CMPSCI 677 Operating Systems **Spring 2023**

Lecture 4: February 15

Lecturer: Prashant Shenoy
Scribe: Ge Shi (2019), Shubham Shetty (2022), Aishwarya Malgonde (2023)

Lecture 3 cont.

Questions on Failure semantics:

Q: What if the replies from the server were lost?
A: The client did not receive a response here so it will send a request again, this will be a problem when the server is making stateful responses - it will give an error saying the same request is being processed again. Solution is to make all the operations idempotent, irrespective of how many times a request is sent, it will be executed only once and the same response will be sent back every time.

Q: How does a system become idempotent?
A: This is a server side issue, if the server receives same request again it should execute it only once. The client is unaware what is happening at the client side.

Q: For example: Every time a person makes a payment, it generates a new transaction id, in this case how does the server know if its the same request? (here the server is idempotent)
A: The application will re-try with the same rpc request with the same transaction-id.

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 the calling process and the called process are on the same machine. They help in reducing the runtime.

Remember the two forms of communication of a distributed system – explicit (passing data) and implicit (sharing memory). You can think of using a special RPC system where both processes are on the same machine but, using a shared piece of memory instead of network messages. 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 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 bit/flag 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,

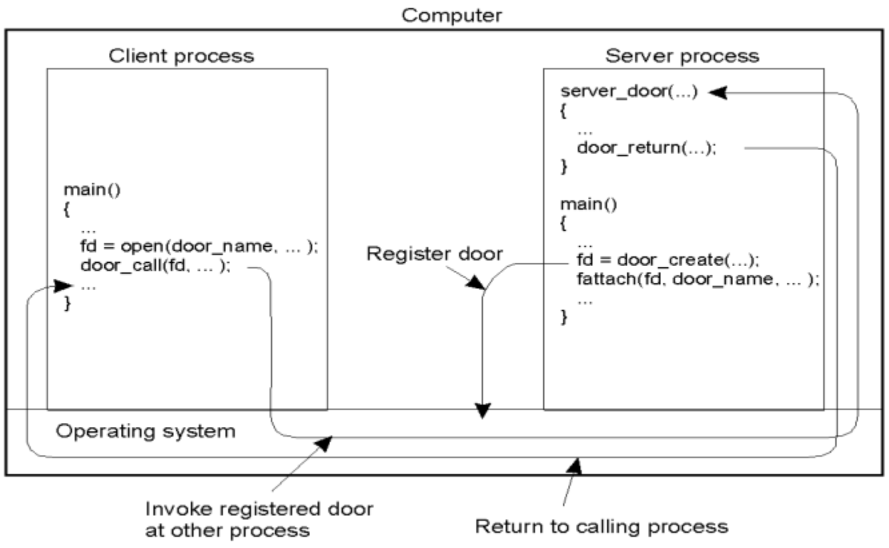


Figure 4.1: Lightweight RPC

Q: Can you use simple IPC instead of LRPC?
A: Yes, we can use IPC when both server and client are on the same machine, but shared memory was the preferred option here.

Q: First step in RPCs is a lookup where we try to locate the server, in LRPCs how do you figure out where the server is?
A: There is an inherent ability in LRPC to figure out where the server is running, it doesn't need to find the network port but it finds the process-id to communicate with the server.

Q: Where is the shared memory stored and is there a capacity limit?
A: Most OS have a concept called shared memory, for example: C/C++ uses malloc/new that allocates memory. In a similar way an OS can make a call which says 'make a shared memory segment' which it can explicit share with other processes. Shared memory segments are specifically designed to be shared across different processes, by default sharing memory is not allowed.

Q. How is the runtime bit determined?
A: It is automatically determined during the compile time or runtime by the program.

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.

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.

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

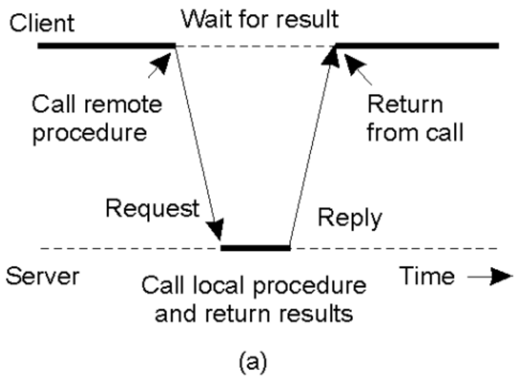


Figure 4.2: Traditional (Synchronous) RPC

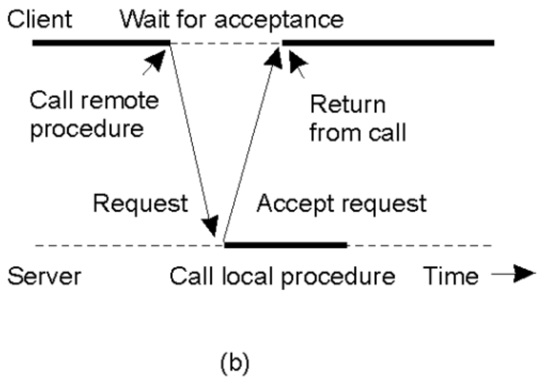


Figure 4.3: Asynchronous RPC

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

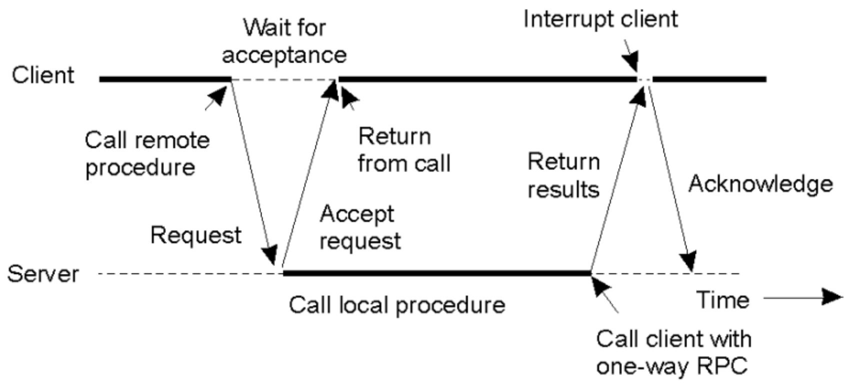


Figure 4.4: Deferred Synchronous RPC

is not possible in normal RPC)

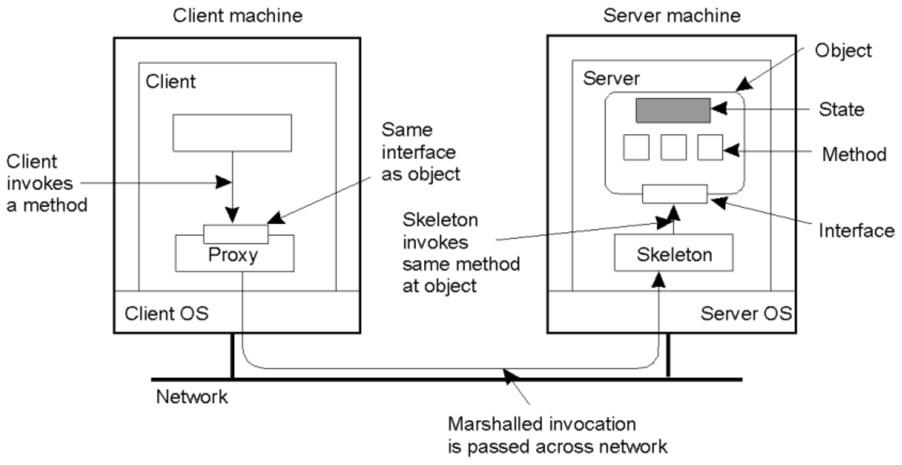


Figure 4.5: Distributed Objects

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.

4.4.1 Proxies and Skeletons : Client and Server Stub

- Working of Proxy : Client stub

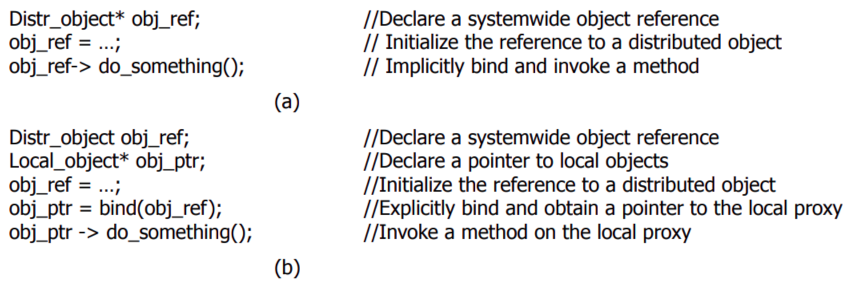
1. Maintains server ID, endpoints, object ID.
2. Sets up and tears down connection with the server
3. Serializes (Marshalling) the local object parameters.

- 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.

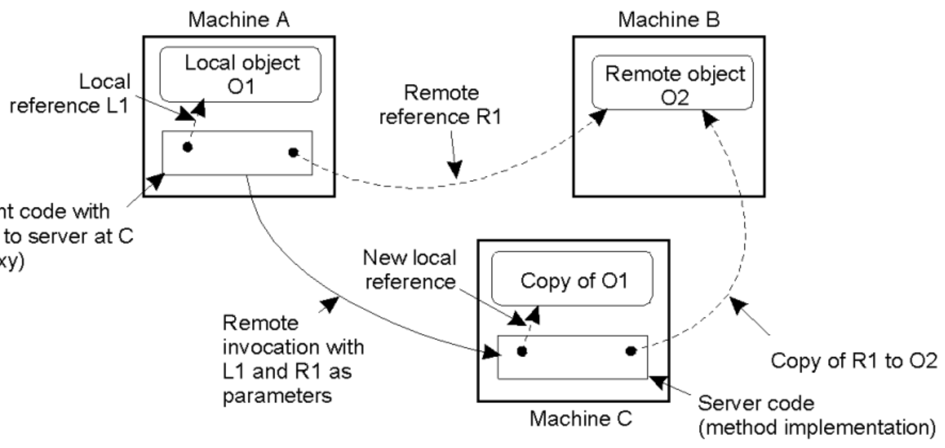


Figure 4.7: Parameter Passing : RMI

Q: Can Machine C make a call to O1 by reference? (Look at the figure above)
A: No, because the value of O1 is copied on Machine C. Call by reference only happens when the object is strictly a remote object / UnicastRemoteObject.

Q: What is the difference between RPCs and RMIs?
A: In an RPC you can only make call by value, whereas in an RMI the default call is call by value, but it has an exception where special remote objects can only be called by reference.

Q: How is the memory allocation work for a remote object on a local machine?
A: Since remote objects are passed by reference you don't have the actual object. When you call a method on that object, it is essentially going to go as a message over the network to that object.

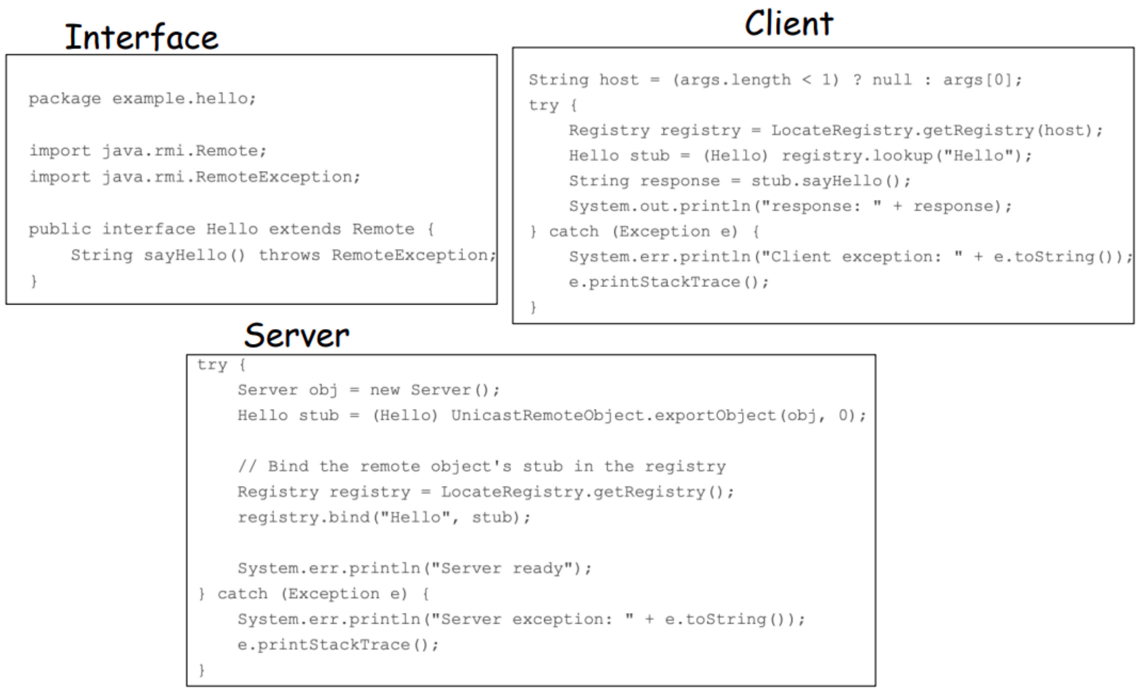
Q: How do network pointers interact with Java's garbage collection?
A: Garbage collection of Java is going to delete the memory that is not in use. A short answer is the remote machine shouldn't do garbage collection because you don't know if the object is being used by other machine.

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 binds it to the registry called as "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.



```
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

```
String host = (args.length < 1) ? null : args[0];
try {
    Registry registry = LocateRegistry.getRegistry(host);
    Hello stub = (Hello) registry.lookup("Hello");
    String response = stub.sayHello();
    System.out.println("response: " + response);
} catch (Exception e) {
    System.err.println("Client exception: " + e.toString());
    e.printStackTrace();
}
```

```
try {
    Server obj = new Server();
    Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

    // Bind the remote object's stub in the registry
    Registry registry = LocateRegistry.getRegistry();
    registry.bind("Hello", stub);

    System.err.println("Server ready");
} catch (Exception e) {
    System.err.println("Server exception: " + e.toString());
    e.printStackTrace();
}
```

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). This requires an explicit distributed locking across clients.

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.

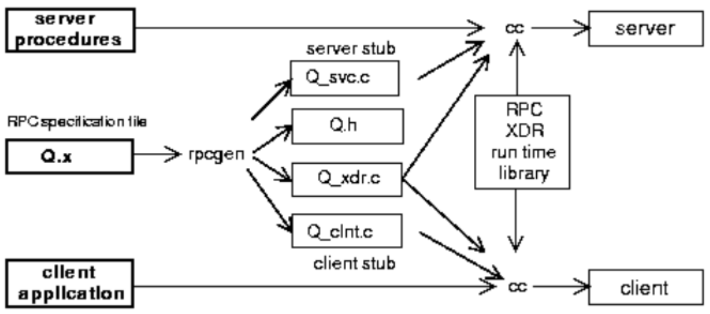


Figure 4.9: rpcgen Compiler

Rpcgen: Generating stubs

There is a **RPC specification file** which lists all the RPC methods that the server exposes and has details like methods names, arguments and more. It is a standard interface description of the server. A special compiler called the **rpcgen** compiles this specification file and automatically generates code called the stub code.

Q: Where does the specification file reside?
A: The specification file has to be common for both the server and the client and has to be created by the development team.

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

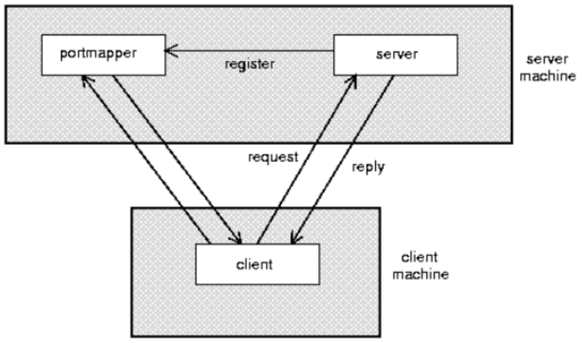


Figure 4.10: Port Mapper

Q: When you register with a port mapper (binder), how do you identify the server?
A: The server has to be given a name, the client needs to know this name and the request sent by the client is executed at the assigned port number.

Q: If the port mapper identifies the port, does the client need to identify where the server is running (ip address)?
A: RPC systems atleast require you to figure what machine the server is running, but not the port itself. There can be two ways to handle this - first, the client can figure out by connecting to the port mapper on the machine. Second, there can be a network wide service lookup where you can search using the server name and get the location of where the server is running.

Q: Does there have to be a mapping of the host name?
A: That is done by the DNS, they can translate machine names to ip addresses.

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: Is PyRO wrapping RPC?
A: Think of PyRO as a RPC runtime in python, it is going to do all the heavy lifting and simplify implementation for you.

Q: Is PyRO compatible with Python 3?
A: Yes

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.

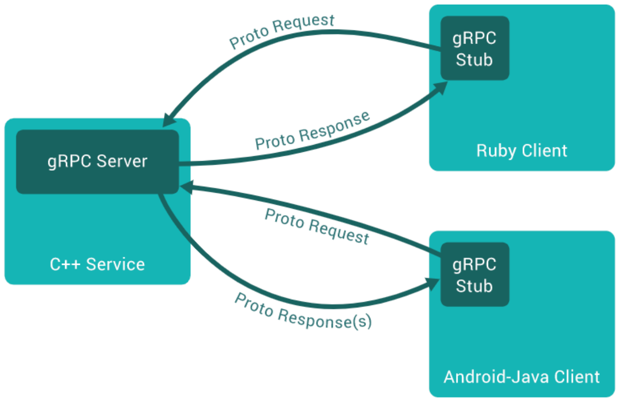


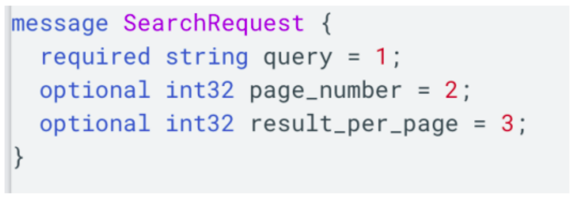
Figure 4.11: gRPC

Q: Is gRPC faster than RPC?
A: It depends on the design because there are http overheads in gRPC. Standard request-response will be faster in TCP but gRPC supports 4 types of RPC calls - unary, server streaming, client streaming and bi-directional (see section 4.8.3). A standard RPC is unary (including rmi and pyro) and cannot do any kind of streaming. So gRPC amortizes the overhead by streaming.

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: Why does PyRO not need a ProtoBuf like system?
A: In PyRO, both the client and the server are python programs. PyRO has its own data formats and it has internally declared how messages and arguments are to be sent. ProtoBuf is helpful for language independence which is a feature of gRPC.

Q: Since python language is interpreted do you need to know method definitions before?
A: Yes, because interpreted doesn't mean you can bind at runtime. The client needs to know what the server methods are.

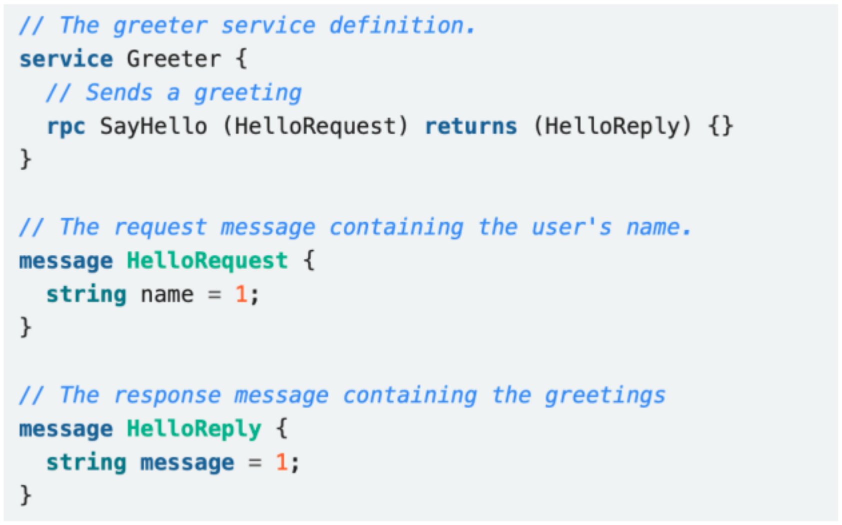
Q: How does ProtoBuf handle objects?
A: ProtoBuf cannot send code, objects contain code. An object written in Java would not make sense in another language like Python. However you can send arbitrary data structure like arrays, vectors, hashmaps etc.

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

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 send stream of requests.



```
// 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. 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

Q: Do we use http in gRPC because we are streaming?
A: No, http is a common way by which services communicate. gRPC uses a specific version called http/2, it has some optimizations which makes such interactions efficient.

Q: Does gRPC need a web server because of http? And is it another version of REST protocol?
A: A gRPC server will have the capability to serve a rpc http request, no web server is required. A REST api requires urls as resources which are pre-defined, in gRPC we are just defining methods and responses which are layered with http transfer.