Nachos Assignment #5: Networking

Tom Anderson
Computer Science 162
Due date: Tuesday December 7, 5:00 p.m.

The final assignment will be a little more open ended than the others, calling for more experimentation and creativity on your part. We will provide you with some low level network communications facilities; you will build a nicer abstraction on top of those, and then use that abstraction in building a distributed application.

Each separate Nachos (emulated as a UNIX process) is a node in the network; a network (emulated via UNIX sockets) provides the communication medium between these nodes. You can test out the basic network functionality by running 'nachos -m 0 -o 1' and 'nachos -m 1 -o 0' simultaneously (preferably, on different terminals or in different windows). [Note: this test case requires a working implementation of locks and condition variables from assignment 1, but nothing else.]

There are only a few files for this assignment:

nettest.cc – network test routines.

post.h, post.cc – a post office abstraction, built in software on top of the network. This provides synchronized delivery and receipt of messages to/from specific mailboxes; there may be multiple mailboxes per machine.

network.h, network.cc – emulation of the physical network hardware. The network interface is similar to that of the console, except that the transmission unit is a packet rather than a character. The network provides ordered, unreliable transmission of limited size packets between nodes. All routing issues (how the message gets from node to node) are taken care of by the network.

The post office provides a more convenient abstraction than the raw network; you are to continue this layering process – at each level, the software removes one physical constraint and replaces it with an abstraction. Thus, reliable messages can be built on top of an unreliable service, large messages can be built on top of fixed-length messages, etc.

A separate document (appended) goes into more detail on network issues.

1. Implement reliable messages with no size limits. Currently the communications facility provides unreliable transmission of limited size packets. You must implement protocols to fix this. It is up to you how to implement

these protocols; for instance, you may build your support on top of the post office or underneath, on top of the raw network device.

Warning: you are strongly advised to do a very careful paper design of your protocol before starting to implement it. It is very easy to design protocols (particularly for reliability) that do not work, and it is typically very difficult to find and fix protocol bugs after the protocol has been implemented.

The Nachos network emulation can be made to randomly drop packets by using the command line argument '-n #'; the number, between 0 and 1, reflects the likelihood that a packet will be successfully delivered. To simplify matters, you may assume that packet delivery is "fail-safe"; packets may be dropped, but if a packet is delivered, its contents have not been corrupted. (In practice, a hardware or software checksum would be needed to detect this kind of error).

2. Implement a distributed application. We leave the choice up to you. Use your imagination! Here are some possibilities to consider (since some of these are obviously more work than others, we will offer up to 10% extra credit for solutions implementing any of c - f):

   a. Support multi-user talk (cf. UNIX talk), between more than two machines.

   b. Build a network gateway protocol.

      Currently, we assume that the network hardware allows each Nachos machine to directly send messages to every other Nachos machine, but this is unrealistic, even in a local area network. Implement a protocol that would allow Nachos machines to send messages to each other in a point-to-point network. Make your protocol robust, so that it will work even as machines are added or removed from the network.

   c. Build a caching network file system.

      The network file system should be transparent. If you do this carefully then you should be able to page over the network, or execute a program which lives on a remote disk, without changing any of the virtual memory code. Similarly `cat` should work on remote files without even a recompilation. In fact, you should be able to run a nachos machine which gets all of its disk needs met by a remote machine (a diskless client).

   d. Build a network-wide shared memory abstraction.

      With this, two user programs running on different machines can read or write a shared-memory region. Thus, you could have a parallel program that uses a network of workstations as a multiprocessor.

e. Create a process migration system.

   Process migration means moving an executing process from one machine to another machine. This is different from remote execution, which means starting a process on a remote machine. For process migration, you will need to copy the whole virtual address space of the migrating process to the new machine. A clever programmer might page it in over the network from the source machine, using a remote file as the backing file.

   Process migration is a delicate matter. When the process moves to a new machine, all of the open files and other kernel resources it was accessing on the initial machine are inaccessible. One way to get around this problem is catch all accesses to previously opened files on the remote machine and forward them back to the host (a remote file access). A more general solution is to send all system calls the process executes back to the original host machine. This is perhaps the safest solution, but can lead to performance problems.

f. Build a distributed game, such as tic-tac-toe or battleship, where each player is on a separate machine.