Networking

Networking and Operating Systems
Kinds of Networking
Applications
The Stack
Protocol Suites
Connection-Oriented or Connectionless?
Mux/Demux
Layers
Application Access
Example: tcpdump
at Link Layer
Example: DHCP
Example: OSPF
Example: ping
Interfaces
Applications
Middleware
The Global Grid

Networking
Networking and Operating Systems

- Many different pieces
- Some pieces are in the kernel; others are in user space
- Apart from the division, we need appropriate interfaces
Kinds of Networking

- Different protocols; different protocol suites
- User versus kernel consumption
- Synchronous versus asynchronous
Applications

- Service
- Servers
- Clients
- Peer-to-peer
- Which are part of the OS?
The Stack

- Seven layers: physical, link, network, transport, session, presentation, application
- Well, not really, on the Internet
- Link layer — device drivers
- Network: IP (Internet Protocol)
- Transport (and a bit of session): TCP, UDP
- Presentation, application: applications
- All but the last are in the kernel
Protocol Suites

- TCP/IP — The Internet
- Subclass: IPv6
- OSI
- Novell IPX
- Appletalk
- NetBIOS
- Many others that have faded from the scene
Connection-Oriented or Connectionless?

- Some protocols are *connection-oriented* — once things are set up, you always talk to a single endpoint
  - Example: TCP
- Others are connectionless — each packet can go to or come from a different place
  - Example: UDP, as used in the DNS
Mux/Demux

- Many layers of multiplex/demultiplex
- Link layer selects different network layers (and maybe different protocol suites)
- Network layer selects different transports
- Transport layer selects different applications
- User space can read all layers
Layers

Networking
Networking and Operating Systems
Kinds of Networking
Applications
The Stack
Protocol Suites
Connection-Oriented or Connectionless?
Mux/Demux

Application
Transport
Network
Link
Application Access

- Why do applications have access to all layers?
- Debugging
- Implement layer at user level
Example: tcpdump at Link Layer

```
# tcpdump -v -v -s 1500 not ip and not arp
tcpdump: listening on bge0, link-type EN10MB (Ethernet), capture size 1500 bytes 23:05:57.642505 00:48:54:71:ce:32 > Broadcast null I (s=0,r=0,C) len=42
```
Example: DHCP

- DHCP is used to assign IP addresses to hosts
- At the time a host issues a DHCP request, it has no IP address, so it can’t speak IP
- Both the DHCP client and the DHCP server have to listen – and speak – at link level
The OSPF routing protocol runs directly on top of IP

To implement this at user level, the program has to read IP packets
Example: ping

- ping uses ICMP messages
- ICMP lives directly on top of IP
- In other words, ICMP packets have to be available both in the kernel and at user level
What is the interface to the networking stack?
Is it file-like? Something special?
The answer, of course, is “it depends”
Network Connections Aren’t Files

- We *always* do much more than just read or write
- We often have to pass extra information, such as source or destination address for connectionless protocols
- But sometimes, we do just read and write...
The Socket Interface

- Due originally to Berkeley, circa 1983
- Network access is initiated by the `socket()` system call
- On Unix, `socket()` returns an ordinary file descriptor; you can (eventually) do `read()`/`write()`/`close()`
- On Windows, it returns a special type of file descriptor; you can only do special socket operations, such as `send()`/`recv()`
Sockets and Layers

- Sockets are used at all layers — parameters on the `socket()` call specify the layer and protocol.
- The semantics of the returned file descriptor are layer-dependent.
- A variety of options can be set with `setsockopt()` and `ioctl()`.
- Sockets are also used for configuration control, such as assigning IP addresses to interfaces.
- You control the routing tables by writing to a special socket.
Connectionless Networks

- Every packet written needs a destination address as well as data
- Every packet read contains a source address as well as data
- Use `sendto()`/`recvfrom()`
- (Address format will vary, depending on the type of network; some networks even use variable-length addresses)
 Couldn't We Just Open /dev/tcp?

- On some systems, such as Solaris, you can!
- But — we still need special operations
- `setsockopt()` could just be an `ioctl`, but
- `accept()` returns a new file descriptor
- Connectionless networks use special data format
Network connections aren’t really files

Given all the special stuff that has to go on anyway, there’s little advantage to using the file system instead of sockets

There may be some advantage, though, to getting a normal file descriptor
Applications

- Service
- Servers
- Clients
- Peer-to-peer
Service Applications

- Service applications function as an extension of the OS
- Example: with NFS, the client and server are in the kernel, but other pieces are at user level: mounting and unmount the file system, locking, etc.
- Example: Routing — IP is pretty useless without it
Service Dispatchers

- Certain applications exist just to run other applications
- ineted — Runs most TCP and UDP applications
  (Original idea was to keep the process table small, to improve performance. Now, it permits (some) servers to just use stdin/stdout)
- Example: portmapper — dispatches inbound RPC requests
The Port Mapper

- Subprocedures — or rather, their stubs — register with the portmapper
- The caller’s stubs contact the port mapper to find out the actual port number for the subprocedure
- This is not quite invisible to the application programmer; at the least, the registration has to be set up
The Port Mapper

$ rpcinfo -p cluster.cs.columbia.edu

<table>
<thead>
<tr>
<th>program</th>
<th>vers</th>
<th>proto</th>
<th>port</th>
<th>service</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>4</td>
<td>tcp</td>
<td>111</td>
<td>portmapper</td>
</tr>
<tr>
<td>1000000</td>
<td>3</td>
<td>tcp</td>
<td>111</td>
<td>portmapper</td>
</tr>
<tr>
<td>100005</td>
<td>3</td>
<td>tcp</td>
<td>32782</td>
<td>mountd</td>
</tr>
<tr>
<td>100003</td>
<td>2</td>
<td>udp</td>
<td>2049</td>
<td>nfs</td>
</tr>
</tbody>
</table>

and many more besides
Most applications have *nothing* to do with the OS

A Web server could run, almost unchanged, just as well on Windows as on Unix

(Minor differences in a few system calls)

But some do *authentication*
Telnet and SSH permit remote logins. They have to authenticate users, using OS-specific mechanisms. Telnet can invoke login; SSH, which can do its own authentication, cannot.
Anonymous FTP

- Ordinary FTP has to authenticate users; see above
- Anonymous FTP has a different problem: confining the remote user
- Must use OS-specific sandbox mechanism
Middleware

- Common middle layer between applications and the stack
- I.e., common middle ground between applications and the OS
- Arguably as much a part of the OS as the C runtime library
Goals of Middleware

- Common interface
- Naming
- Replication
- Access
Types of Middleware

- Document-based: the Web
- File-based: distributed file system
- Object-based: CORBA
- Publish-subscribe
- More...
CORBA: *Common Object Request Broker Architecture*

- RPC extended procedure calls to the net
- CORBA extends object-oriented programming to the net
- Instead of referencing files or web pages, you invoke methods
- The CORBA Object Broker handles naming, location, access method, etc.
Publish-Subscribe

- Processes that have information *publish* it
- On a LAN, implemented as a broadcast
- Information router forwards such broadcasts to other interested LANs
- If a process *subscribes* to a topic, its local information broker tells other LANs it wants such data
The Global Grid

- Share computing resources around the world
  (The original goal of the ARPANET!)
- Built on high performance computers and high performance networks
The Grid functions like a large-scale distributed operating system.
It has to solve all the distributed OS problems we talked about.
These include security, scheduling, locking, communications, and more.
Actually, There Are Many Grids

- The Grid is a concept and a set of protocols, not a single Internet-wide virtual machine
- You and your friends can pool your own machines
- Each Grid sets up its own access control policy
Scenarios

- Specific computer; local I/O only — simply have to authenticate to host computer
- Specific computer; remote I/O — must delegate security credentials to host OS to allow file retrieval and update
- Run on “best” computer — must talk to scheduler and delegate credentials; scheduler must verify acceptability of credentials on each candidate machine
- Multiprocessing — each remote job needs credentials to talk to other pieces
Security Issues

- User authentication — straight-forward
- Must protect grid computers from remote users — standard operating system problem
- Must have large-scale, secure, distributed file system
- Users want to protect their data from the remote OS — hard!
Sandboxing

- If untrusted users are running jobs on your machine, you may want to sandbox them.
- But each user needs different resources; you need a flexible sandbox.
- You want to isolate each remote user from every other, and (if possible) from your local users.
The Grid is for *big* problems; these take a lot of storage space

- How do we manage allocation?
- How do we transfer that much over the net?
- How do we encrypt that much data during storage?
Accounting

- Someone has to pay for resources used
- The Grid includes accounting standards — record CPU, disk, and RAM usage (and perhaps bandwidth)
Other Essential Components

- Name space
- File I/O and format conversion
- File replication and cache engine
- Brokers, which talk to schedulers
- These are middleware