Outline

1 Networking overview
2 Systems issues
3 OS networking facilities
4 Implementing networking in the kernel
5 Network file systems

Computer networking

- Goal: two applications on different computers exchange data
- Requires inter-process (not just inter-node) communication

The 7-Layer and 4-Layer Models

<table>
<thead>
<tr>
<th>OSI</th>
<th>TCP/IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Application</td>
</tr>
<tr>
<td>6</td>
<td>Presentation</td>
</tr>
<tr>
<td>5</td>
<td>Session</td>
</tr>
<tr>
<td>4</td>
<td>Transport</td>
</tr>
<tr>
<td>3</td>
<td>Network</td>
</tr>
<tr>
<td>2</td>
<td>Data link</td>
</tr>
<tr>
<td>1</td>
<td>Physical</td>
</tr>
</tbody>
</table>

Physical Layer

- Computers send bits over physical links
  - E.g., Coax, twisted pair, fiber, radio, …
  - Bits may be encoded as multiple lower-level "chips"
- Two categories of physical links
  - Point-to-point networks (e.g., fiber, twisted pair):
    - Shared transmission medium networks (e.g., coax, radio): Any message can be seen by all nodes
    - Allows broadcast/multicast, but introduces contention
- One important constraint: speed of light
  - $\sim 300,000 \text{ km/sec}$ in a vacuum, slower in fiber
  - SF $\geq \sim 15 \text{ msec}$ → NYC Moore’s law does not apply!

Link Layer, Indirect Connectivity

- When no direct physical connection to destination
- Hop through multiple devices
- Allows links and devices to be shared for multiple purposes
- Must determine which bits are part of which messages intended for which destinations

Packet switched networks
- Pack a bunch of bytes together intended for same destination
- Slap a header on packet describing where it should go

Link Layer: Ethernet

- Originally designed for shared medium (coax), now generally not shared medium (switched)
- Vendors give each device a unique 48-bit MAC address
  - Specifies which card should receive a packet
- Ethernet switches can scale to switch local area networks (thousands of hosts), but not much larger

Packet format:
- Preamble helps device recognize start of packet
- CRC allows receiving card to ignore corrupted packets
- Body up to 1,500 bytes for same destination
- All other fields must be set by sender’s OS (NIC cards tell the OS what the card’s MAC address is, Special addresses used for broadcast/multicast)
**Network Layer: Internet Protocol (IP)**

- **IP used to connect multiple networks**
  - Runs over a variety of physical networks
  - Hence can connect Ethernet, DSL, mobile networks, etc.
  - Most computers today speak IP
- **Every host has a unique 4-byte IP address (16-bytes for IPv6)**
  - (Or at least thinks it has, when there is address shortage)
  - E.g., www.ietf.org → 104.20.0.85
- **Packets are routed based on destination IP address**
  - Address space is structured to make routing practical at global scale
  - E.g., 171.66.* goes to Stanford
  - So packets need IP addresses in addition to MAC addresses

**UDP and TCP**

- **UDP and TCP most popular protocols on IP**
  - Both use 16-bit port number as well as 32-bit IP address
  - Applications bind a port & receive traffic to that port
- **UDP – unreliable datagram protocol**
  - Exposes packet-switched nature of Internet
  - Sent packets may be dropped, reordered, even duplicated (but generally not corrupted)
- **TCP – transmission control protocol**
  - Provides illusion of a reliable “pipe” between two processes on two different machines
  - Masks lost & reordered packets so apps don’t have to worry
  - Handles congestion & flow control

**Outline**

1. Networking overview
2. Systems issues
3. OS networking facilities
4. Implementing networking in the kernel
5. Network file systems

**Principles: Packet Switching & Layering**

- **Packet switching**
  - A packet is a self contained unit of data which contains information necessary for it to reach its destination
  - Independently, for each arriving packet, compute its outgoing link
  - Makes forwarding simple (depends only on packet)
- **Layering**
  - Break system functionality into a hierarchy of layers
  - Each layer uses only the service of the layer below it
  - Layers communicate sequentially with the layers above or below

**Unreliability of IP**

- **Network does not deliver packets reliably**
  - May drop packets, reorder packets, delay packets
  - May even corrupt packets, or duplicate them
- **How to implement reliable TCP on top of IP network?**
  - Note: This is entirely the job of the OS at the end nodes
- **Straw man: Wait for ack for each packet**
  - Send a packet, wait for acknowledgment, send next packet
  - If no ack, timeout and try again
- **Problems?**
Unreliability of IP

- Network does not deliver packets reliably
  - May drop packets, reorder packets, delay packets
  - May even corrupt packets, or duplicate them
- How to implement reliable TCP on top of IP network?
  - Note: This is entirely the job of the OS at the end nodes
- Straw man: Wait for ack for each packet
  - Send a packet, wait for acknowledgment, send next packet
  - If no ack, timeout and try again
- Problems:
  - Low performance over high-delay network
    (bandwidth is one packet per round-trip time)
  - Possible congestive collapse of network
    (if everyone keeps retransmitting when network overloaded)

A little bit about TCP

- Want to save network from congestion collapse
  - Packet loss usually means congestion, so back off exponentially
- Want multiple outstanding packets at a time
  - Get transmit rate up to \( n \)-packet window per round-trip
- Must figure out appropriate value of \( n \) for network
  - Slowly increase transmission by one packet per acked window
  - When a packet is lost, cut window size in half
- Connection set up and tear down complicated
  - Sender never knows when last packet might be lost
  - Must keep state around for a while after close
- Lots more hacks required for good performance
  - Initially ramp \( n \) up faster (but too fast caused collapse in 1986 [Jacobson]), so TCP had to be changed
  - Fast retransmit when single packet lost

Performance: Bandwidth-delay

- Network delay over WAN will never improve much
- But throughput (bits/sec) is constantly improving
- Can view network as a pipe

\[
\text{Bandwidth} \times \text{Delay} = \text{Network throughput}\]

- For full utilization want \( \# \text{bytes in flight} \geq \text{bandwidth} \times \text{delay} \)
  (But don’t want to overload the network, either)
- What if protocol doesn’t involve bulk transfer?
  - E.g., ping-pong protocol will have poor throughput
- Another implication: Concurrency & response time critical for good network utilization

Lots of OS issues for TCP

- Have to track unacknowledged data
  - Keep a copy around until recipient acknowledges it
  - Keep timer around to retransmit if no ack
  - Receiver must keep out of order segments & reassemble
- When to wake process receiving data?
  - E.g., send calls write (fd, message, 8000);
  - First TCP segment arrives, but is only 512 bytes
  - Could wake recipient, but useless w/o full message
  - TCP sets “PUSH” bit at end of 8000 byte write data
- When to send short segment, vs. wait for more data
  - Usually send only one unacknowledged short segment
  - But bad for some apps, so provide NODELAY option
- Must ack received segments very quickly
  - Otherwise, effectively increases RTT, decreasing bandwidth

Outline

1. Networking overview
2. Systems issues
3. OS networking facilities
4. Implementing networking in the kernel
5. Network file systems

Sockets

- Abstraction for communication between machines
- Datagram sockets: Unreliable message delivery
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: send/receive
- Stream sockets: Bi-directional pipes
  - With IP, gives you TCP
  - Bytes written on one end read on the other
  - Reads may not return full amount requested—must re-read
Socket naming
- TCP & UDP name communication endpoints by
  - E.g., /three.pnum/two.pnum-bit IPv/four.pnum address specifies machine (/one.pnum/two.pnum/eight.pnum bits for IPv/six.pnum)
  - /one.pnum/six.pnum-bit TCP/UDP port number demultiplexes within host
- A connection is thus named by 5 components
  - Protocol (TCP), local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP
- OS keeps connection state in protocol control block (PCB) structure
  - Keep all PCB's in a hash table
  - When packet arrives (if destination IP address belongs to host), use 5-tuple to find PCB and determine what to do with packet

System calls for using TCP

Client
- socket – make socket
- bind – assign address
- listen – listen for clients

Server
- socket – make socket
- bind* – assign address
- connect – connect to listening socket
- accept – accept connection

*This call to bind is optional; connect can choose address & port.

Using UDP
- Call socket with SOCK_DGRAM, bind as before
- New system calls for sending individual packets
  - int sendto(int s, const void *msg, int len, int flags, const struct sockaddr *to, socklen_t tolen);
  - int recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, socklen_t *fromlen);
  - Must send/get peer address with each packet
- Can use UDP in connected mode
  - connect assigns remote address
  - send/receive syscalls, like sendto/recvfrom w/o last 2 args

Uses of connected UDP sockets
- Kernel demultiplexes packets based on port
  - Allows different processes getting packets from different peers
  - For security, ports < 1024 usually can’t be bound
  - But can safely inherit UDP port below that connected to one particular peer
- Feedback based on ICMP messages
  - Say no process has bound UDP port you sent packet to...
  - With sendto, you might think network dropping packets
  - Server sends port unreachable message, but only detect it when using connected sockets

Outline
1 Networking overview
2 Systems issues
3 OS networking facilities
4 Implementing networking in the kernel
5 Network file systems

Socket implementation
- Need to implement layering efficiently
  - Add UDP header to data, Add IP header to UDP packet, …
  - De-encapsulate Ethernet packet so IP code doesn’t get confused by Ethernet header
- Don’t store packets in contiguous memory
  - Moving data to make room for new header would be slow
- BSD solution: mbufs [Leffler]
  (Note [Leffler] calls m_nextpkt by old name m_act)
  - Small, fixed-size (256 byte) structures
  - Makes allocation/deallocation easy (no fragmentation)
- BSD Mbufs working example for this lecture
  - Linux uses sk_buffs, which are similar idea
### mbuf details

- **Packets made up of multiple mbufs**
  - **Chained together by** `m_next`
  - Such linked mbufs called **chains**

- **Chains linked with** `m_nextpkt`
  - Linked chains known as **queues**
  - E.g., device output queue

- **Total mbuf size 256 B ⇒ 230 data bytes (depends on size of pointers)**
  - First in chain has `pkt.header`

- **Cluster mbufs have more data**
  - ext header points to data
  - Up to 2 KB not collocated with mbuf
  - `m_data` not used

- **m_flags** is bitwise or of various bits
  - E.g., if cluster, or if `pkt.header` used

### Adding/deleting data with mbufs

- **m_data always points to start of data**
  - Can be `m_data`, or `ext.buf` for cluster mbuf
  - Or can point into middle of that area

- **To strip off a packet header (e.g., TCP/IP)**
  - Increment `m_data`, decrement `m_len`

- **To strip off end of packet**
  - Decrement `m_len`

- Can add data to mbuf if buffer not full

- Otherwise, add data to chain
  - Chain new mbuf at head/tail of existing chain

### mbuf utility functions

- `mbuf *m_copym(mbuf *m, int off, int len, int wait);`
  - Creates a copy of a subset of an mbuf chain
  - Doesn’t copy clusters, just increments reference count
  - `wait` says what to do if no memory (wait or return NULL)

- `void m_adj(struct mbuf *mp, int len);`
  - Trim `len` bytes from head or (if negative) tail of chain

- `mbuf *m_pullup(struct mbuf *n, int len);`
  - Put first `len` bytes of chain contiguously into first mbuf

**Example: Ethernet packet containing IP datagram**
- Trim Ethernet header using `m_adj`
- Call `m_pullup(n, sizeof(ip_hdr));`
- Access IP header as regular C data structure

### Socket implementation

- **Each socket fd has associated socket structure with:**
  - Send and receive buffers
  - Queues of incoming connections (on listen socket)
  - A **protocol control block (PCB)**
  - A **protocol handle** (`struct protosw *`)

- **PCB contains protocol-specific info. E.g., for TCP:**
  - Pointer to IP TCB with source/destination IP address and port
  - Information about received packets & position in stream
  - Information about unacknowledged sent packets
  - Information about timeouts
  - Information about connection state (setup/teardown)

### protosw structure

- **Goal: abstract away differences between protocols**
  - In C++, might use virtual functions on a generic socket struct
  - Here just put function pointers in `protosw` structure

- Also includes a few data fields
  - `type, domain, protocol` – to match socket syscall args, so know which `protosw` to select
  - `flags` – to specify important properties of protocol

- **Some protocol flags:**
  - `ATOMC` – exchange atomic messages only (like UDP, not TCP)
  - `ADDR` – address given with messages (like unconnected UDP)
  - `CONNREQUIRED` – requires connection (like TCP)
  - `WANTRCVBD` – notify system of consumed data (e.g., so TCP can wake up a sending process blocked by flow control)

### protosw functions

- `pr_slowtimo` – called every 1/2 sec for timeout processing
- `pr_drain` – called when system low on space
- `pr_input` – returns mbuf chain of data read from socket
- `pr_output` – takes mbuf chain of data written to socket
- `pr_usrreq` – multi-purpose user-request hook
  - Used for bind/listen/accept/connect/disconnect operations
  - Used for out-of-band data
Network interface cards

- Each NIC driver provides an *ifnet* data structure
  - Like *protosw*, tries to abstract away the details

- Data fields:
  - Interface name (e.g., “eth0”)
  - Address list (e.g., Ethernet address, broadcast address, …)
  - Maximum packet size
  - Send queue

- Function pointers
  - *if_output* – prepend header and enqueue packet
  - *if_start* – start transmitting queued packets
  - Also ioctl, timeout, initialize, reset

Input handling

- NIC driver figures out protocol of incoming packet
- Enqueues packet for appropriate protocol handler
  - If queue full, drop packet (can create livelock [Mogul])
- Posts “soft interrupt” for protocol-layer processing
  - Runs at lower priority than hardware (NIC) interrupt … but higher priority than process-context kernel code

Routing

- An OS must route all transmitted packets
  - Machine may have multiple NICs plus “loopback” interface
  - Which interface should a packet be sent to, and what MAC address should packet have?

- Routing is based purely on the destination address
  - Even if host has multiple NICs w. different IP addresses
  - (Though OSes have features to redirect based on source IP)

- OS maintains routing table
  - Maps IP address & prefix-length → next hop

- Use radix tree for efficient lookup
  - Branch at each node in tree based on single bit of target
  - When you reach leaf, that is your next hop

- Most OSes provide packet forwarding
  - Received packets for non-local address routed out another interface

Outline

1. Networking overview
2. Systems issues
3. OS networking facilities
4. Implementing networking in the kernel
5. Network file systems

Network file systems

- What’s a network file system?
  - Looks like a file system (e.g., FFS) to applications
  - But data potentially stored on another machine
  - Reads and writes must go over the network
  - Also called distributed file systems

- Advantages of network file systems
  - Easy to share if files available on multiple machines
  - Often easier to administer servers than clients
  - Access way more data than fits on your local disk
  - Network + remote buffer cache faster than local disk

- Disadvantages
  - Network + remote disk slower than local disk
  - Network or server may fail even when client OK
  - Complexity, security issues

NFS version 2 [Sandberg]

- Background: ND (networked disk)
  - Creates disk-like device even on diskless workstations
  - Can create a regular (e.g., FFS) file system on it
  - But no sharing—Why?

- ND idea still used today by Linux NBD
  - Useful for network booting/diskless machines, not file sharing

- Some Goals of NFS
  - Access same FS from multiple machines simultaneously
  - Maintain Unix semantics
  - Crash recovery
  - Competitive performance with ND

- NFS version 2 protocol specified in [RFC 1094]
NFS version 2 [Sandberg]

- Background: ND (networked disk)
  - Creates disk-like device even on diskless workstations
  - Can create a regular (e.g., FFS) file system on it
  - But no sharing—Why?
  - FFS assumes disk doesn’t change under it
- ND idea still used today by Linux NBD
  - Useful for network booting/diskless machines, not file sharing
- Some Goals of NFS
  - Access same FS from multiple machines simultaneously
  - Many requests are idempotent
  - Competitive performance with ND
- NFS version 2 protocol specified in [RFC 1094]

Stateless operation

- Designed for “stateless operation”
  - Motivated by need to recover from server crashes
- Requests are self-contained
- Requests are idempotent
  - Unreliable UDP transport
  - Client retransmits requests until it gets a reply
  - Writes must be stable before server returns
- Can this really work?

NFS version 3

- Same general architecture as NFS 2
- Specified in RFC 1813 (subset of Open Group spec)
  - XDR defines C structures that can be sent over network; includes tagged unions (to know which union field active)
  - Protocol defined as a set of Remote Procedure Calls (RPCs)
- New access RPC
  - Supports clients and servers with different uids/gids
- Better support for caching
  - Unstable writes while data still cached at client
  - More information for cache consistency
- Better support for exclusive file creation

Virtualized the file system with vnodes
- Basically poor man’s C++ (like proto struct)

Vnode structure represents an open (or openable) file

Bunch of generic “vnode operations”:
- lookup, create, open, close, getattr, setattr, read, write, fsync, remove, link, rename, mkdir, rmdir, symlink, readdir, readdir, ...
- Called through function pointers, so most system calls don’t care what type of file system a file resides on

NFS vnode operations perform Remote Procedure Calls (RPC)
- Client sends request to server over network, awaits response
- Each system call may require a series of RPCs
  - System mostly determined by RPC [RFC 1831] Protocol
- Uses XDR protocol specification language [RFC 1832]

NFSv3 File handles

```c
struct nfs_fh3 {
    /* XDR notation for variable-length array
       * with 0-64 opaque bytes: */
    opaque data<64>;
};
```

- Server assigns an opaque file handle to each file
  - Client obtains first file handle out-of-band (mount protocol)
  - File handle hard to guess – security enforced at mount time
  - Subsequent file handles obtained through lookups
- File handle internally specifies file system & file
  - Device number, i-number, generation number, ...
  - Generation number changes when inode recycled
- Handle generally doesn’t contain filename
  - Clients may keep accessing an open file after it’s renamed
File attributes

```
struct fattr3 {
    specdata3 rdev;
    ftype3 type;
    uint64 fsid;
    uint32 mode;
    uint64 fileid;
    uint32 nlink;
    nfstime3 atime;
    uint32 uid;
    nfstime3 mtime;
    uint32 gid;
    nfstime3 ctime;
    uint64 size;
    uint64 used;
};
```

- Most operations can optionally return fattr3
- Attributes used for cache-consistency

Lookup

```
struct diropargs3 {
    struct lookup3resok {
        nfs_fh3 dir;
        nfs_fh3 object;
        filename3 name;
        post_op_attr obj_attributes;
    };
    post_op_attr dir_attributes;
};
```

```
union lookup3res switch (nfsstat3 status) {
    case NFS3_OK:
        lookup3resok resok;
    default:
        post_op_attr resfail;
};
```

- Maps ⟨directoryhandle, filename⟩ → handle
  - Client walks hierarchy one file at a time
  - No symlinks or file system boundaries crossed
  - Client must expand symlinks

Create

```
struct create3args {
    diropargs3 where;
    createhow3 how;
};
```

```
union createhow3 switch (createmode3 mode) {
    case UNCHECKED:
    case GUARDED:
    case EXCLUSIVE:
        createverf3 verf;
};
```

- UNCHECKED – succeed if file exists
- GUARDED – fail if file exists
- EXCLUSIVE – persistent record of create

Read

```
struct read3args {
    struct read3resok {
        nfs_fh3 file;
        post_op_attr file_attributes;
        uint64 offset;
        uint32 count;
        bool eof;
    };
    opaque data<>>;
};
```

```
union read3res switch (nfsstat3 status) {
    case NFS3_OK:
        read3resok resok;
    default:
        post_op_attr resfail;
};
```

- Offset explicitly specified (not implicit in handle)
- Client can cache result

Data caching

- Client can cache blocks of data read and written
- Consistency based on times in fattr3
  - mtime: Time of last modification to file
  - ctime: Time of last change to inode
    (Changed by explicitly setting mtime, increasing size of file, changing permissions, etc.)
- Algorithm: If mtime or ctime changed by another client, flush cached file blocks

Write discussion

- When is it okay to lose data after a crash?
  - Local file system?
  - Network file system?
    - Application not killed, so shouldn’t lose previous writes
    - NFSv牵/状/状 addresses problem by having server write data to disk before replying to a write RPC
      - Caused performance problems
    - Could NFS牵/状/状 clients just perform write-behind?
      - Implementation issues – used blocking kernel threads on write
    - Semantics – how to guarantee consistency after server crash
      - Solution: small # of pending write RPCs, but write through on close; if server crashes, client keeps re-writing until acked
Write discussion

• When is it okay to lose data after a crash?
  - Local file system?
    If no calls to fsync, OK to lose 30 seconds of work after crash
  - Network file system?
    What if server crashes but not client?
    Application not killed, so shouldn’t lose previous writes

• NFSv2 addresses problem by having server write data to disk before replying to a write RPC
  - Caused performance problems

• Could NFSv2 clients just perform write-behind?
  - Implementation issues – used blocking kernel threads on write
  - Semantics – how to guarantee consistency after server crash
  - Solution: small # of pending write RPCs, but write through on close; if server crashes, client keeps re-writing until acked

NFSv2 write call

```c
struct writeargs { union attrstat fhandle file; switch (stat status) {
  unsigned beginoffset; case NFS_OK:
  unsigned offset; fattr attributes;
  unsigned totalcount; default:
  nfsdata data; void;
}; attrstat NFSPROC_WRITE(writeargs) = 8;
```

• On successful write, returns new file attributes
• Can NFSv2 keep cached copy of file after writing it?

NFSv3 Write arguments

```c
struct write3args {
  enum stable_how {
    nfs_fh3 file;
    uint64 offset;
    uint32 count;
    opaque data<>;
  };
  stable_how stable;
  opaque data<>;
};
```

• Two goals for NFSv3 write:
  - Don’t force clients to flush cache after writes
  - Don’t equate cache consistency with crash consistency
    i.e., don’t wait for disk just so another client can see data

Write results

```c
struct write3resok {
  struct wcc_attr {
    wcc_data file_wcc;
    uint32 count;
    stable_how committed;
    writeverf3 verf;
  };
  wcc_attr resok;
  union write3res;
  switch (nfsstat3 status) {
    case NFS3_OK:
      write3resok resok;
      default:
      wcc_data resfail;
    }
};
```

• Several fields added to achieve these goals
**Data caching after a write**

- Write will change mtime/ctime of a file
  - "after" will contain new times
  - With NFSv2, would require cache to be flushed
- With NFSv3, "before" contains previous values
  - If before matches cached values, no other client has changed file
  - Okay to update attributes without flushing data cache

**Write stability**

- Server write must be at least as stable as requested
- If server returns write UNSTABLE
  - Means permissions okay, enough free disk space, …
  - But data not on disk and might disappear (after crash)
- If DATA_SYNC, data on disk, maybe not attributes
- If FILE_SYNC, operation complete and stable

**Commit operation**

- Client cannot discard any UNSTABLE write
  - If server crashes, data will be lost
- COMMIT RPC commits a range of a file to disk
  - Invoked by client when client cleaning buffer cache
  - Invoked by client when user closes/flashes a file
- How does client know if server crashed?
  - Write and commit return writeverf3
  - Value changes after each server crash (can be boot time)
  - Client must resend all writes if verf value changes

**Attribute caching**

- Close-to-open consistency
  - Annoying if writes not visible after a file close
    (Edit file, compile on another machine, get old version)
  - Nowadays, all NFS opens fetch attributes from server
- Still, lots of other need for attributes (e.g., `ls -al`)
- Attributes cached between 5 and 60 seconds
  - Files recently changed more likely to change again
  - Do weighted cache expiration based on age of file
- Drawbacks:
  - Must pay for round-trip to server on every file open
  - Can get stale info when `stat`ting a file