with the server
  3. Serializes (Marshalling) the local object parameters.

- Working of Skeleton : Server stub
  It deserializes and passes parameters to server and sends results back to the proxy.

Q: If the remote object has some local state (variable), you make a remote call and changed the variable, will the local machine see the change?

A: There's only one copy on the server and no copy of it on the client at all. The objects on the server and client are distinct objects. The change will be visible to subsequent methods from client. It doesn't mean there's a copy of the object on the client. If you make another call and see what's the value that variable, you'll get the new one.

4.4.2 Binding a Client to an Object

Binding can be of two types : implicit and explicit. Section (a) of Figure 4.6 shows an implicit binding, which is using just the global references and it is figured out on the run-time that it is a remote call (by the client stub). In section (b), explicit binding is shown, which is using both global and local references. Here, the client is explicitly calling a bind function before invoking the methods. Main difference between both the methods is written in Line 4 of the section (b), where the programmer has written an explicit call to the bind function.

```
Distr_object* obj_ref;              //Declare a systemwide object reference
obj_ref = ...;                      // Initialize the reference to a distributed object
obj_ref-> do_something();           // Implicitly bind and invoke a method
```

(a)

```
Distr_object obj_ref;               //Declare a systemwide object reference
Local_object* obj_ptr;              //Declare a pointer to local objects
obj_ref = ...;                      //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);            //Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();          //Invoke a method on the local proxy
```

(b)

Figure 4.6: Implicit and Explicit Binding of Clients to an Object

4.4.3 Parameter Passing

RMIs are less restrictive than RPCs as it supports system-wide object references. Here, Passing a reference to an object means passing a pointer to its memory address over the network. In Java, local objects are passed by value, and remote objects are passed by reference. Figure 4.7 shows an RMI call from Machine A (client) to the Machine C (server - called function is present on this machine) where Object O1 is passed as a local variable and Object O2 is passed as a reference variable. Machine C will maintain a copy of Object O1 and access Object O2 by dereferencing the pointer.

Note: Since a copy of Object O1 is passed to the Machine C, so if any changes are made to its private variable, then it won't be reflected in the Machine C. Also, Concurrency and synchronization need to be taken care of.

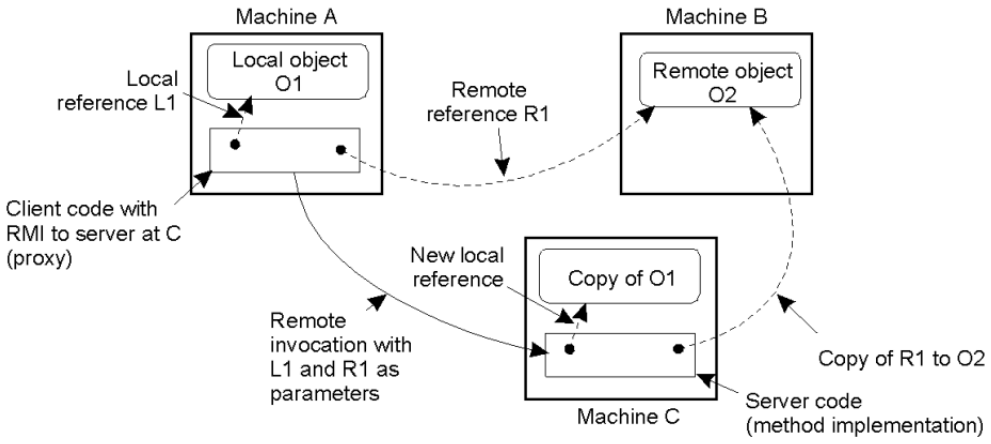

Figure 4.7: Parameter Passing : RMI

Q: How do network pointers interact with Java's garbage collection?
A: Garbage collection of Java is going to delete the memory that is not in use. A short answer is the remote machine shouldn't do garbage collection because you don't know if the object is being used by other machine.

Q: What is a remote reference?
A: A remote reference is an interface which allows to invoke a remote method.

4.5 Java RMI

- Server:
  - The server defines the interface and implements the interface methods. The server program creates a server object and registers object with "remote object" registry (Directory service).
- Client:
  - It looks up the server in remote object registry, and then make the normal call to the remote methods.
- Java tools:
  - `rmiregistry`: Server-side name server
  - `rmic`: Uses server interface to create client and server stub. it is a RMI Compiler, which creates an autogenerated code for stubs.

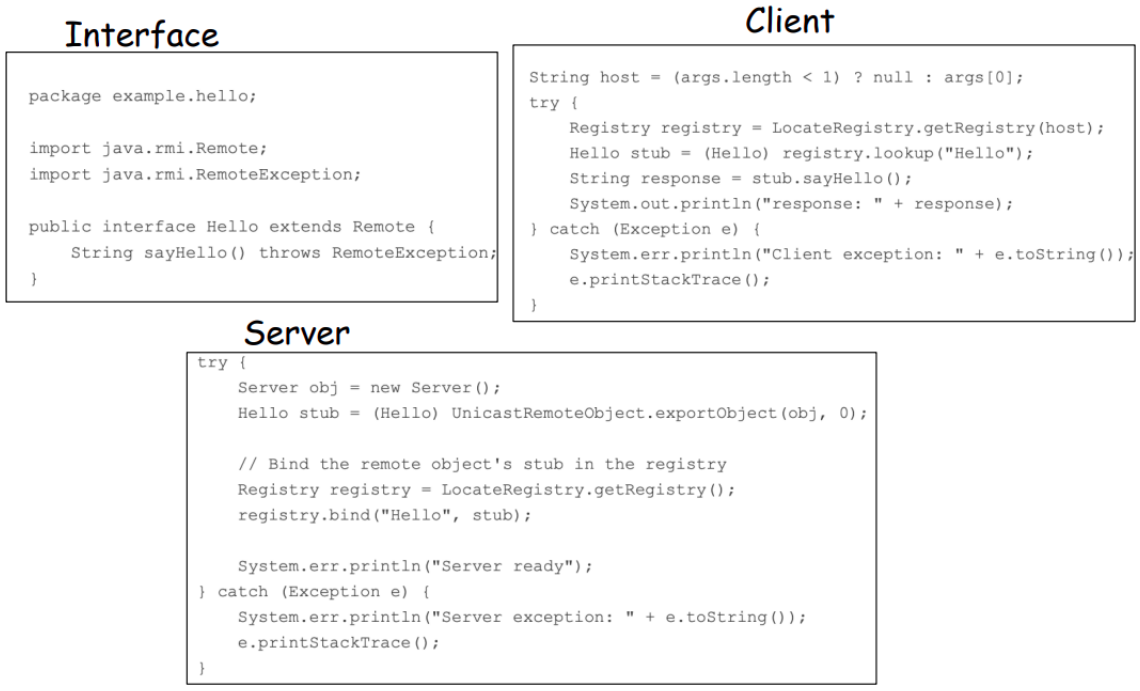

Figure 4.8: Java RMI Example Code Snippet

Q: How is interface code shared between client and server?
A: Interface is declared in server code and then imported in both client code. Server has to provide an implementation of the interface.

Q: Is Java RMI synchronous or asynchronous?
A: Default abstraction of Java RMI is synchronous.

Q: Where is the RMI registry running?
A: RMi registry can run on any machine. Client and server have to agree on which machine the registry is running on. Default machine is same as server.

4.5.1 Java RMI and Synchronisation

Java supports monitors , which are the synchronised objects. The same method can be used for remote method Invocation which allows concurrent requests to come in and synchronisation them. So for synchronisation, lock has to applied on the object which is distributed amongst the clients. How to implement the notion of the distributed lock?

1. Block at the server : Here, clients will make requests to the server, where they will contend for the lock and will be blocked (waiting for the lock).
2. Block at the client (proxy) collectively : They will have some protocol which decides which client will get the lock and rest others will be blocked (waiting for the lock)

Note: Java uses proxies for blocking (which means client side blocking). Applications need to implement distributed locking.

4.6 C/C++ RPC

Similar to Java RMIs. C++ defines interface in a specification file (*.x* file) which is fed to rpcgen compiler. rpcgen compiler generates stub code, which links server and client C code.

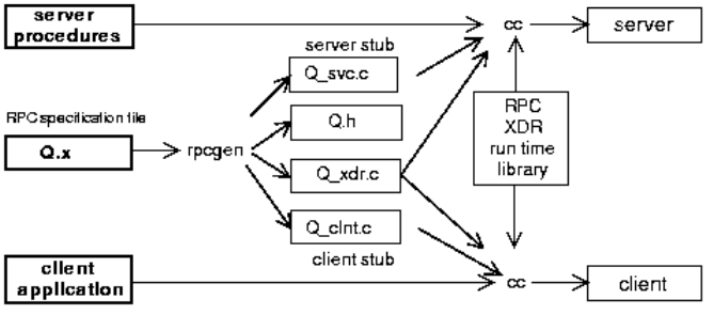

Figure 4.9: rpcgen Compiler

4.6.1 Binder: Port Mapper

Similar to `rmiregistry` in Java, it is a naming server for C/C++. It maintains a list of port mappings. Steps involved for port mapper are -

1. Server start-up: creates a port
2. Server stub calls `svc_register` to register program number, version number with local port mapper.
3. Port mapper stores prog number, version number, and port
4. Client start-up: call `clnt_create` to locate server port
5. Upon return, client can call procedures at the server

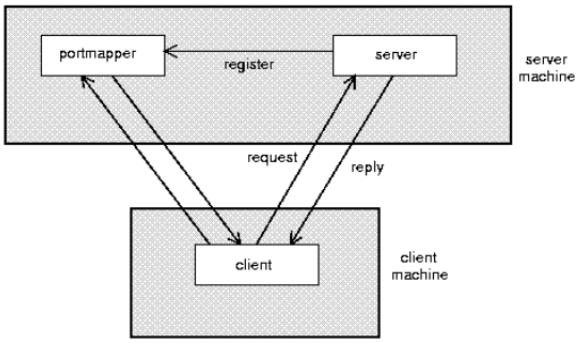

Figure 4.10: Port Mapper

4.7 Python Remote Objects (PyRO)

Basically an RMI for Python objects. It is a library that enables you to build applications in which objects can talk to each other over the network, with minimal programming effort. You can just use normal Python method calls to call objects on other machines.

Steps involved in using PyRO for Python RMI -

1. In server code,
   (a) PyRO daemon instantiated
   (b) Remote class registered aas PyRO object
   (c) Get URI so we can use it in the client later
   (d) Start the event loop of the server to wait for calls
2. Start server
3. In client code,
   (a) Get a Pyro proxy to the remote object using its URI
   (b) Call method normally
4. Start client (from remote machine)

Q: How is the client supposed to get the uri, if we're not copy-pasting it?
A: Pyro provides a name server that works like an automatic phone book. You can name your objects using logical names and use the name server to search for the corresponding uri.

4.8 gRPC

gRPC is Google's RPC platform, developed for their internal use but which is now open-source and available to all developers. It is a modern, high-performance framework for developing RPCs, which was designed for cloud based applications. gRPC is designed for high inter-operability - it works across OS, hardware,

and programming languages. Client and server do not have to be written in same langugae, gRPC supports multiple languages including (python, java, C++,C#, Go, Swift, Node.js, etc). It uses http/2 as transport protocol, and ProtoBuf for serializing structured messages. Http/2 is more efficient than TCP/IP, and ProtoBuf allows for interoperability.

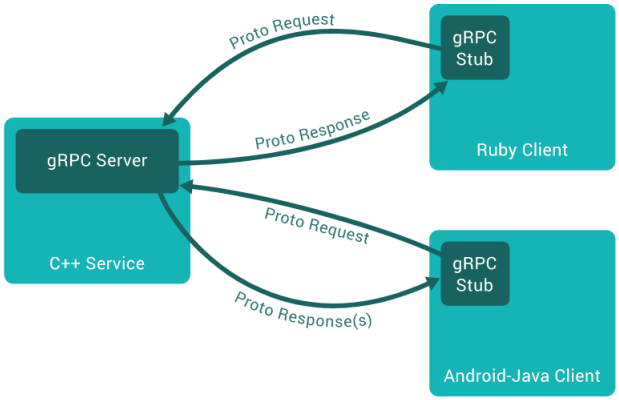

Figure 4.11: gRPC

4.8.1 Protocol Buffer (ProtoBuf)

ProtoBuf is a way to define a message and send it over a network which can be reconstructed at the other end without making any assumptions about the language, OS, or hardware used (platform independent). ProtoBuf has marshalling/serialization built-in.

A ProtoBuf message structure is defined in a `.proto` file (see Fig 4.12). It uses protocol compiler protoc to generate classes. Classes provide methods to access fields and serialize/parse from raw bytes e.g., set_page_number(). It is similar to JSON, but in binary format and more compact.

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

Figure 4.12: ProtoBuf Message Structure

Q: How does ProtoBuf handle objects?
A: ProtoBuf cannot send code, objects contain code. An object written in Java would not make sense in another language like Python. However you can send arbitrary data structure like arrays, vectors, hashmaps etc.

4.8.2 gRPC Example

1. Define gRPCs in proto file with RPC methods

- params and returns are protoBud messages
- use protoc to compile and get client stub code in preferred language

2. gRPC server code on server side

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Figure 4.13: gRPC Example

4.8.3 gRPC Features

- Four types of RPCs supported -
  1. Unary: Receive one request and send back single response
  2. Server Streaming: Server can return stream of responses.
  3. Client Streaming: Client can return stream of requests.
  4. Bi-directional: Client and server can both send stream of messages to each other.
- Supports synchronous and asynchronous calls
- Deadlines/timeouts: client specifies timeout, server can query to figure out how much time is left to produce reply
- Cancel RPC: server or client can cancel RPC to terminate it