CMPSCI 677 Operating Systems **Spring 2016**

Lecture 8: February 17

Lecturer: Prashant Shenoy *Scribe: Ravi Choudhary*

8.1 Communications in Distributed Systems

This lecture will deal with communication between different processes in distributed systems. Basic background of networking and RPCs will also be covered.

In a distributed system there are more than one component that are communicating over the network. Communication between any two components is inherent in any distributed system and the question is what abstraction the system should provide to enable this communication. At a very high level communication can be of two types

Unstructured Communication - It is essentially sharing memory or sharing data structures. Data can be put in shared buffer, another process can come and pick it up. In this manner process can interact with each other and pass messages between them. Application designer has to decide whether to use shared memory or shared data structures. No network support is required in such communications. But this is not a popular way of message passing in commercial systems.

Structured Communication - Most of the communication in distributed system is structured. These constitute of IPC's (Interprocess communications), RPC's or sending explicit message over sockets. Processes on same machine can use either of the above two methods. However, in Distributed Systems processes are on different machines. To communicate across machine, networking is required to pass messages. In this course mainly Structured Communication will be discussed.

8.1.1 Communication Protocols

This section gives a brief recap of the network protocol stack. Suppose an application on Machine A wants to communicate with an application on Machine B, the message from application needs to be processed through several layers of stack on Machine A, then go over a physical network and then reverse processing takes place in the protocol stack of Machine B, which then delivers the message to the application. Each layer has protocol which it follows to communicate with other layer in the network stack. Different layers in the network stack are discussed below :-

Physical Layer - It is essentially the lowest level of the stack (Ethernet, WiFi, 3G etc) which is the medium for communication between devices on network.

DataLink Layer - This layer is known as MAC protocol which is used to construct the packet in lowest layer and puts it out on the medium, essentially the ethernet protocol runs on this layer. Data packets are encoded and decoded into bits. It furnishes transmission protocol knowledge and management and handles errors in the physical layer, flow control and frame synchronization.

Network Layer - Routing and forwarding are functions of this layer, as well as *addressing, internetworking, error handling, congestion control and packet sequencing.* This layer is responsible for hop by hop

communication. A data packet sent out by machine A might have to traverse through multiple hops before it reaches its final destination. When packet is delivered at one hop it decides what next has to be taken at this point until it reaches its final destination.

Transport Layer - This layer is responsible for end to end communication. This layer provides transparent transfer of data between end systems, or hosts, and is responsible for end-to-end error recovery and flow control (e.g. rate of transmission etc.). It ensures complete data transfer.

Application Layer - This layer supports application and end-user processes. This layer provides application services for file transfers, e-mail, and other network software services. Telnet, HTTP and FTP are applications that exist entirely in the application level.

8.1.2 Layered Protocols

The actual message which needs to be sent to a destination is called *Payload*, as the message traverses down the network stack each layer processes the message and adds a headers to the message packet. For example, *http* is an application layer protocol, thus http header may be added in front of the payload. In Transport Layer, TCP will add a header to the payload. In Network Layer, IP will add a header which denotes the source and destination addresses of the machines which is used at each hop to identity where the packet came from and where it is headed to according to which routing of that packet is done. In DataLink layer, suppose ethernet is being used to transmit the message over network, then a ethernet header is added to the payload. At the trailer end, there can be a checksum which is used to check for errors in transmission. At the receiving end, each layer analyses the header and then strips it off and hands it to the layer above it. Receiver application layer receives the original message passed from sender application layer.

8.1.3 Middleware Protocols

In Distributed Systems environment, the middleware layer sits in between Application Layer and the Transport Layer or the Operating System. The middleware protocol provides higher level service to application which is even much more advanced than session and presentation layer. For example, Multicast, Broadcast, Unicast services are provided by the middleware. Application just needs to send a packet for broadcast to the middleware and then later takes care of sending the message to the network. Essentially middleware abstracts the Application from the TCP / IP and other networking aspects of the distributed system. This will be discussed more in upcoming classes.

8.1.4 Client-Server TCP

In a typical client server architecture, client constructs a request and sends it over to the server, server processes the request at its end and sends back the response to the client. The diagram in the slides show how message passing is done using TCP / IP. For a client to send a message to the server, the client has to establish a TCP connection to the server first, i.e. a 3-way handshake is required to establish a TCP connection. The client sends a SYN packet to the server, which means that the client wants to establish a TCP connection with the server. Server in turn sends back a SYN and a ACK(SYN) packet back to the client. ACK(SYN) is the acknowledgement sent from the server to the client and SYN is sent to the client to denote that server wants to setup a connection with it. The client then acknowledges the SYN request of server by sending a ACK(SYN) message back to the server. Thus this is a 3-way handshake mechanism to establish a TCP connection. Only after this TCP connection have been setup, the client can send a request message to the server. After the client sends the request to the server, it sends a message FIN to the server denoting that the client has finished sending the message to the server. Now server processes the request

and while it is processing the request it sends back a ACK(req + FIN) to denote that it has received the request and finish message. Once the request is processed in the server the answer is send back to the client and then a FIN message is sent to the client to denote end of reply. The client then acknowledges back to the server by sending ACK(FIN) message that it has received the answer. The last 3 messages were to tear down the TCP connection. Thus we see that in total 9 packets were sent for the whole process which in turn results in overhead.

To overcome this overhead, Client-Server TCP is used to optimize the entire process. Essentially the messages are batched together to reduce the overhead. The first packet sent from client is (SYN, request, FIN) , which means, that client wants to setup TCP connection with the server, the request is embedded in the same packet and then the client says it is done sending the message. The server then processes the request and sends back SYN, ACK(FIN), answer, FIN which means that it wants to set up a connection with the client, then acknowledge clients Finish request, answer to the request and then finish message from the server side. Then client just send ACK(FIN) which means that it has received the answer from the server and TCP connection can be torn down now. This architecture was merely a proposal but in real it is seldom used. This architecture is most useful for sending one message and receiving one reply from server. Otherwise in general, the 1st method of sending messages is used.

8.1.5 Push or Pull Architecture

- **Client - Pull Architecture**: The architecture that we discussed in the previous section is client pull architecture. The client sends in request to the server to *pull* data from it. For example, HTTP is a client pull architecture protocol where a client sends a webpage request to get a reply from the web server. The pros and cons of this architecture are the following:
  - Pros:
    - In this type of architecture the servers are *stateless* , i.e. servers don't need to keep track of the clients its communicating with. If the client wants to get some more data, it will send a new request to the server, but server doesn't need to worry about it.
    - From a failure prospective this architecture is more *robust* since no state of clients is maintained, thus if the server crashes, the client just needs to send a new request for the data.
  - Cons: This architecture is more scalable from memory point of view since states of client need not be maintained on the server. But this architecture is not efficient from communications point of view. To get one piece of data two messages are exchanged i.e. a request and a reply, which results in a overhead.
- **Server - Push Architecture**: In this architecture, the client may not send explicit requests for data, server asynchronously pushes data to the client. For example, in Video and Audio Streaming, the client just sends a one time request for the data on the server, after that the server starts pushing data to the client at certain intervals, the client needn't send explicit requests for different pieces of video file from the server. The client just listens to the socket and data keeps coming in from the server. The pros and cons of this architecture are the following:
  - Cons:
    - In this type of architecture the servers are *stateful*, i.e. the servers keep tracks of the clients its communicating with. The video can be a lengthy clip and server needs to know to which client it has to push the data and at what intervals.
    - This architecture is *not robust* from failure point of view. If the server reboots, then all the state of clients is lost from memory and the server will lose track of the data it was streaming to different clients, thus everything will have to done from the start.

- Pros: This architecture is less scalable from memory point of view since states of clients needs to be maintained in the servers memory. But from communication point of view this architecture is more efficient, since client just sends in one request and server starts pushing data to the client which has less overhead than client - pull architecture.

Thus during the design of Distributed System architecture we need to keep in mind whether we want a Push based or Pull based architecture.

8.2 Group Communication

This is one-to-many communication, also referred to as broadcast or multicast where the same message is sent to more than one recipient. For example, sending an email to a mailing list, in where one email is sent to multiple users at once who have subscribed to that mailing list. Same can be done for network messages, clients can subscribe to a group and anyone who posts a message to that group will be sent to all the clients automatically. For example, audio broadcast, where lot of users have tuned in to listen to it, in this case server / sender is not sending the message to each of the users separately, it sends a message to the middleware and this middleware then sends the message to users in the group.

8.2.1 Remote Procedure Calls

The goal of Remote Procedure Calls(RPCs) is to make distributed computing look like centralized computing. It allows remote services to be called as procedures. Rather than sending abstract messages between a client and a server and letting the server process the message and send back the response, the idea was to directly let the client call functions on the server. This makes things easier for the programmer as now he/she just have to call a remote function on the server instead of sending explicit message to the server.

One example of RPC implementation is given on the lecture slides. The server has a function *add()* which accepts two integers as addends. The client calls this remote function on the server and passes two integers which needs to be added. When client calls this *add()* function, the RPC runtime system internally sends a request to the server, which in turn returns the value.

The RPC takes care of passing the variables to the server, constructing a message to call the function, TCP/IP connection etc, the programmer needn't deal with all this complexity, he/she just need call the function as if it was a local function call. C doesn't have support for RPC. It has library which provides API to use RPCs. Java has inbuilt RPC support.

8.2.2 RPC Parameter Passing

There are two choices to offer the local procedure parameter passing. You can call-by-value or call-by-reference. Call-by-value says that when you provide an argument to a function call, you make a copy of that variable and give it to the call function. When you call-by-reference, you are giving a reference, essentially a pointer to the argument, so that means you can actually change the original value as well.

While calling Remote procedures, the user can't pass parameter by reference, i.e. a pointer to an address space can't be passed to the server because when it is dereferenced, it will not have the same value which the address space on client was holding. Thus the user needs to pass parameter by value. Likewise, global variables are not allowed in RPC.

8.2.3 Client and Server Stubs

The system generates stubs, these are essentially proxies at both ends that system generates for the application that deals with setting up a network connection to the server, constructing a message whenever a remote call is made, sending the message, wait for reply, all of this is taken care by these stubs. All of this is abstracted from the programmer with the help of stubs.

Flattening and Marshaling of arguments, it takes care of the arguments passed to the function call, whether it be a structure or an array, whether it is calling the function on the server with different OS or hardware architecture (little endian or big endian), everything is taken care by the stubs which converts native data format into universally agreed format, external data representation (XDR), thus the programmer needn't worry about all these complications.

8.2.4 Binder: Port Mapper

When the client and the server startup, and the client invokes a message at the server, it needs to first find the server. This is done through Port Mapper, when the server starts up it registers itself at the port mapper with the list of API's that it exposes and the port that it is listening on. Therefore when the client starts up, it first checks with the port mapper for the server with the remote procedure that the client wants to execute. From the port mapper it gets the server's port number, this is a one time setup. Once this is done, the stub takes over and performs required operations. Also it is assumed that the IP address of the server is known but which port it is listening on is unknown, which is then found through port mapper.

8.2.5 Rpcgen: generating stubs

Here the professor shows an example of RPC systems. This is how you write a program to run(see the picture in the corresponding slide). You're going to write a specification file, also in some languages called an ideal interface definition. This is going to say that these are the procedures that the server are supporting. Then you write a client code and server code. Server implements many procedures for its own application and client is going to call some of these procedures.

So you take all of this code, take the interface and running to a special compiler, which is RPC compiler, in C this is called Rpcgen compiler. So you run it to Rpcgen, it will auto generate some code for you such as the client stub code, server stub code and some additional header files. It also generates the XDR code which linked the client and server. You got some additional c files in this case.

Then you compile the client code and the generated code together. The resulting client code and client stub object files are then linked with the runtime library to produce the executable binary for the client. Similarly, the server code and server stub are compiled and linked to produce the server's binary. At runtime, the client and server are started so that the application is actually executed as well.

Therefore, there are two compilers, RPC compiler and actual C compiler. The RPC compiler will first run and generate some additional code. Then you take that generated code and written code together and compile the applications. This is an standard pipeline for any RPC-like system.

8.2.6 Steps of a Remote Procedure Call

Let us examine how remote procedure calls are implemented. The entire trick in making remote procedure calls work is in the creation of stub functions that make it appear to the user that the call is really local.

On the client, a stub function looks like the function that the user intends to call but really contains code for sending and receiving messages over a network. The sequence of operations that takes place, shown in Figure below, is

- The client calls a local procedure, called the client stub.To the client process, this appears to be the actual procedure, because it is a regular local procedure.
- Network messages are sent by the client stub to the remote system (via a system call to the local kernel using sockets interfaces).
- Network messages are transferred by the kernel to the remote system via some protocol (either connectionless or connection-oriented).
- A server stub, sometimes called the skeleton, receives the messages on the server. It unmarshals the arguments from the messages and, if necessary, converts them from a standard network format into a machine-specific form.
- The server stub calls the server function (which, to the client, is the remote procedure), passing it the arguments that it received from the client.
- When the server function is finished, it returns to the server stub with its return values.
- The server stub converts the return values, if necessary, and marshals them into one or more network messages to send to the client stub.
- Messages get sent back across the network to the client stub.
- The client stub reads the messages from the local kernel.
- The client stub then returns the results to the client function, converting them from the network representation to a local one if necessary.

8.2.7 Marshalling

Marshalling is a data presentation conversion, performed according to special rules, usually for network transfer. The following data presentation factors have to be took into account to perform marshalling.

- Different platforms use their own data formats. Byte order is also dependent on processor (Big Endian or Little Endian). Standard representation of data across the machines can resolve this issue. For example, External data representation (XDR) protocol is useful for transferring data between different computer architectures and has been used to communicate data between very diverse machines.
- Passing pointers between different machines is a problem, different machines have different memory allocated to their variables. If the passed pointer tries to deference it variable then it may get garbage value. We can overcome this problem if the pointer points to a well-defined data structure, then pass a copy and the server stub passes a pointer to the local copy.
- Another problem in marshalling is how to deal with pointers. Possible solutions are either prohibit the use of pointers or convert the local client pointer into a network pointer, also known as generalization of pointer which allow us to deference the pointer over the network.

8.2.8 Binding

An issue that we have glossed over so far is how the client locates the server.

- One method is just to hardwire the network address of the server into the client. The trouble with this approach is that it is extremely inflexible. If the server moves or if the server is replicated or if the interface changes, numerous programs will have to be found and recompiled.
- To avoid all these problems, some distributed systems use what is called dynamic binding to match up clients and servers. We use directory service where server register itself at start-up WITH the directory service (in Unix also known as binder or RPC port mapper) and the server tells the name of the server, the version number, and a list of procedures provided by the server (read, write, create, and delete). And when client calls one of the remote procedures for the first time, say, read, The client stub sees that it is not yet bound to a server, so it tries to find out where the server is by contacting the directory service if a suitable server exists, the directory service gives its handle and unique identifier to the client stub.

8.2.8.1 Binding: Comments

- The extra overhead of exporting and importing interfaces costs time.
- Since many client processes are short lived and each process has to start all over again, the effect may be significant.
- In a large DS, the binder may become a bottleneck, multiple binders are needed whenever an interface is registered or deregistered, a substantial number of messages will be needed to keep all the binders synchronized and up to date, creating even more overhead.

8.2.9 Failure Semantics

Failures should be taken into consideration when RPC calls are made. Failures can be due to client or server crash or data transmission loss in the network, thus we need to have predictable behavior in scenarios where such failures can occur. There should be mechanisms to figure out the cause of failures and the steps to be taken to overcome them.

- If the client is unable to find the server, then the client should return an error message.
- If client and server both are running but something goes wrong on the network and RPC request packet doesnt reach the server for processing or the server sent the reply back to client but client never received the reply. To handle this scenario the client should implement a timeout mechanism. If the client doesnt receive response from server before the timeout then client should resend the RPC request.
- Another approach of averting failures is by offloading the error correction task to the TCP layer, since TCP provides abstraction where the packet losses are dealt by the transport layer. Thus any packet that is sent will arrive at the other end eventually.
- If the RPC system is running over UDP, then timeout mechanism needs to be in place since UDP doesnt implement any error correction or handles packet losses in the network.
- If UDP is used to transfer data, before making a request to a stateful server one need to check if the RPC calls are idempotent, i.e. re-executing the calls on the server shouldnt cause errors.

8.2.9.1 Server Failure Semantics

There can be scenarios where the server crashes while it is executing the RPC call, thus client should be aware of what was the state of the server before it crashed, so that it will know what action to take when the server comes up. This can be handled by different semantics provided by the RPC system.

- At least once semantics, denotes that if the client received a reply from the server, it means that call has been executed at least once on the server. It may be possible server executed the call and then crashed before sending the reply, it then comes up and then again executes the call and sends the reply to the client.
- At most once semantics, if the client gets a reply from the server, it means that the RPC call have been executed at most once on the server.
- Exactly once semantics, this is the most desirable scenario, no matter what happened on the server side (crashes / restarts), if the client gets back reply exactly once, then it is confirmed that the RPC call was executed exactly once on the server.

Exactly once semantics are difficult to implement and systems generally implement weaker semantics and thus the error handling needs to be taken care explicitly.

8.2.9.2 Client Failure Semantics

There can be failures scenarios at the client side too. For example, client made a RPC request, but before the client received the response from the server, it crashed, in this case server will need to know the corrective action in order to maintain system sanity. Thus there are failure semantics provided by the RPC system to handle such scenarios.

- If server executed a RPC call, but mean while the client crashed and is not available to receive the response, then most servers will discard the response.
- If server was executing a long running RPC computation and in between the execution it comes to know that client has crashed then such calls are called Orphan RPC calls. In this scenario server can do one of the following
  - Extermination - when the server gets to know that client has crashed, it can terminate the ongoing RPC computation. This will incur an overhead on server side to maintain list of clients and their current state.
  - Reincarnation - In order to minimize the overhead, the server can check for long running RPC calls within certain periodic time intervals. If it finds that a certain RPC call has been running for a long duration, then it goes and checks for the client state, if the client has crashed then server can delete the computation. Thus server doesnt need to keep state of every client connected to it.
  - Gentle Reincarnation - If the client that made the request has crashed, then server can broadcast and check if the client restarted with a different process id and then try to deliver the RPC response to it.
  - Expiration - Every time a RPC computation starts, the server gives it a timeout, once the timeout happens it checks if the client is still alive, and if it is then it continues computation with another timeout, this process recursively goes on until the response is sent back to the client

8.2.10 Implementation Issues

RPC on LAN using UDP works fine because the transmission losses are less and data packets dont get lost in the network, but it wont pan out well when sent over WAN since routers gets congested and packet dropping etc might disrupt the RPCs. TCP works out to be a better way to send packets over WAN because packet drops, error correction everything is handled by TCP. Thus a RPC programmer has to choose which protocol he/she wants to implement for RPC.

There is a lot of overhead in sending the RPC message over the network, when a RPC call is made, the call goes from the client into the stub, in the stub the parameters are copied and then a message is formed to send it to the server, the message is then handed to the kernel which then is copied on to the NIC card and sent over the medium, similar process is done at the server side. Thus we see that lot of copies of the message are made which can contribute to overhead, specially when there are lot of message with fairly large number of arguments

8.2.11 Case Study : SUNRPC

To implement NFS, SUN Microsystems choose to use RPCs rather than doing socket communication between the client machine and the File Server. Thus they built an RPC layer and implemented NFS on top of it. They realized that the RPC layer was more generic, therefore other applications could be written using the same RPC abstraction, thus they made tools like RPC compilers available to programmers. This system was initially built on top of UDP since it was designed to work on LAN, but now-a-days it can work both on UDP and TCP which depends on the discretion of the programmer and application it is designed for. To implement Marshaling and Un-marshaling SUN came up with XDR (eXternal Data Representation), i.e. any packets that are sent out with arguments is converted to XDR format and sent as a message. SUNRPC provides At-least once semantics and thus not a preferred way, and calls need to be idempotent to avoid redundancies.