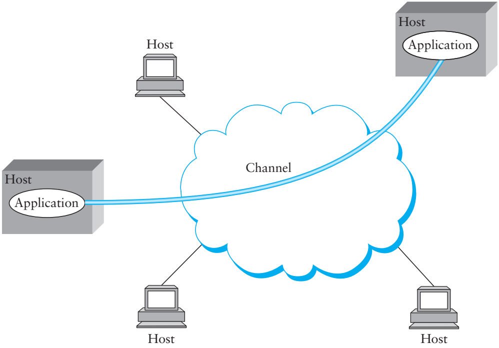
Outline

Computer networking



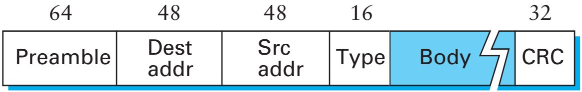
- **Goal: two applications on different computers exchange data**
- **Requires inter-process (not just inter-node) communication**

The 7-Layer and 4-Layer Models

	OSI	TCP/IP
7	Application	Applications (FTP, SMTP, HTTP, etc.)
6	Presentation	
5	Session	
4	Transport	TCP (host-to-host)
3	Network	IP
2	Data link	Network access (usually Ethernet)
1	Physical	

Link Layer: Ethernet

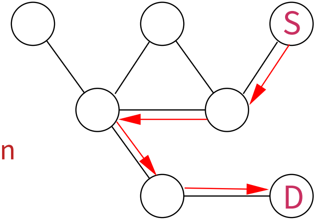
- **Originally designed for shared medium (coax), now generally not shared medium (switched)**
- **Vendors give each device a unique 48-bit *MAC address***
  - Specifies which card should receive a packet
- **Ethernet switches can scale to switch local area networks (thousands of hosts), but not much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

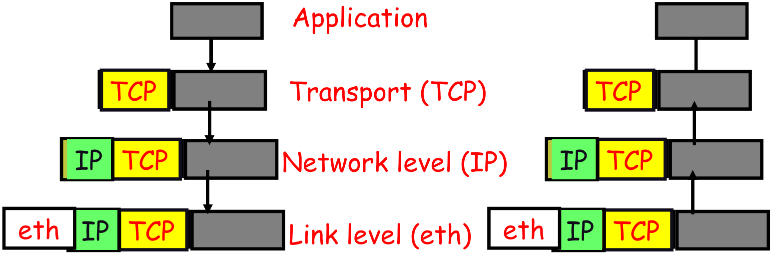
- **IP used to connect multiple networks**
  - Runs over a variety of physical networks—Ethernet, DSL, 4G
- **Every host has a unique 4-byte IP address (16-bytes for IPv6)**
  - (Or at least thinks it has, when there is address shortage)
- **Packets are *routed* based on destination IP address**
  - 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



- **Inside IP: UDP or TCP transport layer adds 16-bit *port number***
  - UDP – unreliable datagram protocol, exposes lost/reordered/delayed (but typically not corrupted) packets
  - TCP – transmission control protocol ≈ reliable pipe

Principle: Encapsulation

- **Stick packets inside packets**
- **How you realize packet switching and layering in a system**
  - E.g., an Ethernet packet may *encapsulate* an IP packet
  - An IP router *forwards* a packet from one Ethernet to another, creating a new Ethernet packet containing the same IP packet
  - In principle, an inner layer should not depend on outer layers (not always true)



Outline

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

Unreliability of IP

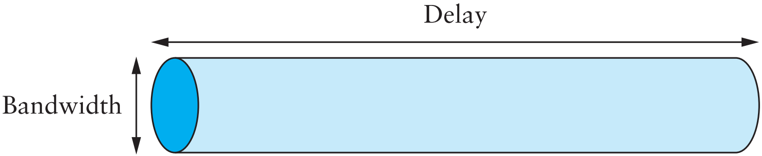
- **Network does not deliver packets reliably**
  - May drop, reorder, delay, corrupt, duplicate packets
- **OS must implement reliable TCP on top of IP**
- **Straw man: Wait for ack for each packet**
  - Send a packet, wait for acknowledgment, send next packet
  - If no ack, timeout and try again
- **Problems?**

Unreliability of IP

- **Network does not deliver packets reliably**
  - May drop, reorder, delay, corrupt, duplicate packets
- **OS must implement reliable TCP on top of IP**
- **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)

Performance: Bandwidth-delay

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



  - For full utilization want # bytes in flight $\geq$ bandwidth $\times$ 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**

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 teardown complicated**
  - Sender never knows when last packet might be lost
  - Must keep state around for a while (2MSL, e.g., 4 min) 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

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., sender 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 unacked 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. Implementing networking in the kernel
4. Network file systems

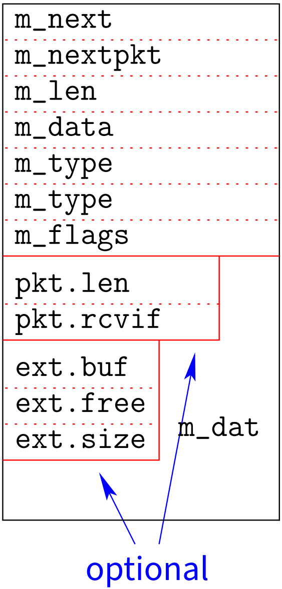
Sockets

- **Sockets $\approx$ bi-directional pipes**
- **Name endpoints by IP address and 16-bit *port number***
- **A *connection* is thus named by 5 components**
  - Protocol (TCP), local IP, local port, remote IP, remote port
  - Note TCP requires connected sockets, while UDP does not
- **Kernel stores connection state in a *protocol control block* structure (PCB)**
  - 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

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_dat` 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_dat`, 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:**
  - 5-tuple of protocol (TCP), 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**
  - *domain, type, protocol* – to match `socket` syscall args, so know which `protosw` to select
  - *flags* – to specify important properties of protocol
- **Some protocol flags:**
  - ATOMIC – exchange atomic messages only (like UDP, not TCP)
  - ADDR – address given with messages (like unconnected UDP)
  - CONNREQUIRED – requires connection (like TCP)
  - WANTRCVD – notify socket of consumed data (e.g., so TCP can wake up a sending process blocked by flow control)

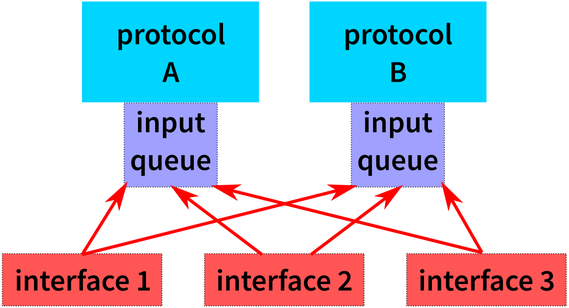
protosw functions

- 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 $\rightarrow$ 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. Implementing networking in the kernel
4. 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
  - Maintain Unix semantics
  - Crash recovery
  - Competitive performance with ND
- **NFS version 2 protocol specified in [RFC 1094]**

NFS implementation

- **Virtualized the file system with *vnodes***
  - Basically poor man's C++ (like `protosw` 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, readlink, . . .
  - 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]

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?**

Stateless operation

- **Designed for "stateless operation"**
  - Motivated by need to recover from server crashes
- **Requests are self-contained**
- **Requests are** *mostly* **idempotent**
  - Unreliable UDP transport
  - Client retransmits requests until it gets a reply
  - Writes must be stable before server returns
- **Can this really work?**
  - Of course, FS not stateless – it stores files
  - E.g., *mkdir* can't be idempotent – second time dir exists
  - But many operations, e.g., *read*, *write* are idempotent

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**

NFSv3 File handles

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

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

Lookup

```
struct diropargs3 {
  nfs_fh3 dir;
  filename3 name;
};
```

```
struct lookup3resok {
  nfs_fh3 object;
  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** $\langle$directory handle, filename$\rangle \rightarrow$ handle
  - Client walks hierarchy one file at a time
  - No symlinks expanded 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:
  sattr3 obj_attributes;
case EXCLUSIVE:
  createverf3 verf;
};
```

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

Read

```
struct read3args {
  nfs_fh3 file;
  uint64 offset;
  uint32 count;
};
```

```
struct read3resok {
  post_op_attr file_attributes;
  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?*

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?*

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 NFS2 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

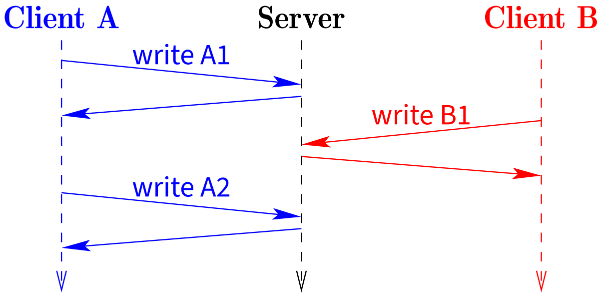
```
struct writeargs {
  fhandle file;
  unsigned beginoffset;
  unsigned offset;
  unsigned totalcount;
  nfsdata data;
};
```

```
union attrstat
    switch (stat status) {
case NFS_OK:
  fattr attributes;
default:
  void;
};
```

```
attrstat NFSPROC_WRITE(writeargs) = 8;
```

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

Write race condition



- **Suppose client overwrites 2-block file**
  - Client A knows attributes of file after writes A1 & A2
  - But client B could overwrite block 1 between the A1 & A2
  - No way for client A to know this hasn't happened
  - Must flush cache before next file read (or at least open)

NFSv3 Write arguments

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

```
enum stable_how {
  UNSTABLE = 0,
  DATA_SYNC = 1,
  FILE_SYNC = 2
};
```

- **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

```
struct write3resok {
  wcc_data file_wcc;
  uint32 count;
  stable_how committed;
  writeverf3 verf;
};

union write3res
    switch (nfsstat3 status) {
case NFS3_OK:
  write3resok resok;
default:
  wcc_data resfail;
};
```

```
struct wcc_attr {
  uint64 size;
  nfstime3 mtime;
  nfstime3 ctime;
};

struct wcc_data {
  wcc_attr *before;
  post_op_attr after;
};
```

- **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/flushes 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

NFS version 4 [RFC 3530]

- **Much more complicated than version 3**
  - NFS2: 27 page spec, NFS3: 126 pages, NFS4: 275 pages, NFS4.1: 617 pages
- **Designed to run over higher-latency networks**
  - Support for multi-component lookups to save RTTs
  - Support for batching multiple operations in one RPC
  - Support for leases (in two slides) and stateful (open, close) operation
- **Designed to be more generic and less Unix-specific**
  - E.g., support for extended file attributes, etc.
- **Lots of security stuff**
- **NFS 4.1 [RFC5661] has better support for NAS**
  - Store file data and metadata in different places

Callbacks

- **NFSv2 and v3 poll server for cache consistency**
  - Client requests attributes (via ACCESS) when file opened
  - Attributes validate or invalidate cached copy of file
- **Alternative: Server calls back to clients caching file**
  - Invalidate immediately, rather than when cache needed
  - Requires server to maintain list of all clients caching info
- **Advantages**
  - Tight consistency, 0 RTT opens of cached files
- **Disadvantages**
  - Server must maintain a lot of state
  - Updates potentially slow
    - Must persistently record who is caching things on server
    - Must wait for *n* clients to acknowledge invalidations
  - When a client goes down, other clients will block

Leases

- **Hybrid mix of polling and callbacks**
  - Server agrees to notify client of changes for a limited period of time – the lease term
  - After the lease expires, client must poll for freshness
- **Avoids paying for a server round trip in many cases**
- **Server doesn't need to keep long-term track of callbacks**
  - E.g., lease time can be shorter than crash-reboot—no need to keep callbacks persistently
- **If client crashes, resume normal operation after lease expiration**