Nachos Networking Background

1 The Network

The network is like a sequence of wires which transport packets among a small group of machines. Some machines sit on more than one of these wires, these machines are called gateways. To send a message to a machine, you need to decide how to get to that machine. It would be difficult for each machine to know how to get to any other machine it might want to talk to, since the network is constantly changing due to new networks being added, old networks breaking and becoming unavailable for transport for a while, and networks being replaced with different ones to meet changing needs.

It would be impractical for every machine on the network to keep up to date versions of this information about every network that it could possibly reach. Instead what is done is each machine only needs to know how to make the first step towards the destination. Usually this means that each machine knows how to get to the gateway which will put the packet on the right track towards the destination. The gateway knows how to get to the next gateway, and so on until the packet reaches the final gateway, and this machine sends it directly to the destination.

In Nachos, we will have a much simpler system. There will only be a small number of machines, and each machine has a direct link to every other machine, so there is no need for this kind of routing. To send a packet to a machine, specify which machine it is to go to and the kernel takes care of sending it to that machine.

This leaves us with the problem of identifying who on that machine is to receive the packet. This is a naming problem: how do we name something on a destination machine. While machine objects several unique identifiers (for instance, in Unix a process has a unique process identification), these names change over time. We want a more global name, so that we can send the message without inspecting the foreign name space.

The model we use is similar to the postal system. Lets say you want to send a letter to your friend Joe in New Jersey. Well, you can't just say *Joe in New Jersey* on the letter, since the post office doesn't know how to deliver that. Even if there were only one Joe in all of New Jersey, he might move around some (drive to work, go to a movie, play in some toxic waste) so the post office can't deliver the message directly to him. Rather what you do is specify some well known drop point, say a post office box. To send the letter to Joe you say *Post Office Box 10202, Joisey City, NJ*, and the post office moves the message to that box. Some time later, when convenient, Joe comes to the post office building and looks in his box for new letters.

Keep in mind that the only reason you know that sending a letter to a given PO Box (or address) will get to a specific individual is that you know that the person will look for messages in that place. There is no inherent connection between an address or a PO Box and the person who gets mail at that location in the postal system. If you were to give the correct name but the wrong PO box number, the post office would stick the message in the wrong box and Joe would never get the message.

Interprocess communication in Nachos works in a similar fashion. Each machine provides a number of message holders (called boxes in Nachos and sockets in Unix), which can be the destination of a message. These slots are identified by a number. When a process wants to get messages from other processes, it publishes which box number it will receive messages in, and waits for the messages to come there.

This is very much like how Unix works. For Unix there is a set of standard services, which do things like get the mail, report who is on the machine, do file transfer and remote login stuff, and so forth. Each of these is a process which accepts messages from other machines and does some kind of work. The problem is, how do you know where to send the message to get in touch with the service you are interested in. In Unix, there is a text file called `/etc/services` which is a list of the programs and the location at which they can be reached. This is how they publish their mail box number (kind of like a phone book).

Keep in mind that this just specifies an initial contact point. If a couple of processes want to initiate an extended communication session, then they will probably want to get their own pair of mail boxes to communicate through, so someone who wants to initiate a conversation won't be sending messages into the same box as they are using to carry out their conversation.

Remote login could work something like this. There is a remote login server which sits on, say, mail box 513. When someone tries to rlogin to the machine, this person sends a message to this box saying which box he is listening to and what he wants. The rlogin process forks itself and the child goes off to service this request, and the parent continues listening to the published box for more requests.

The child looks at the intro packet and determines what to do. The intro packet contains the box number on the remote machine, and the remote machine name, to talk to. The child doesn't want to continue using the published box for communication with the client since other people will be sending messages to that box to start other rlogin sessions. Instead it gets a new box that is unused and tells the client to send further communication to the new box. This is kind of like a business starting a new department. They will want to give the new department its own phone number and mail box, rather than having all communication come to one address where they will have to sort it according to whether it is for the old or the new business.

In this situation we have three boxes in use. One is the box the client is using to get messages from the rlogin server. One is the published communications end point that the rlogin server listens to for new connections. The third is a private box used on a per communications bases by the client and the server on the server machine, which is the destination of all of the messages from the client to the server except the first.

2 Transferring Messages

A letter that you might send a friend contains 3 pieces of information. There is the address it is being sent to, the return address, and the letter itself which is just a bunch of information. By convention, the destination and the return address appear on the outside of the letter, so the post office workers can find it easily. The only service they provide is moving the letter to its destination (if it has a stamp). There is also a limitation on the weight of a letter, so you couldn't send the OED [1] to your friend in Boston with one 29 cent stamp.

In the Nachos network, a packet will contain the source machine and box number, the destination machine and box number, and some number of bytes of packet data. The network itself just moves the packet, all of the other structure is convention. These conventions on the information content and structure of this information in a packet are called protocols.

2.1 Protocols

There are many many protocols in use today. Common services provided by a protocol are routing and reliability. A route is a sequence of machines (or gateways) which are used to move a packet from the source to the destination. A simple routing protocol might only contain the destination of the packet, and the machines along the way would make the decisions about how to get it there. Other routing protocols might contain information about how fast the packet should be moved and how important it is that it not be lost. These can affect the decisions about which network path to use.

Another service typically provided by protocols is reliability. When a packet is sent from a machine, it is gone. The network promises the machine that it will attempt to deliver the packet uncorrupted to the destination, but it is making no guarentees. If it is important that the packet arrive, then the protocol must provide some mechanism for doing this.

Reliability has two aspects. The first is that the packet arrive at all. Protocols which implement this usually do so by having the destination machine send a message back to the source when the packet arrives, acknowledging that it got the packet. The source machine sends the message and waits a given amount of time for the *ack* (acknowledgement packet), and if it doesn't arrive in that time, retransmits the message.

[1] Oxford English Dictionary

For this to work, each message needs to have a unique identification, so the destination machine can tell the source which message arrived (there may be several in transit at any time). Usually a packet is identified by the source machine id number and a serial number, unique to the source machine.

When machine *danube* sends a packet to another machine, it generates a serial number (by incrementing the number of the last packet it sent), places this number in a field in the packet header, and sends it. It waits to hear from the destination that this packet has been received before deleting the memory buffer used to hold the message. If it doesn't hear before the timeout it retransmits. After some number of retransmits it decides that for some reason it is not possible at this time to communicate with that machine and reports an error.

The other aspect of reliability is corruption detection. The bits in a packet can sometimes be changed when they are transmitted over the network. There are various error detecting and correcting procedures which can be used to deal with this. The easiest thing to do is compute the parity of all of the bits in the packet (that is, take the sum of the bits and mod the result by two). This is not that good because, while it will detect when one bit has been changed, it will not notice if two bits were changed.

The check sum is a kind of a strengthening of the parity check idea. For each packet the source machine runs a procedure which computes a fairly large number (say 32 bits) according to the bits in the packet. One possibility is to break the packet into 4 byte words, treat these words as numbers, and add the numbers together ignoring overflow. This checksum is added to the packet. The receiving end computes the same function on the packet (minus the checksum field of course) and compares the two numbers. If they are different it reports a corrupted packet transmission to the source.

There are more elaborate schemes, called error correcting codes, which add enough information to every byte of the message to correct one changed bit and to detect when one or two bits in the byte have been changed. Such things are called error correcting codes.

A final service which simple protocols might provide is fragmentation. Sometimes the network hardware won't allow the transmission of packets larger than a certain size. Often it is not the first network, but some intermediate network between the source and the destination, which has the limitation on size. Since the user doesn't know, and doesn't want to know, about all of the networks between her and the destination machine, the protocol should hide this limitation transparently. In other words, the software should make it seem like you can transmit any size packet, and if the size exceeds the maximum size the hardware supports, the software should manage transmitting the packet without the users intervention.

The protocol, when given a large packet, breaks it up into small enough packets and transmits them separately. This is called fragmentation. The fragments themselves may be fragmented again by another network, so the whole packet may arrive in many pieces of varying sizes. The receiving end is responsible for collecting all of the fragments, putting them in the correct order (the order of receipt is not necessarily the order of transmission) and presents the destination process with the large message as it was transmitted.

3 Models of Communication

We have considered how to implement basic message communication on real hardware. The next issue to consider is how does the communications client interact with the communications facility. The client in this case could be either a user program or the kernel itself. In general the kernel may provide several services and these services may in turn be clients of each other: for example the virtual memory system in Nachos is a client of the file system, and you may extend it to page over the network, which would make the virtual memory system a client of the network.

One possibility is to present the network communications as they really are, a packet transporting service. The client would be allowed to `send` and `receive` packets to and from the network. These calls do just what it sounds like they do, put a packet on the wire, and wait around for a packet to come off of the wire, respectively. The problem with these is that this is not like any other thing that programmers do, so it is not always so easy to design and implement communicating programs with these primatives.

Another possible model is to treat the communications facility like a file, using read to get a packet from the network (and wait for one if one is not available), and write to send a packet. Of course there is no natural way to talk about packet lengths and boundaries in read, so this mechanism typically loses information about when one packet begins and another ends. On the other hand, it is a simple and natural way for programmers to think about the problem.

One kind of message passing that programmers do all the time is procedure call. Calling a procedure is like passing the arguments in a message to the procedure and waiting for the procedure to send back the result, a.k.a. the return value, in another message. This is a popular form of interprocess communication which seems to be just the right model for many problems, and is something that people can work with comfortably. In fact, most systems make system calls, another form of interprocess communication, look like procedure calls for the same reason. Some systems also make remote procedure calls look exactly like procedure calls, so that the programmer might not know whether the procedure is local or remote.

3.1 Remote Procedure Call

Remote procedure call is abstractly like a regular procedure call. In implementation, they are very much like system calls.

A system call is a request to the kernel to do some work for you. Since the kernel lives in a different address space, you can't do this with a procedure call. You need to make a request message, which tells the kernel what work you want done, for instance open a file, and the arguments, in this case the name of the file. You send this request to the kernel with the `syscall` instruction.

This is very similar to a remote procedure call. In an RPC you need to create a package which can be sent to another machine (or even the same machine) and contains enough information to run the procedure call. This information includes the procedure to run and the arguments.

There is one important difference. The kernel has access to the address space of the calling process in a system call, so it is possible to pass pointers in a system call. For an RPC this won't work, since the address space lives in a separate physical memory from the process running the call. So, when sending arguments through RPC all of the information the procedure needs to run must be in the packet. If a string is to be passed, then the string itself must be present, not just a pointer to the string.

Often procedures share information through global variables (for instance global data structures) or kernel resources (open files, file location). This kind of sharing doesn't work with RPC, since the global space and kernel of the caller are separate from those of the callee.

Side effects also generally don't carry over between machines. A procedure to draw a picture on the screen might not be a good candidate for RPC-hood, since the display is a global resource on the machine that the procedure runs on. You don't want to draw the picture on the wrong machine.

Many window systems, such as X11, allow windowing applications to run on machines other than the one the display lives on, by sending all drawing requests off to the machine with the display. This is essentially using an RPC mechanism to give machines access to a machine local resource, the screen that the user is looking at, which lives on another machine.

Procedures you might want to run remotely would be things like computation intensive routines (which could be run on the local cray) or procedures which access resources not on your local machine (like remote file system access, remote data base access).

An RPC mechanism has a client side and a server side, analogous to the user code and the kernel code in the system call mechanism. The client side code packages the request up and sends it to the server, waits for the reply and unpackages the result. The server side code waits for requests, unpackages the arguments, runs the procedure, and packages and sends the result back to the caller.

RPC is typically implemented on top of unreliable hardware. Network problems include lost packets, corrupted packets, and reordering of packets. The first problem can be solved by giving each packet a unique number (serial number) and having the receiver send back an acknowledgement packet with this serial number, each time it correctly receives a packet. The serial number can also be used to solve the reordering problem, if the packets on each machine are given numbers in increasing order according to when they are sent. This also helps with the lost packet problem, since the receiving machine knows that a packet was lost or reordered as soon as an incoming packet doesn't have the next serial number.

The calling side of an RPC needs to know how to get in touch with the procedure to run on the server side. Two obvious ways of doing this come to mind. The first is to have a universally agreed upon RPC mail box number, which always gets messages for the RPC server. This server then figures out which procedure to run by inspecting the packet, and fires it up. This is like the system call implementation, in which the system call number always lived in the same register and inside the kernel there was a procedure which inspected this register and generated the call.

The other option would be to have each procedure which could be called remotely publish which mail box number it would listen to. To call this procedure the client would send a packet with the arguments to the correct mail box.

The second method is probably better in that it would be easier to debug and saves the demultiplexing step that the first one has to do on packet receipt. On the other hand it could potentially use up a large number of mail box slots, and if these are a limited resource (currently only 16 bits are available for box numbers on Unix), it would not be a good solution.

3.2 Byte Streams

The other common model of interprocess communication is byte streams. In this model there is a (one way or two way) connection between two processes, sometimes called a virtual circuit. When one side puts data into this connection, the data appears in the same order on the other side. There is no structure on the data. This is like a Unix pipe. [2]

The socket communications facility in Unix uses this model. When you create a socket, before you can start sending data on it you need to connect it to another socket, which will be the destination of that data. Then whenever you put data into the socket using `write`, the process on the other end can get the data by executing a `read` on its socket.

Stream communication is connection based. This means that the two endpoints of the communication facility always send their data to one another. You will need to implement a method for making the initial connection, and for sending data once the connection is made.

In Unix, when a process is ready to start a stream communication session, it announces this by `accept`ing connections to a mail box and `listen`ing for activity. When a process wants to communicate with another process which is

[2] Pipes are exclusively unidirectional. There is a related object in Unix called a socketpair. See "man 2 pipe" and "man 2 socketpair" for more information.

accepting connections, it `connect`s to it. Once the connection has been made the two processes communicate by `read`ing and `write`ing on the connected sockets.

The Unix man pages on `connect`, `accept`, `listen` and `bind` describe the function of these procedures in the Unix environment. You will be specifically interested in `SOCK STREAM` type sockets. You can combine the function of several of these (for instance bind, listen and accept) in your implementation.

The `rlogin` program is a good example of an application in which streams are the correct communication model. On the client side of rlogin the program takes the user key events and sends them off to the other end. The client also takes the data out of the socket whenever some appears and puts it on the users screen. The server end gets data from the socket and sends it to the program which is being run currently [3] on the server end (this is initially the shell), and takes the output of that program and sticks it into the socket.

[3] This is done by pretending this data was typed into a terminal. Since the rlogin session isn't running on a real terminal at the remote end, the kernel provides a bunch of pseudo terminals, `ptys`, which act like terminals in that they send data to a program through interrupts and allow terminal editing, like back space and kill line.