Networks and distributed computing

Hardware reality

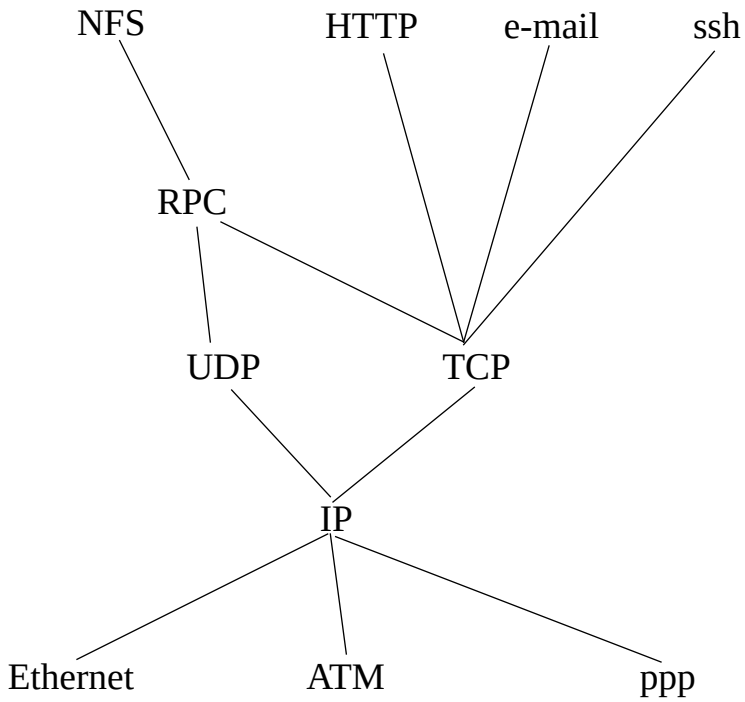
- lots of different manufacturers of NICs
- network card has a fixed MAC address, e.g. 00:01:03:1C:8A:2E
- send packet to MAC address (max size 1500 bytes)
- packets may be reordered, corrupted, dropped, duplicated
- anyone can sniff the packets from the network

What abstractions does the OS provide for network communication?

Distributed computing (not covered much in EECS 482): making multiple computers look more like a single computer

- distributed shared memory: make multiple memories look like 1 memory
- remote procedure call, process migration, parallelizing compilers: make multiple CPUs look like one CPU
- distributed file systems: make disks on multiple computers look like one file system

Abstractions and protocol layers



Why build up abstractions in layers?

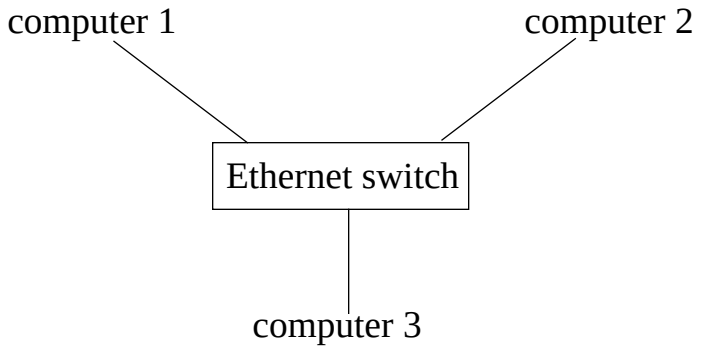
Routing

Hardware interface: deliver to neighbor computer on LAN

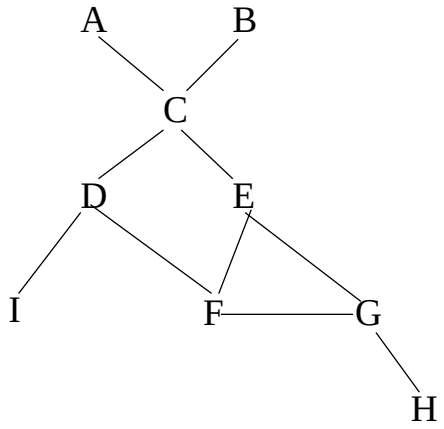
Application interface: deliver to final destination through several hops

Provided by the IP (Internet Protocol) layer

Messages on LAN (e.g. Ethernet) are sent via the physical ID of the network interface card (e.g. 0:a0:c9:95:f5:58)



Internet is composed of lots of connected LANs



How does computer on Ethernet D know how to get to Ethernet H? Should it send to C, F, or I?

This is hard because Internet has no centralized state

Routing is difficult for large systems, and for systems that change rapidly. Internet is both large and dynamic.

Basic idea: routers propagate information to each other and hope the global picture converges before its out of date (take EECS 489 for more details)

Try running "traceroute www.usatoday.com"

Symbolic naming

Low-level interface: destination specified by MAC address, e.g. 00:01:03:1C:8A:2E

Middle-level interface: destination specified by IP address, e.g. 141.213.8.32

High-level interface: name destination by hostname (e.g. life.eecs.umich.edu)

Translation from IP address to MAC address is provided by ARP protocol (address resolution protocol)

How does computer 1 know computer 2's Ethernet (MAC) address?

- start with the IP address for the computer (e.g. 141.213.8.32)
- use ARP (Address Resolution Protocol) to get the mapping from IP address to MAC address
- to find out mapping from IP address to MAC address (e.g. 141.213.8.32 maps to 0:a0:c9:95:f5:58), ARP broadcasts to all computers on the LAN
- cache the mapping for next time

Use broadcast when all else fails (but this doesn't scale well)

Translation from hostname to IP address is provided by DNS (domain name system)

Used to be done with one central server
- central server has to learn about all changes
- central server has to answer all lookups

Split up the data into a hierarchical database (each DNS server stores part of the database). Hierarchy allows local management (so everybody doesn't notify one central server whenever their hostname changes), and spreads the lookup work across multiple servers

Example: translating www.eecs.umich.edu
- start with the (well-known) IP address of the root name server (A.ROOT-SERVERS.NET, 198.41.0.4)
- ask root name server for IP address of the edu name server (also A.ROOT-SERVERS.NET, 198.41.0.4)
- ask edu name server for the IP address of the umich.edu name server (dns.itd.umich.edu, 141.211.144.15)
- ask umich.edu name server for the IP address of eecs.umich.edu name server (zip.eecs.umich.edu, 141.213.4.4)
- ask eecs.umich.edu name server for the IP address of www.eecs.umich.edu: 141.213.4.18

Message size

Hardware interface: physical network type limits size of a message (e.g. Ethernet maximum packet size is 1500 bytes)

Application interface: can send larger message (e.g. IP maximum packet size is 64 KB)

IP layer can fragment a packet when it's larger than the next hop's MTU (maximum transmission unit), then re-assemble it at the destination

Sockets and ports

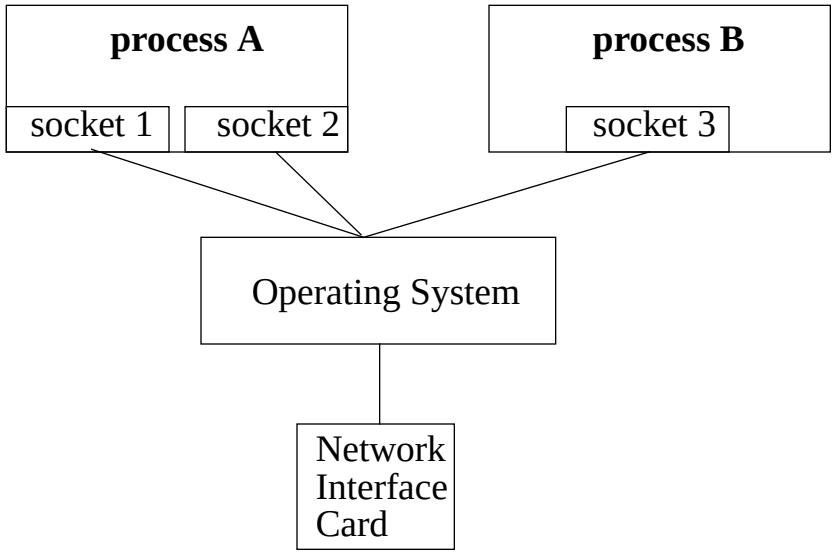
Hardware interface: machine-to-machine communication (one network endpoint per machine)

Application interface: process-to-process communication (one or more network endpoints per process)

A process can ask the OS to create a "socket", which will be one endpoint of a network connection
- thread is like a virtual processor
- address space is like a virtual memory
- an endpoint (socket) is like a virtual network interface card

Each socket on a computer has a unique "port" number
- a process can associate a specific port number with a socket using the `bind` call
- when sending to a socket, the destination port number is included in each message. This allows the destination machine to know which process (and which socket in that process) should receive the message.

The OS to multiplex several network connections onto a single physical card



UDP (user datagram protocol) provides this process-to-process abstraction on top of IP

TCP (transmission control protocol) is also built on IP
- provides additional abstractions beyond UDP: ordered, reliable, byte streams

Ordered messages

Hardware interface: networks can re-order messages that IP layer sends out
- e.g. Send: A, B. Arrive: B, A

Application interface: all messages are received in the order in which they were sent

How to provide ordered messages?

To have a notion of order, we must first have the notion of a network "connection" (so we know that multiple messages are related)
- with TCP, process opens a connection (using `connect`), then sends a sequence of messages, then closes the connection
- sequence # is specific to a socket-to-socket connection

Reliable messages

Hardware interface: networks can drop, duplicate, or corrupt messages

Application interface: each message is delivered exactly once (without corruption)

How to fix a dropped message?

How does sender know a message has been dropped?

Duplicate messages are easy to detect (look at the sequence #) and fix (just drop the duplicate)

To detect corrupted messages, add some redundant information, e.g. checksum

- if message is corrupted, simply drop it. This transforms the problem of a corrupted message into the problem of a dropped message, and we already know how to handle that).

Byte streams

Hardware interface: send information over network in distinct messages

Application interface: send data in a continuous stream (similar to reading/writing a file)

TCP provides byte streams instead of distinct messages

Sender sends messages of arbitrary size that are combined into a single stream

TCP layer breaks up the stream into fragments, sends them as distinct messages, then reassembles them at the destination into a byte stream for the receiver. In contrast, UDP preserves the message boundary between sender and receiver.

E.g.

- sender sends 100 bytes, then sends another 100 bytes
- TCP receive may return 1-200 bytes

If receiver wants to receive a certain number of bytes, it must loop around the receive call

How to know # of bytes to receive?

Why build distributed applications?

Performance: aggregate performance of many machines can be higher than the performance of a single fast machine

Co-location: locate different computers near local resources
- examples of local resources: people, sensors, actuators

Reliability: can provide continuous service, even if one computer is down

Building distributed applications

Send/receive as communication primitive
- how did we communicate between threads running on a single computer?
- this doesn't work for threads running across different computers (distributed applications)
- to communicate, must send/receive messages

Send/receive as synchronization primitives
- what hardware primitives did we build on top of to synchronize between threads on a single machine?
- these don't work for synchronizing between multiple machines
- we'll use send/receive as the atomic primitives that allow us to synchronize distributed applications

Send and receive are atomic

- two sends to the same machine don't get intermingled (the messages stay distinct)
- a process calling receive can't interrupt the act of receiving from the network—can't receive just a portion of a message (all or nothing)

What is really enforcing the atomic action for send/receive?

Client-server

Lots of different ways for multiple computers to cooperate on a single distributed application. One of the most common is **client-server**.

File server example

- server stores the files (e.g. AFS server)
- clients are computers that run the applications and ask for file data from the file servers.
- when client wants to read a file, it sends a request to the server, then waits for a response (which includes the file data)
- when client wants to write a file, it sends request to the file server (which includes the new data for the file), then waits for a response that acknowledges the successful write operation

Producer-consumer with client-server

- server manages state associated with coke machine
- clients computers can call client_produce() or client_consume(), which sends a request to the server and returns when the request is done
- client_produce() blocks at the server if the coke machine is full; client_consume() blocks at the server if the coke machine is empty

```
client_produce() {
   send produce request message to server
   wait for response
   }

client_consume() {
   send consume message to server
   wait for response
   }

server() {
   receive request (from any producer or any
      client)
   if (request is from a producer) {
      put coke in machine
   } else {
      take coke out of machine
   }
   send response
}
```

Problems with this code?

How to fix the code?

```
server() {
   receive request (from any producer or any
      client)
   if (request is from a producer) {
      fork a thread that calls server_produce()
   } else {
      fork a thread that calls server_consume()
   }
}

server_produce() {
   lock
   while (machine is full) {
      wait
   }
   put coke in machine
   send response to producer client
   unlock
}

server_consume() {
   lock
   while (machine is empty) {
      wait
   }
   take coke out of machine
   send response to consumer client
   unlock
}
```

This creates a new thread for each request. How to lower the overhead of creating threads?

There are other ways of solving the problem (but threads are the cleanest, because each thread just has to keep track of one thing at a time (and it can be blocking, as long as it doesn't hold a lock)

- polling (using select())
- signals (using SIGIO)

RPC

We've been using send/receive. E.g. client sends request to server, server receives request, then sends response message to client

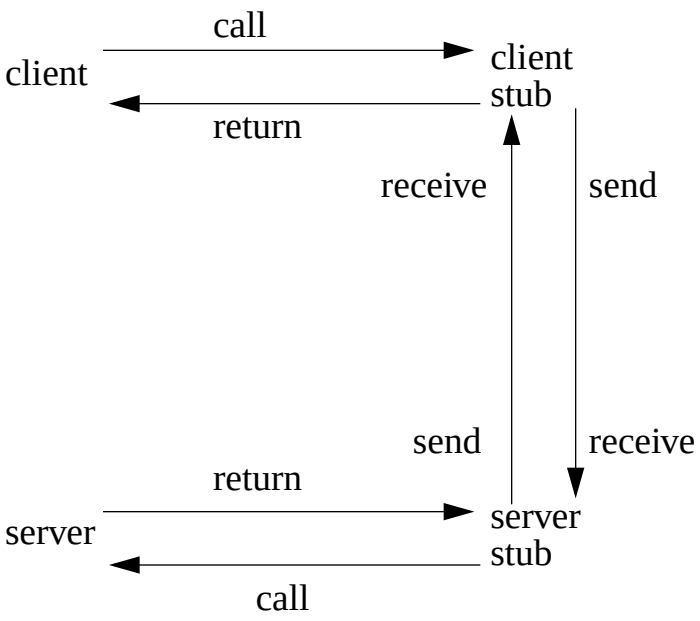
- this exposes the distributed nature of the system to the programmer
- we'd like to make building a distributed application as similar as possible to building a centralized application

What else in programming is like making a request to a server and getting a response?

How to make a message send/receive look like a function call to both the client and server?

- client wants sending of the request to the server to look like calling a function, and wants the reply from the server to look like returning from the function call
- server wants to receive a request from the client in the same way as getting called by a function, and wants to send the response to the client just like returning to the caller

Stub functions on client and server can provide this abstraction



Client stub

Server stub

Note that client makes a normal function call, server function is called like a normal function

RPC is the mechanism behind CORBA and COM

Producer consumer using RPC

This uses datagrams (like UDP) and assumes messages are reliable

Client stub is named produce()

```
int produce(int n){
   int status;
   send(sock, &n, sizeof(n));
   recv(sock, &status, sizeof(status));
   return(status);
}
```

Server stub can be named anything

```
produce_stub() {
   int n;
   int status;
   recv(sock, &n, sizeof(n));
   status = produce(n); /* call “produce”
                           function on server */
   send(sock, &status, sizeof(status));
}
```

Client and server stubs can be generated automatically. What information do you need to generate the stub?

Problems with RPC

RPC tries to make request/response to remote server look like a function call, but some differences remain

Hard to pass pointers and global variables

- what happens if you pass a pointer to the remote server and the server de-references it?

- one way to fix this is to also send the data being pointed to, then change the pointers on the server to point to the remote copy of the data, then copy the data back to the client when the server is finished

Data might have different representations on different machines

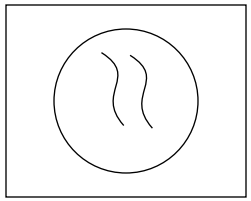
- solve by agreeing to some conventional format

Different failure modes can occur in RPC than a normal function call

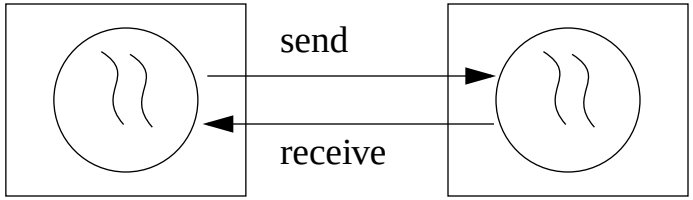
- e.g. server fails but client stays up

Structuring a concurrent system

1 multi-threaded process on 1 computer

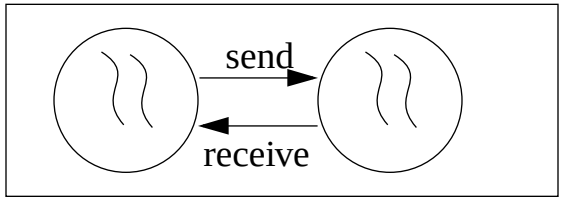


1 multi-threaded process on each of several computers



Can divide cooperating threads on a single computer into separate processes (i.e. different address spaces), then use messages to communicate between these processes instead of shared memory

several multi-threaded processes on 1 computer



Why might you do this?

One operating system design technique (microkernels) separates out the kernel into different server processes (each in its own address space), then uses messages to communicate

- this way, one part of the kernel doesn't have to trust the others to not trash its memory