CMPSCI 677 Operating Systems **Spring 2018**

Lecture 8: February 14

Lecturer: Prashant Shenoy *Scribe:* ***Sanjay Reddy S***

8.1 Communication in distributed systems

In a distributed system, each entity may want to share information with other distributed entities. For example, a temperature sensor may want to share its information with a climate control system. The processes run on different machines and the applications implemented by these processes might include communication between them.

8.1.1 Types of communication in distributed systems

Communication among distributed processes can be categorized into:
1. Unstructured communication - uses shared memory or shared data structures; Update a distributed shared memory to let others know of your local state. Others can then read the distributed shared memory to get your local state. No explicit messages are used.
2. Structured communication - also called as 'message passing', this uses explicit messages (or inter-process communication mechanisms) over network. There is no shared memory in this type of communication.

8.1.2 Distributed communication protocols

Communication Protocols : Protocols are a set of rules for communication agreed by all the entities participating in communication. Protocols can be connection oriented (TCP) or connectionless (UDP). In a layered network model, when a message is sent over network, each layer adds its own header and trailer information. (In brief: Physical layer corresponds to the actual transmission of bits on Network [using WiFi/cable]; Data Link or MAC layer creates 'frames' and passes down; Network Layer is involved in hop by hop communication and all the routing mechanisms come into play here; Transport Layer usually is end to end communication. Flow and error control [as in TCP] are present here).

Middleware Protocols : Middleware layer resides between an application and the operating system. Middleware layer may implement its own general purpose protocol that may even result in more layers. Example: Distributed system. As part of this course, we deal a lot, with this Middleware layer.

Client - Server using TCP : In a standard TCP communication, for just a single request from client we need almost 9 packets (and that too assuming there are no losses). In a TCP connection, every packet needs to be acknowledged. Figure 8.2 (a) shows this in detail (along with the customary 3-way handshake to initiate the connection and tear it down). To make this process efficient (especially for short request scenarios from client side, as above) mechanisms like grouping do exist (Figure 8.2 (b))

8.1.3 Communication Models

Client pull architecture : Clients pull data from servers by sending requests. Example: HTTP. This

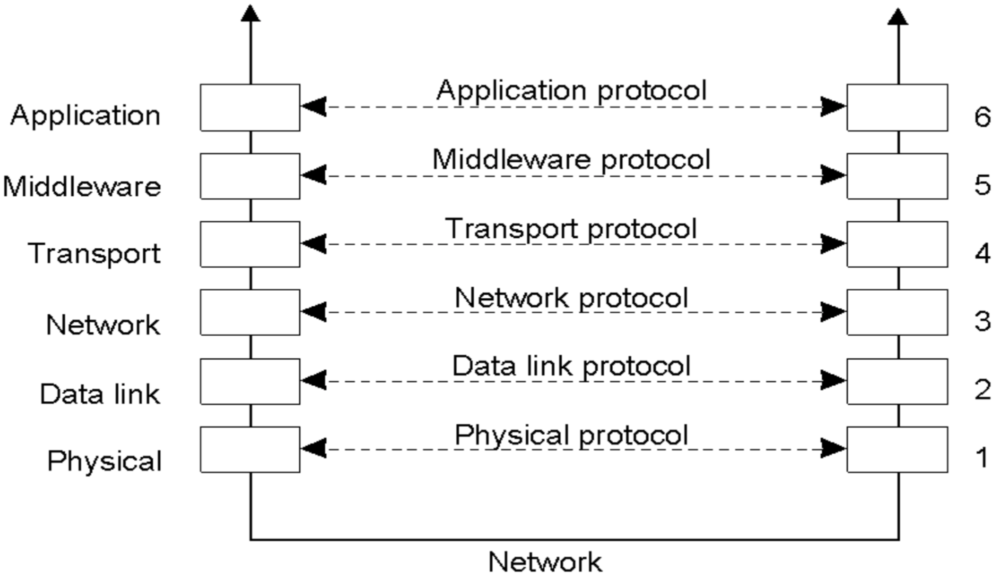

Figure 8.1: Middleware protocol

model is the more commonly seen one.
Pros: Server need not maintain state information, easy failure handling
Cons: Scalability problem (one reason is the overhead cause by lot of messages being exchanged), fault tolerance

Server push architecture : Servers push data to clients. Example: Video streaming, stock tickers
Pros: Relatively more scalable (one reason is because the client doesn't have to continuously poll the server for fresh data and the server automatically pushes when new data arrives.)
Cons: Servers have to maintain client state information, less resilient to failures.

8.1.4 Group communication

In the previous two we assumed there is only one client and one server (1-1 communication). Distributed communication usually requires one-to-many communication and messages can be delivered to entire groups instead of a unique entity (often called multi-cast). Eg- Webcast, where one video is sent to multiple receivers. This will be dealt more in depth in future lectures

Issues in group communication:

- **Static/dynamic groups** : Should groups be static or dynamic? Can an entity change groups dynamically?
- **Closed/open** : Can entities outside the group communicate with entities within the group?
- **Group addressing choices** : Broadcast, multicast, application level multicast (still unicast at network level)
- **Other issues** : Atomicity, Ordering of messages, scalability

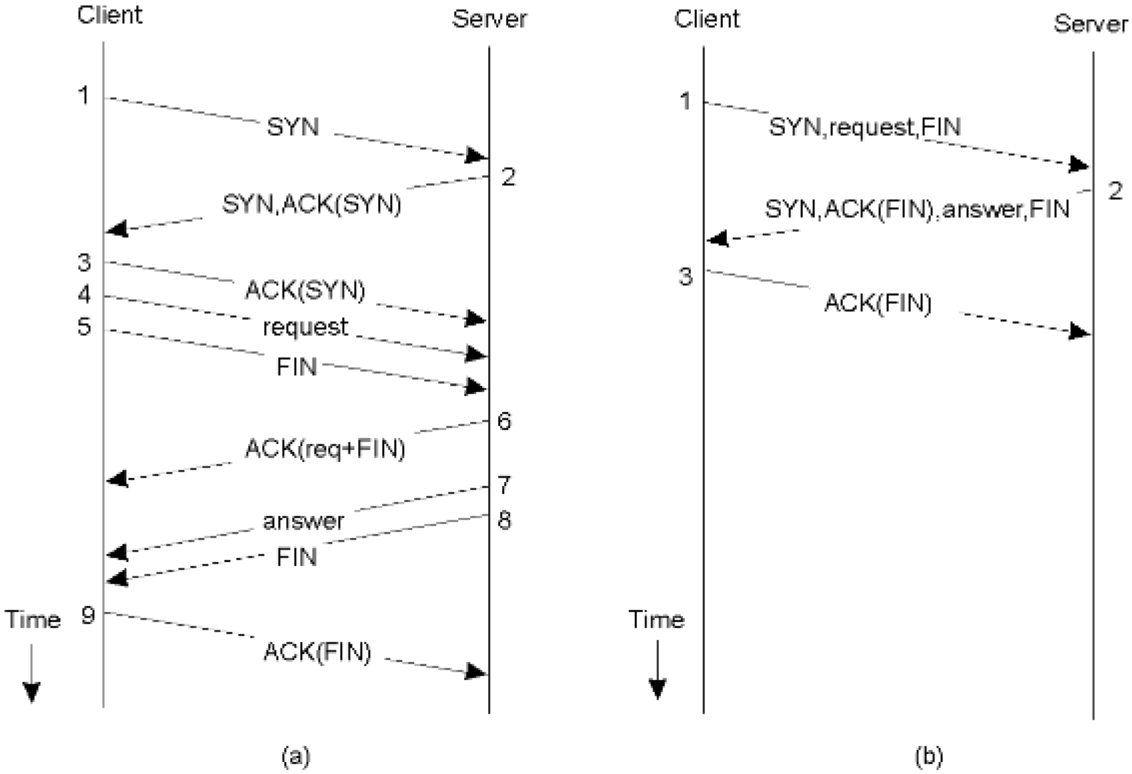

Figure 8.2: TCP connection

8.2 Remote procedure calls

Local procedure calls (function calls) are used to transfer data and control within a program. Remote procedure calls or RPCs extend this across processes. These processes may be even running on different machines. They allow remote services to be called as procedures. RPCs transparently handle location, implementation and language details to give flexibility to programmer. The goal here is to make distributed computing look like centralized computing. Communication in RPC typically happens through parameter passing and by obtaining return values. RPCs appear exactly like local procedure calls to the client program. A blocking RPC call looks like as shown in figure (when contrasted with a routine program the client resembles a caller and the sever resembles a callee. The caller waits until the callee returns with a value.)

- The client needs to know which methods are remote and which are global. And also where exactly are these global procedures are present. The first is dealt with using a interface specification file and the second uses a directory service

- Distributed systems sometimes simply expose APIs and these are also a form of RPC's (just that here RPC in HTTP form instead of a native specific RPC)

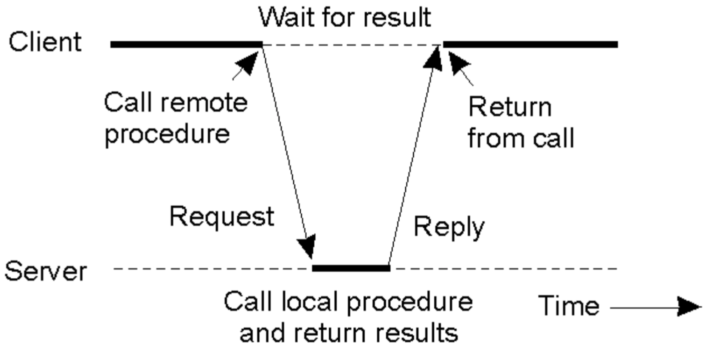

Figure 8.3: RPC as a blocking call

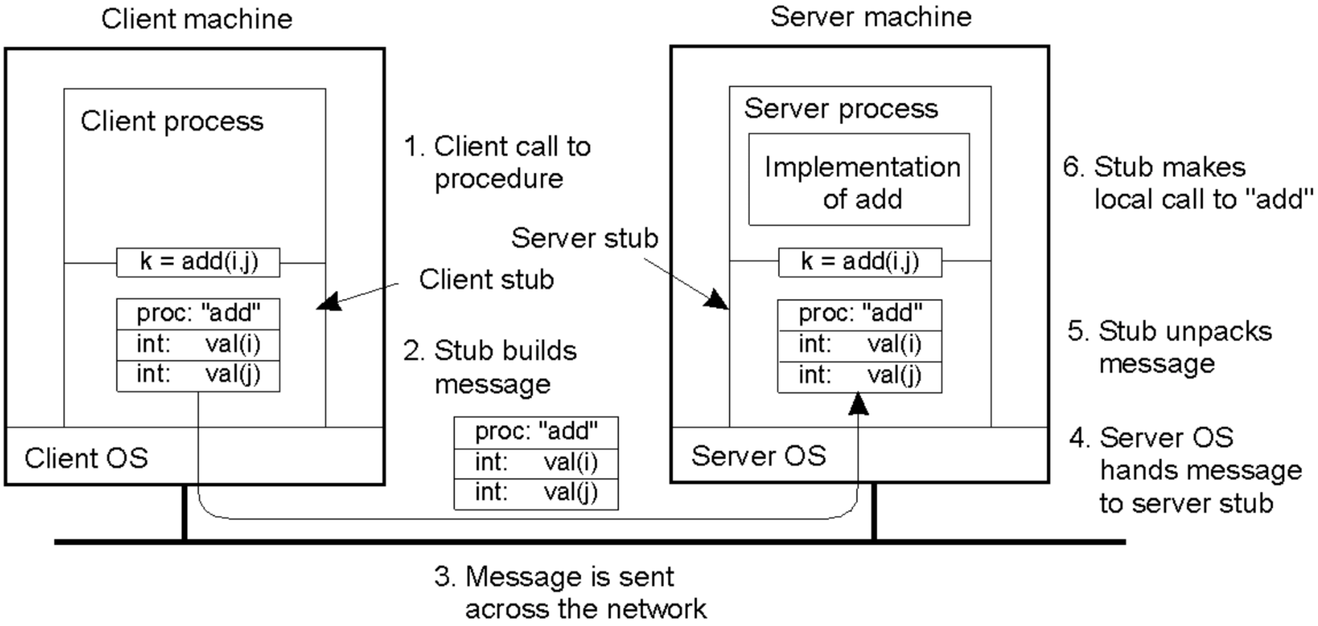

Figure 8.4: RPC - an example

8.2.1 Parameter passing

In local procedure calls, parameter passing happens either through call by value or call by reference and they use memory stack/pointers to achieve this. Global variables also have to be taken care of. Remote procedure calls use stubs and marshalling to make parameter passing look transparent (although typically, not all functionality is achieved).

8.2.2 Client and Server stubs

Remote procedure calls are handled transparently at the client and server sides using stubs. They remove burden of explicitly programming remote procedure calls from the programmer. At the client side, the client makes a local procedure call to the client stub. Stub takes care of all parameter packing and sending them as messages. This process of packing arguments and sending messages is called marshalling. This stub code is automatically generated by a stub compiler using an interface definition language. At the server side, the server stub, automatically unmarshalls/unpacks the parameters and calls the relevant procedure. Stubs on both the sides are generated automatically by the RPC run-time system.

The steps involved in a RPC call is summarized below:
1. Client procedure calls client stub just like a local procedure call
2. Client stub prepares a message and calls the local OS to deliver it
3. Client OS sends message to the remote OS
4. Remote OS delivers the message to server stub
5. Remote stub unmarshalls parameters and calls server procedure
6. Server procedure does computation and returns result back to server stub
7. Server stub packs result in a message, calls server OS to deliver it back
8. Server OS responds back to client OS
9. Client OS gets message and delivers it to the client stub
10. Client stub unpacks result and returns it to client

8.2.3 Marshalling

Marshalling is a process during which one process packs all the arguments and information in a format understood by the other process. Marshalling is necessary because different architectures have different data representation formats. For example, Intel architectures use little endian representation while SPARC uses big endian representation. In order to make RPC operate seamlessly across architectures, marshalling is required. Therefore, by using a standard external data representation (XDR), it is possible to have a process on an Intel machine make an RPC call to a process on a SPARC machine. A related problem is that, if the data we're sending contains pointers, the remote machine cannot resolve the pointer without accessing the memory of client. Possible solutions to this problem include prohibiting use of pointers or resolving pointers over network when required (The latter is complicated and typically RPCs don't use pointers as arguments).

8.2.4 Binding

Binding allows clients to easily locate servers and to query the services exposed by servers. A server can export its interface to a binding server (directory service as mentioned earlier) during initialization. This involves sending name, version, UID and handle or address to the binding server. The client then first sends a message to the binding server to import the server interface. The client can then use the handle to communicate with the server. The directory service keeps track of what services are running and where the functions are exposed. Binder can also do load balancing by if a particular server is overloaded and the same interface is exported by another server. Binding is usually done at runtime.
Related issues are:
1. Exporting and importing interfaces adds more overhead
2. A single binding server can be a bottleneck; need replicated binders if necessary

8.2.5 Failure semantics

Failures need to be appropriately dealt with. What if a client is blocked, waiting for a server and server dies? What if network is bad? (To deal with bad network we can use re-transmission techniques but must be careful and make sure that the requests are idempotent in nature. Think of a banking server example and client re-transmits multiple ”deposit $ in my account” requests). We now classify the types of errors a distributed system using RPCs can experience:

- **Client unable to locate server** : Return error to the client.
- **Lost request messages** : Use timeout mechanisms. This can be detected if no response is obtained after a certain amount of time.
- **Lost reply messages** : Use timeout to detect. Make operation idempotent - triggering the operation multiple times will yield the same result. Alternatively, sequence numbers can be used and retransmissions can be explicitly marked to let the server know.
- **Server failures** : Did the server fail before or after executing the operation?
  At least once: The operation is executed at least once. Operation is retried until a response is received.
  At most once: The operation is either executed exactly once, or not executed at all.
  Exactly once: The desirable choice ideally but difficult in practice.
- **Client failures** : When server is executing an RPC and client dies, it is called as an orphan RPC
  Extermination: In extermination, client logs requests at the client stub (i.e., detect client crashes) and explicitly kill orphans.
  Reincarnation: When client reboots, it sends a new epoch number to the server. The server then deletes computations of old epoch. A variation, called Gentle Reincarnation tries to locate the owner first and delete computations only if no owner was found.
  Expiration: Each RPC is given a fixed quantum T. Computation is aborted after T. Clients have to explicitly request extensions for computations taking longer than T.

8.2.6 Implementation issues

- Choice of protocol: Choice of protocol greatly affects the choice of communication. This decision depend on packet size restrictions of the available protocol, whether flow control is required, reliability vs speed tradeoff, etc. If TCP is used, transmission losses needn't be taken care of RPC designer.
- Overhead of copying: Message has to be copied multiple times - this adds to overhead. At least 2 copies are required. This can go up to 7 copies per message (Eg- Stack in stub - Message Buffer in Stub - kernel - NIC's buffer - physical medium - Server's NIC - Server kernel - Server stub. 7 copies). Although, copying mechanisms like scatter-gather operations (which pass pointers within a system as much as possible) do exist which reduce overhead to an extent.

8.2.7 Case study: SUNRPC

SUNRPC is one of the most widely used RPC mechanism. Either TCP or UDP can be used as the underlying protocol. During marshalling, multiple arguments are packed into a single structure. It uses SUN's external data representation (XDR) to provide compatibility across different architectures. This internally uses big endian ordering. SUNRPC is used by NFS (network file system).

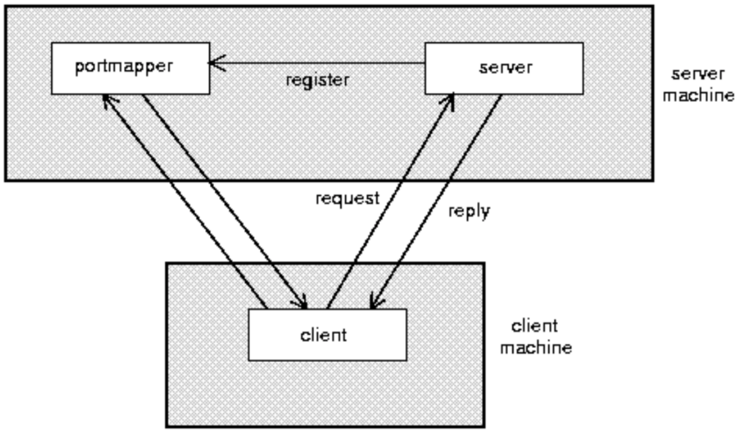

Figure 8.5: Port mapper

SUNRPC's binder is called port mapper. When a server starts up, it creates a port and registers its service with the local port mapper (using svc_register). On client start up, it locates server port using (clnt_create) and then can use services offered by the server.

SUN's RPC package also includes an RPC compiler that automatically generates stubs for client and server. The programmer has to write client code, server code and a .x file (this interface specification file contains interfaces of all RPC messages). Then RPC compiler creates four files: server stub, client stub, a header file and the XDR code. The client application + client stub + header can then be sent to C compiler creating the client binary and the server application + server stub + XDR together create the server binary. When both these binaries are run, client-server communication is possible with RPCs.