CS144 – Introduction to Computer Networking

Instructor: Philip Levis
CAS: Kirian Abraham, Tahir Azim, Arda Kara, Behram Mistree, Harsh Nayyar
Section Leaders: Alex Quach, Pong Eksombatchai
cs144-aut1112-staff@stanford.edu

http://cs144.scs.stanford.edu/

Networks class

• Goal: Teach the concepts underlying networks
  - How do networks work? What can one do with them?
  - Give you a basic understanding of the Internet
  - Give you experience using and writing protocols
  - Give you tools to understand new protocols & applications

• Prerequisites:
  - CS110 or equiv; class assumes you are comfortable with C
  and gdb, some socket programming helpful (e.g., CS110 web server)

Administrivia

• All assignments are on the web page

• Text: Kurose & Ross, Computer Networking: A Top-Down Approach, 4th or 5th edition
  - Instructors working from 4th edition, either OK
  - Don’t need lab manual or Ethereal (used book OK)

• Syllabus on web page
  - Gives which textbook chapters correspond to lectures
    (Lectures and book topics will mostly overlap)
  - Extra (not required) questions for further understanding
  - Papers sometimes, to make concepts more concrete
    (Read the papers before class for discussion)
  - Subject to change! (Reload before checking assignments)

Online Resources

• Old school web page: http://cs144.stanford.edu

• We use Piazza for assignments: please send all assignment questions there
  - Link on http://cs144.stanford.edu
  - Piazza allows you to answer questions and rate answers

• Send all staff communication to staff list
  - Goes to whole staff, so first available person can respond
  - CCing list ensures we give students consistent information
  - Also, some of us get lots of email… much easier for us to prioritize a specific mailing list

Grading

• Exams: Midterm & Final

• Programming labs
  - 5 lab assignments implemented in C
  - If you are not comfortable with C and gdb they will be painful

• Two writing homework assignments
  - One question each week, pick 2
  - Opportunity to rewrite and resubmit based on feedback

• Administrative handout has details on grading, please put questions on Piazza

Labs

• Labs are due by the beginning of class
  - Lab 1: Stop & wait
  - Lab 2: Reliable transport
  - Lab 3: Static routing
  - Lab 4: Dynamic routing
  - Lab 5: NAT

• Most assignments due at start of Wednesday lecture
  - Late policy: can turn in late until 8:59PM that Friday, grade capped at 90%
  - Lab 1: Due at start of Monday lecture, late is 8:59PM on Wednesday
Section

- Place and time TBA
- Led by TAs and section leaders
- Practical assignment help and guidance
- This week: gdb tutorial

Why You Should Care About the Internet
(just 12 months)
Mobile Devices

Mobile Internet Rampin Faster than Desktop Internet Did – Apple Leading Charge

Morgan Stanley

Why You Should Take This Course

• The Internet is driving tremendous changes in the world
• It is continuously growing and evolving
  - Any facts you learn will inevitably be out of date
  - Learn general *principles* of networks
• Goal: Teach the underlying concepts of networks
  - How do networks work? What can one do with them?
  - Give you a basic understanding of the Internet
  - Give you experience using and writing protocols
  - Give you tools to understand new protocols & applications

Today’s Lecture

• Basic networking abstractions
  - Protocols
  - OSI layers and the Internet Hourglass
• Transport protocols: TCP and UDP
• Protocol performance tradeoffs
• Programming refresher for lab 1+2
  - Review of file descriptors
  - Some functions from the socket API
• Next lecture: basics of reliability and server socket programming

Networks

• What is a network?
  - A system of lines/channels that interconnect
  - E.g., railroad, highway, plumbing, communication, telephone, *computer*
• What is a *computer* network?
  - A form of communication network—moves information
  - Nodes are general-purpose computers
• Why study computer networks?
  - Many nodes are general-purpose computers
  - You can program the nodes
  - Very easy to innovate and develop new uses of network
  - Contrast: Old PSTN – all logic is in the core

Building blocks

• Nodes: Computers, dedicated routers, …
• Links: Coax, twisted pair, fibers, radio …
  (a) point-to-point
  (b) multiple access – every node sees every packet

From Links to Networks

• To scale to more nodes, use *switching*
  - nodes can connect multiple other nodes, or
  - Recursively, one node can connect multiple networks
Protocol layering

<table>
<thead>
<tr>
<th>Application</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Link Layer

- Can view network encapsulation as a stack
- A network packet (IP) from A to D must be put in link layer packets A to B, B to C, and C to D
  - Each layer produces packets that become the payload of the lower-layer’s packets
  - This is almost correct, but TCP/UDP “cheat” to detect certain errors in IP-level information like address

Layers (and lectures)

- Physical – sends individual bits (9, 13)
- Data link – sends frames, handles access control to shared media (e.g., coax) (9, 13)
- Network – delivers packets, using routing (4, 6, 7)
- Transport – demultiplexes, provides reliability & flow control (3, 5)
- Session – can tie together multiple streams (e.g., audio & video) (15)
- Presentation – crypto, conversion between representations (15)
- Application – what end user gets, e.g., HTTP (2, 11)

Addressing

- Each node typically has unique address
  - (or at least is made to think it does when there is shortage)
- Each layer can have its own addressing
  - Link layer: e.g., 48-bit Ethernet address (interface)
  - Network layer: 32-bit IP address (node)
  - Transport layer: 16-bit TCP port (service)
- Routing is process of delivering data to destination across multiple link hops
- Special addresses can exist for broadcast/multicast

Hourglass

- Many application protocols over TCP & UDP
- IP works over many types of network
- This is “Hourglass” philosophy of Internet
  - Idea: If everybody just supports IP, can use many different applications over many different networks
  - In practice, some claim narrow waist is now network and transport layers, due to NAT (lecture 10)

Internet protocol

- Most computer nets connected by Internet protocol
  - Runs over many physical networks
- Every host has a unique 4-byte IP address
  - E.g., www.ietf.org → 132.151.6.21
  - Given an IP address, the network routes a packet (lectures 4, 6, 7)
  - Next-gen IPv6 uses 16-byte host addresses (lecture 12)
- But how do you build something like the web?
  - Need naming (look up www.ietf.org) – DNS (lecture 8)
  - Need API for browser, server (CS110/this lecture)
  - Need demultiplexing within a host—E.g., which packets are for web server, which for mail server, etc.? (lecture 2)

"For thinks it has"
Inter-process communication

- Want abstraction of inter-process (not just inter-node) communication
- Solution: Encapsulate another protocol within IP

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 to processes on two different machines (lecture 3)
  - Handles congestion & flow control (lecture 5)

Uses of TCP

- **Most applications use TCP**
  - Easier interface to program to (reliability, lecture 3)
  - Automatically avoids congestion (don’t need to worry about taking down network, lecture 5)
- **Servers typically listen on well-known ports**
  - SSH: 22
  - Email: 25
  - Finger: 79
  - Web / HTTP: 80
- **Example**: Interacting with www.stanford.edu

Small request/reply protocol

- Client
  - request
- Server
  - reply

- Small message protocols typically dominated by latency

Large reply protocol

- Client
  - request
  - reply
- Server

- For bulk transfer, throughput is most important

Performance definitions

- **Throughput** – Number of bits/time you can sustain at the receiver
  - Improves with technology
- **Latency** – How long for message to cross network
  - Propagation + Transmit + Queue
  - We are stuck with speed of light . . . 10s of milliseconds to cross country
- **Goodput** – \( \frac{\text{TransferSize}}{\text{Latency}} \)
- **Jitter** – Variation in latency
- **What matters most for your application?**
  - We’ll look at network applications next lecture
Programming Sockets

- Book has Java source code
- CS144 is in C
  - C is the language of choice for low-level systems
  - Many books and internet tutorials
- Berkeley sockets API
  - Bottom-level OS interface to networking
  - Important to know and do once
  - Higher-level APIs build on them

Quick CS110 review: System calls

- System calls invoke code in the OS kernel
  - Kernel runs in a more privileged mode than application
  - Can execute special instructions that application cannot
  - Can interact directly with devices such as network card
- Higher-level functions built on syscall interface
  - printf, scanf, gets, etc. all user-level code

File descriptors

- Most IO done on file descriptors
  - Small integers referencing per-process table in the kernel
- Examples of system calls with file descriptors:
  - int open(char *path, int flags, ...);
    - Returns new file descriptor bound to file path
  - 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 written, -1 on error
    - (Never returns 0 if nbytes > 0)
  - int close (int fd);
    - Deallocates file descriptor (not underlying I/O resource)

Error returns

- What if syscall fails? E.g. open non-existent file?
  - Returns -1 (invalid fd number)
- Most system calls return -1 on failure
  - Always check for errors when invoking system calls
  - Specific kind of error in global int errno
    (But errno will be unchanged if syscall did not return -1)
- #include <sys/errno.h> for possible values
  - 2 = ENOENT “No such file or directory”
  - 13 = EACCES “Permission Denied”
- perror function prints human-readable message
  - perror (“initfile”);
    → “initfile: No such file or directory”

Sockets: Communication between machines

- Network sockets are file descriptors too
- 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/recv, sendto/recvfrom, and sendmsg/recvmsg (most general)
- 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

- Recall how TCP & UDP name communication endpoints
  - 32-bit IP address specifies machine
  - 16-bit TCP/UDP port number demultiplexes within host
  - 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 can be named by 5 components
  - Protocol (TCP), local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP
System calls for using TCP

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket – make socket</td>
<td>struct sockaddr { uint16_t sa_family; /* address family <em>/ char sa_data[14]; /</em> protocol-specific address <em>/ }; /</em> (may be longer than this) */</td>
</tr>
<tr>
<td>bind – assign address</td>
<td>int connect(int fd, const struct sockaddr *, socklen_t);</td>
</tr>
<tr>
<td>listen – listen for clients</td>
<td>int accept(int fd, struct sockaddr *, socklen_t);</td>
</tr>
</tbody>
</table>

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.

Socket address structures

- **Socket interface supports multiple network types**
- **Most calls take a generic sockaddr:**
  ```c
  struct sockaddr {
    uint16_t sa_family; /* address family */
    char sa_data[14]; /* protocol-specific address */
  };
  ```
- **Cast sockaddr * from protocol-specific struct, e.g.:**
  ```c
  struct sockaddr_in {
    short sin_family; /* = AF_INET */
    u_short sin_port; /* = htons (PORT) */
    struct in_addr sin_addr; /* 32-bit IPv4 address */
    char sin_zero[8];
  };
  ```

Dealing with address types [RFC 3493]

- **All values in network byte order (big endian)**
  - htonl converts 32-bit value from host to network order
  - ntohl converts 32-bit value from network to host order
  - ntohs/ntohs same for 16-bit values

- **All address types begin with family**
  - sa_family in sockaddr tells you actual type

- **But not all address types are the same size**
  - E.g., struct sockaddr_in6 is typically 28 bytes,
    yet generic struct sockaddr is only 16 bytes
  - So most calls require passing around socket length
  - Can simplify code with new generic sockaddr_storage big enough for all types (but have to cast between 3 types now)

Looking up a socket address w. getaddrinfo

```c
struct addrinfo hints, *ai;
int err;
memset (&hints, 0, sizeof (hints));
hints.ai_family = AF_UNSPEC; /* or AF_INET or AF_INET6 */
hints.ai_socktype = SOCK_STREAM; /* or SOCK_DGRAM for UDP */
err = getaddrinfo ("www.stanford.edu", "http", &hints, &ai);
if (err)
    fprintf (stderr, "%s
", gia_strerror (err));
else {
    /* ai->ai_family = address type (AF_INET or AF_INET6) */
    /* ai->ai_addr = actual address cast to (sockaddr *) */
    /* ai->ai_addrlen = length of actual address */
    freeaddrinfo (ai); /* must free when done! */
}
```

Address lookup details

- **getaddrinfo notes:**
  - Can specify port as service name or number (e.g., "80" or "http", allows possibility of dynamically looking up port)
  - May return multiple addresses (chained with ai_next field)
  - You must free structure with freeaddrinfo

- **Other useful functions to know about**
  - getnameinfo – Lookup hostname based on address
  - inet_ntop – convert IPv4 or 6 address to printable form
  - inet_pton – convert string to IPv4 or 6 address

EOF in more detail

- **Simple client-server application**
  - Client sends request
  - Server reads request, sends response
  - Client reads response

- **What happens when you’re done?**
  - Client wants server to read EOF to say request is done
  - But still needs to be able to read server reply – fd is not closed!
shutdow

- int shutdown (int fd, int how);
  - Shuts down a socket w/o closing file descriptor
  - how: 0 = reading, 1 = writing, 2 = both
  - Note: Applies to socket, not descriptor—so copies of
descriptor (through dup or fork affected)
  - Note 2: With TCP, can’t detect if other side shuts for reading

- Many network applications detect & use EOF
  - Common error: “leaking” file descriptor via fork, so not
closed (and no EOF) when you exit

---

Today’s Lecture

- Basic networking abstractions
  - Protocols
  - OSI layers and the Internet Hourglass
- Transport protocols: TCP and UDP
- Protocol performance tradeoffs
- Programming refresher for lab 1+2
  - Review of file descriptors
  - Some functions from the socket API
- Next lecture: applications (HTTP, BitTorrent, etc.)
  and server socket programming

---

Structure of Rest of Class

- IP and above (5 weeks)
  - Application layers
  - Network layer: IP and routing, multicast
  - Transport layer: TCP and congestion control
  - Naming, address translation, and content distribution
- Below IP (2 weeks)
  - Network address translation (NAT)
  - Link and physical layers
- Advanced topics (2 weeks)
  - Multimedia
  - Network coding
  - Security