

CS240d – Distributed Storage Systems

Instructor: David Mazières
Stanford University

Course topics

- **Networking background**
- **Local storage**
 - File systems, database consistency, crash recovery
- **Distributed file systems**
 - Scalability, security, availability, consistency
- **Storage Architectures**
 - Virtualizing storage, RAID
- **Other storage systems**
 - Untrusted storage, OO databases, peer-to-peer systems

Class overview

- **Readings & class discussion**
- **Lab 1: TCP proxy (\sim 1 week, solo)**
- **Lab 2: Encrypting file system (\sim 3.5 weeks, solo)**
- **Lab 3: File server (in preparation...)**
- **Final project (group)**
- **Midterm and final quizzes**

Should you take this class?

- **Caveats:**

- Probably *not* the easiest systems elective
- Large programming assignments in C++
- Rely on custom asynchronous I/O, C++ libraries with gnarly template hacks

- **Benefits:**

- Build real systems you might want to use
- Gain a good understanding of a complex system
- Event-driven programming is useful to know (e.g., much of google reportedly built around similar libraries)
- Libraries are free software; if you learn to use them you can prototype fast servers easily

System calls

- **Problem: How to access resources other than CPU**
 - Disk, network, terminal, other processes
 - CPU prohibits instructions that would access devices
 - Only privileged OS “kernel” can access devices
- **Applications request I/O operations from kernel**
- **Kernel supplies well-defined *system call* interface**
 - Applications set up syscall arguments and *trap* to kernel
 - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
 - `printf`, `scanf`, `gets`, etc. all user-level code

I/O through the file system

- **Applications “open” files/devices by name**
 - I/O happens through open files
- `int open(char *path, int flags, ...);`
 - flags: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
 - `O_CREAT`: create the file if non-existent
 - `O_EXCL`: (w. `O_CREAT`) create if file exists already
 - `O_TRUNC`: Truncate the file
 - `O_APPEND`: Start writing from end of file
 - `O_SYNC` – Writes don't return until data on disk
 - mode: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if open fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
 - Specific kind of error in global int errno
- **#include <sys/errno.h> for possible values**
 - 2 = ENOENT “No such file or directory”
 - 13 = EACCES “Permission Denied”
- **perror, strerror print human-readable messages**
 - perror ("initfile");
 - printf ("initfile: %s\n", strerror (errno));
→ “initfile: No such file or directory”

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
 - Returns number of bytes read
 - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, void *buf, int nbytes);`
 - Returns number of bytes read, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
 - whence: 0 – start, 1 – current, 2 – end
 - Returns previous file offset, or -1 on error
- `int close (int fd);`
- `int fsync (int fd);`
 - Guarantee that file contents is stably on disk

Other system calls on pathnames

- `int chdir (const char *dir);`
 - Change working directory (what `cd` command does)
- `int mkdir (const char *dir);`
- `int rmdir (const char *dir);`
 - Make and remove directories
- `int unlink (const char *path);`
 - Delete pathname specified by `path`
- `int link (const char *p1, const char *p2);`
 - Creates `p2`; `p1` & `p2` identical directory entries
- `int symlink (const char *p1, const char *p2);`
 - Creates `p2`; `p2` is an *alias* for name `p1`

The rename system call

- `int rename (const char *p1, const char *p2);`
 - Changes name p2 to reference file p1
 - Removes file name p1
- **Guarantees that p2 will exist despite any crashes**
 - p2 may still be old file
 - p1 and p2 may both be new file
 - but p2 will always be old or new file
- **fsync/rename idiom used extensively**
 - E.g., emacs: Writes file `.#file#`
 - Calls `fsync` on file descriptor
 - `rename (".#file#", "file");`

File descriptor numbers

- **File descriptors are inherited by processes**
 - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
 - 0 – “standard input” (stdin in ANSI C)
 - 1 – “standard output” (stdout, printf in ANSI C)
 - 2 – “standard error” (stderr, perror in ANSI C)
 - Normally all three attached to terminal

Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
 - Closes `newfd`, if it was a valid descriptor
 - Makes `newfd` an exact copy of `oldfd`
 - Two file descriptors will share same offset
(`lseek` on one will affect both)
- `int fcntl (int fd, F_SETFD, int val)`
 - Sets *close on exec* flag if `val = 1`, clears if `val = 0`
 - Makes file descriptor non-inheritable by spawned programs

File descriptor flags

- `int fcntl (int fd, F_GETFL)`
- `int fcntl (int fd, F_SETFL, int val)`
- **Set or get same flags as in 3rd open arg:**
- **For sockets/pipes (discussed later):**
 - `O_NONBLOCK` – Return immediately even if operation cannot complete immediately
 - `O_ASYNC` – Sends `SIGIO` signal when I/O ready (don't confuse with `O_NONBLOCK`, `O_ASYNC` less used)

Pipes

- `int pipe (int fds[2]);`
 - Returns two file descriptors in `fds[0]` and `fds[1]`
 - Writes to `fds[1]` will be read on `fds[0]`
 - When last copy of `fds[1]` closed, `fds[0]` will return EOF
 - Returns 0 on success, -1 on error
- **Operations on pipes**
 - `read/write/close` – as with files
 - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
 - When `fds[0]` closed, `write(fds[1])`:
 - Kills process with SIGPIPE, or if blocked
 - Fails with EPIPE

Sockets: Communication between machines

- **Datagram sockets: Unreliable message delivery**
 - On Internet: User Datagram Protocol (UDP)
 - Send atomic messages, which may be reordered or lost
 - Special system calls to read/write: `send/recv`
- **Stream sockets: Bi-directional pipes**
 - On Internet: Transmission Control Protocol (TCP)
 - Bytes written on one end read on the other
 - Reads may not return full amount requested—must re-read

Socket naming

- **Every Internet host has a unique 32-bit *IP address***
 - Often written in “dotted-quad” notation: 171.66.3.191
 - DNS maps names (www.stanford.edu) to IP addresses
 - Network routes packets based on IP address
- **16-bit *port number* demultiplexes TCP traffic**
 - Well-known services “listen” on standard ports: finger—79, HTTP—80, mail—25, ssh—22
 - Clients connect from arbitrary ports to well known ports
 - A connection consists of five components: Protocol (TCP), local IP, local port, remote IP, remote port
- **All Internet traffic routed as small packets**
 - Each packet contains address information in header

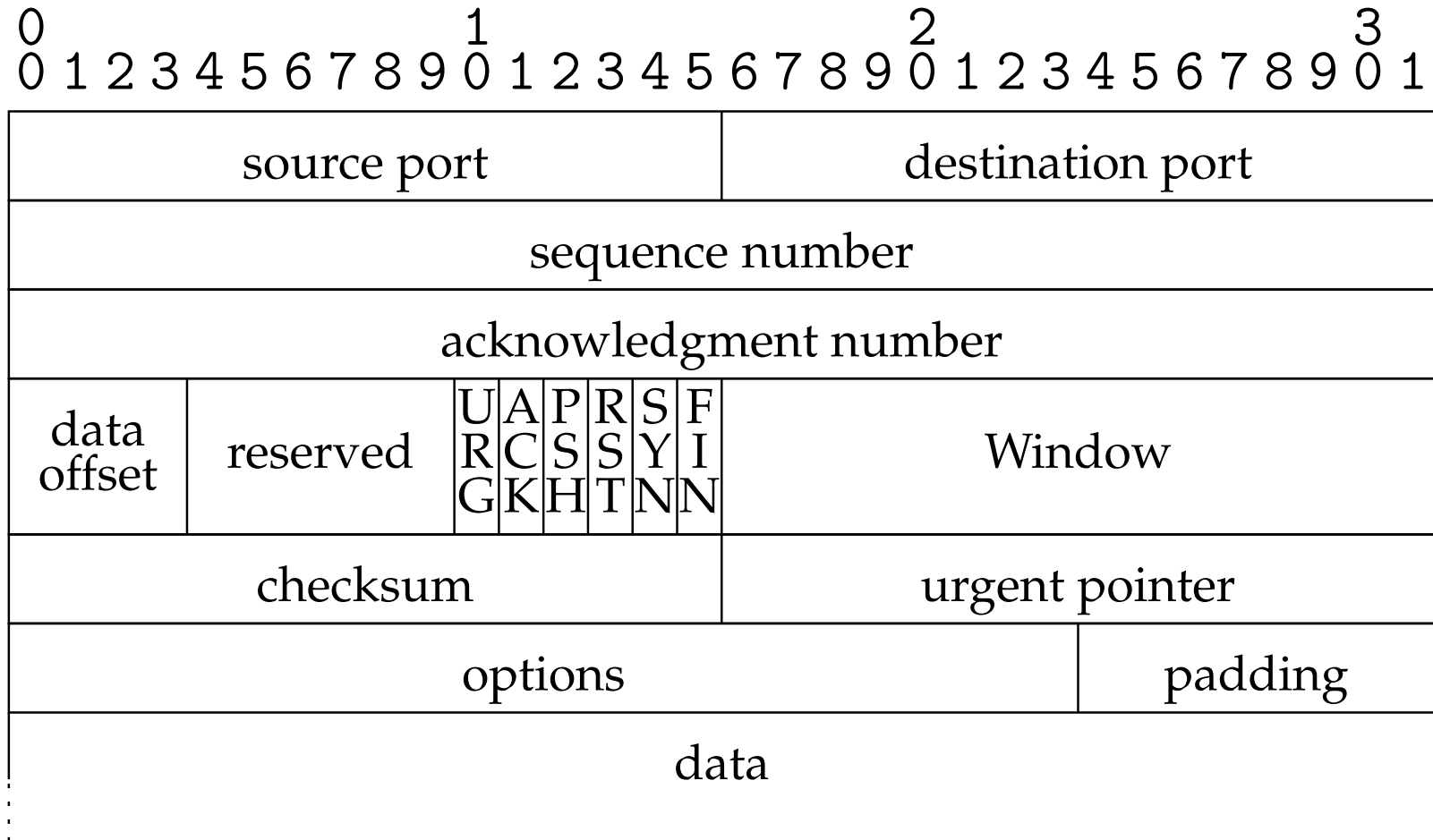
IP header

<div>001234567890123456789012345678901</div> <div>101</div> <div>201</div> <div>301</div>																																
vers			hdr len						TOS						Total Length																	
Identification												0	D	M	Fragment offset																	
TTL						Protocol						hdr checksum																				
Source IP address																																
Destination IP address																																
Options																								Padding								

IP header details

- **Routing is based on destination address**
- **TTL (time to live) decremented at each hop (avoids loops)**
- **Fragmentation used for large packets**
 - Fragmented in network if links crossed with smaller MTU
 - MF bit means more fragments for this IP packet
 - DF bit says “don’t fragment” (returns error to sender)
- **Almost always want to avoid fragmentation**
 - When fragment is lost, whole packet must be retransmitted
- **Following IP header is “payload” data**
 - Typically beginning with TCP or UDP header

TCP header



TCP fields

- Ports
- Seq no. – segment position in byte stream
- Ack no. – seq no. sender expects to receive next
- Data offset – # of 4-byte header & option words
- Window – willing to receive (flow control)
- Checksum
- Urgent pointer

TCP Flags

- **URG** – urgent data present
- **ACK** – ack no. valid (all but first segment)
- **PSH** – push data up to application immediately
- **RST** – reset connection
- **SYN** – “synchronize” establishes connection
- **FIN** – close connection

A TCP Connection (no data)

orchard.48150 > essex.discard:

S 1871560457:1871560457(0) win 16384

essex.discard > orchard.48150:

S 3249357518:3249357518(0) ack 1871560458 win 17376

orchard.48150 > essex.discard: . ack 1 win 17376

orchard.48150 > essex.discard: F 1:1(0) ack 1 win 17376

essex.discard > orchard.48150: . ack 2 win 17376

essex.discard > orchard.48150: F 1:1(0) ack 2 win 17376

orchard.48150 > essex.discard: . ack 2 win 17375

Connection establishment

- **Three-way handshake:**
 - $C \rightarrow S$: SYN, seq S_C
 - $S \rightarrow C$: SYN, seq S_S , ack $S_C + 1$
 - $C \rightarrow S$: ack $S_S + 1$
- **If no program listening: server sends RST**
- **If server backlog exceeded: ignore SYN**
- **If no SYN-ACK received: retry, timeout**

Connection termination

- **FIN bit says no more data to send**
 - Caused by close or shutdown on sending end
 - Both sides must send FIN to close a connection
- **Typical close:**
 - $A \rightarrow B$: FIN, seq S_A , ack S_B
 - $B \rightarrow A$: ack $S_A + 1$
 - $B \rightarrow A$: FIN, seq S_B , ack $S_A + 1$
 - $A \rightarrow B$: ack $S_B + 1$
- **Can also have simultaneous close**
- **After last message, can A and B forget about closed socket?**

TIME_WAIT

- **Problems with closed socket**

- What if final ack is lost in the network?
- What if the same port pair is immediately reused for a new connection? (Old packets might still be floating around.)

- **Solution: “active” closer goes into TIME_WAIT**

- Active close is sending FIN before receiving one
- After receiving ACK and FIN, keep socket around for 2MSL (twice the “maximum segment lifetime”)

- **Can pose problems with servers**

- OS has too many sockets in TIME_WAIT, slows things down
- Hack: Can send RST and delete socket, set SO_LINGER socket option to time 0 (useful for benchmark programs). (Some OSes may disable this hack.)

Sending data

- **Data sent in MSS-sized segments**
 - Chosen to avoid fragmentation (e.g., 1460 on ethernet LAN)
 - Write of 8K might use 6 segments—PSH set on last one
 - PSH avoids unnecessary context switches on receiver
- **Sender's OS can delay sends to get full segments**
 - Nagle algorithm: Only one unacknowledged short segment
 - TCP_NODELAY option avoids this behavior
- **Segments may arrive out of order**
 - Sequence number used to reassemble in order
- **Window achieves flow control**
 - If window 0 and sender's buffer full, write will block or return EAGAIN

Socket options

- `int getsockopt (int s, int level, int optname, void *optval, socklen_t *optlen);`
- `int setsockopt (int s, int level, int optname, const void *optval, socklen_t *optlen);`
- **“Level”** SOL_SOCKET
 - SO_RCVBUF, SO_SNDBUF – send/receive buffer sizes
 - SO_TYPE (get only), SO_LINGER, SO_BROADCAST
- **“Level”** IPPROTO_IP
 - IP_TOS – e.g., set to IPTOS_LOWDELAY
- **“Level”** IPPROTO_TCP
 - TCP_NODELAY

A TCP connection (3 byte echo)

orchard.38497 > essex.echo:

S 1968414760:1968414760(0) win 16384

essex.echo > orchard.38497:

S 3349542637:3349542637(0) ack 1968414761 win 17376

orchard.38497 > essex.echo: . ack 1 win 17376

orchard.38497 > essex.echo: P 1:4(3) ack 1 win 17376

essex.echo > orchard.38497: . ack 4 win 17376

essex.echo > orchard.38497: P 1:4(3) ack 4 win 17376

orchard.38497 > essex.echo: . ack 4 win 17376

orchard.38497 > essex.echo: F 4:4(0) ack 4 win 17376

essex.echo > orchard.38497: . ack 5 win 17376

essex.echo > orchard.38497: F 4:4(0) ack 5 win 17376

orchard.38497 > essex.echo: . ack 5 win 17375

Delayed ACKs

- **Goal: Piggy-back ACKs on data**
 - Echo server just echoes, why send separate ack first?
 - Delay ACKs for 200 msec in case application sends data
 - If more data received, immediately ACK second segment
 - Note: Never delay duplicate ACKs (if segment out of order)
- **Warning: Can interact badly with Nagle**
 - “My login has 200 msec delays”
 - Set TCP_NODELAY
 - In libasync library, call `tcp_nodelay (fd);`

Retransmission

- TCP dynamically estimates round trip time
- If segment goes unacknowledged, must retransmit
- Use exponential backoff (in case loss from congestion)
 - Optimization in case of single lost packet—just halve sending rate
- After ~ 10 minutes, give up and reset connection

System calls for using TCP

Client

Server

socket – make socket

bind – assign address

listen – listen for clients

socket – make socket

bind – assign address (optional)

connect – connect to listening socket

accept – accept connection

Example client

```
struct sockaddr_in {
    short    sin_family;   /* = AF_INET */
    u_short  sin_port;     /* = htons (PORT) */
    struct   in_addr sin_addr;
    char     sin_zero[8];
} sin;
```

```
int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (13); /* daytime port */
sin.sin_addr.s_addr = htonl (IP_ADDRESS);
connect (s, (sockaddr *) &sin, sizeof (sin));
while ((n = read (s, buf, sizeof (buf))) > 0)
    write (1, buf, n);
```


Example server

```
struct sockaddr_in sin;
int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (9999);
sin.sin_addr.s_addr = htonl (INADDR_ANY);
bind (s, (sockaddr *) &sin, sizeof (sin));
listen (s, 5);

for (;;) {
    socklen_t len = sizeof (sin);
    int cfd = accept (s, (sockaddr *) &sin, &len);
    /* do something with cfd */
    close (cfd);
}
```

Concurrent connections

- **Servers must handle multiple clients concurrently**
 - Read or write of a socket connected to slow client can block
 - Overlap network latency with CPU, transmission, disk I/O
 - Keep disk queues full when server accesses disk
- **Can use one process per client: accept, fork, close**
 - High overhead, cannot share state between clients
- **Can use threads for concurrency**
 - Data races and deadlock make programming tricky
 - Must allocate one stack per request
- **Use non-blocking read/write calls**
 - Unusual programming model

Non-blocking I/O

- **Use `fcntl` to set `O_NONBLOCK` flag on descriptor**
 - In `libasync`, can call `make_async (fd);`
- **Non-blocking semantics of system calls:**
 - `read` immediately returns -1 with `errno EAGAIN` if no data
 - `write` may not write all data, or may return `EAGAIN`
 - `connect` may “fail” with `EINPROGRESS` (or may succeed, or may fail with real error like `ECONNREFUSED`)
 - `accept` may fail with `EAGAIN` if no pending connections

How do you know when to read/write?

```
struct timeval {
    long    tv_sec;          /* seconds */
    long    tv_usec;        /* and microseconds */
};

int select (int nfd, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);

FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);
```

Asynchronous programming model

- **Many non-blocking file descriptors in one process**
 - Wait for pending I/O events on file many descriptors
 - Each event triggers some *callback* function
- **Lab: libasync – supports event-driven model**
 - Register callbacks on file descriptors
 - Call `amain()` – main select loop
 - Add/delete callbacks from within callbacks

callback.h

- **Problem: Need state from one callback to next**
- **wrap bundles a function with its arguments**

```
callback<void, int>::ref errwrite = wrap (write, 2);  
(*errwrite) ("hello", 5); // writes "hello" to stderr
```

- `void fdcb(int fd, selop op, cb_t cb);`
registers callbacks on file descriptor fd
 - op is selread or selwrite
 - cb is void callback (no arguments), or NULL to clear

libasync example server

```
void doaccept (int lfd) {
    sockaddr_in sin;
    bzero (&sin, sizeof (sin));
    socklen_t sinlen = sizeof (sin);
    int cfd = accept (lfd, (sockaddr *) &sin, &sinlen);
    if (cfd >= 0) { /* ... */ }
}

int main (int argc, char **argv) {
    // ...
    int lfd = inetsocket (SOCK_STREAM, your_port, INADDR_ANY);
    if (lfd < 0) fatal << "socket: " << strerror (errno) << "\n";
    if (listen (lfd, 5) < 0) fatal ("listen: %m\n");
    fdcb (lfd, selread, wrap (doaccept, lfd));
    amain ();
}
```

Remote procedure call

- **Abstract away network in distributed programs**
 - Idea: Distributed programming looks like function call
 - Reality: Can't abstract away everything (e.g., failure)
- **Next class: Sun RPC**
 - XDR defines structures that can be transmitted on the wire
 - RPC is simple layer that sends arg. structs and gets returns