

TCP and Sockets

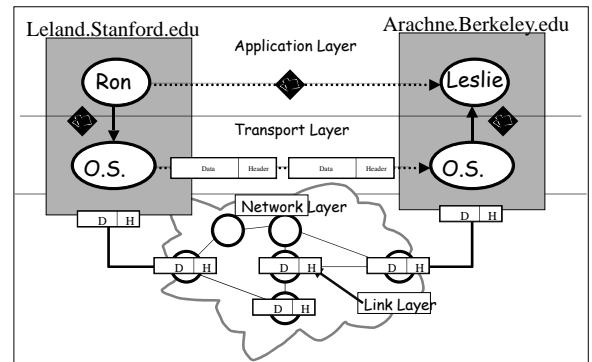
Lecture 3

cs193i – Internet Technologies

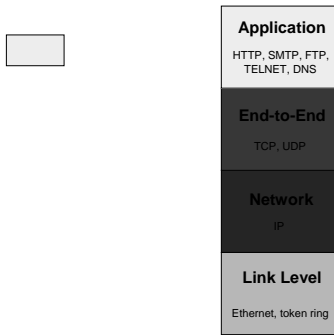
Summer 2004

Stanford University

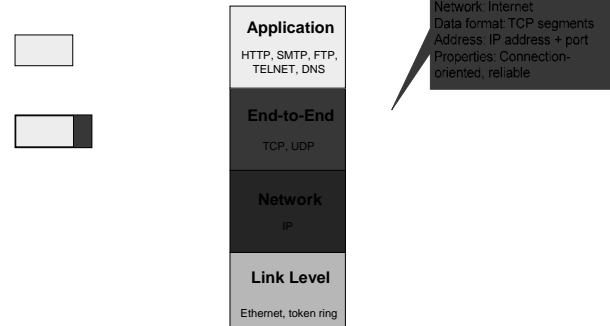
Sending a Message



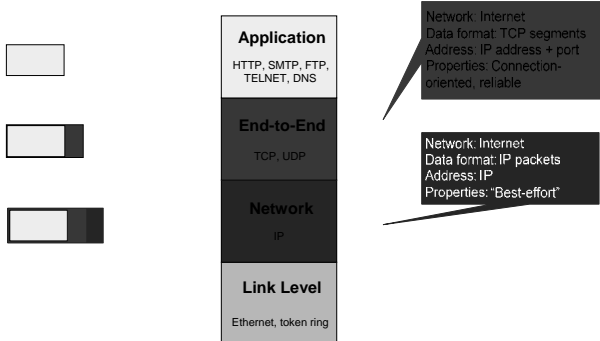
Protocol Stack



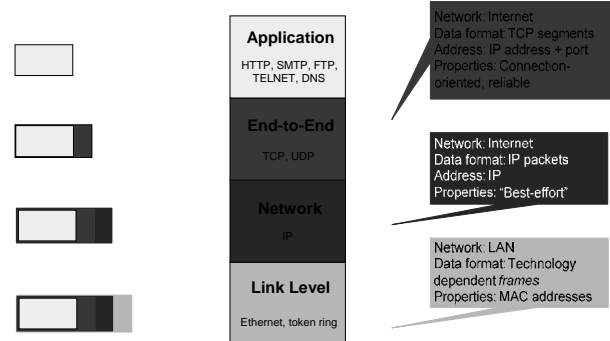
Protocol Stack



Protocol Stack



Protocol Stack

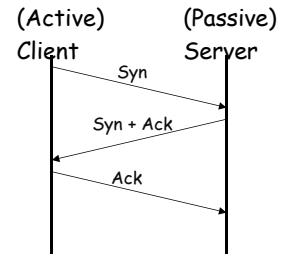


TCP

- Connection-Oriented
 - Port on “client” connects to port on “server”
- Reliable
 - 3-way handshake
- Byte-Stream
 - To application, looks like stream of bytes flowing between two hosts
- Flow Control
 - Prevents overrunning receiver / network capacity

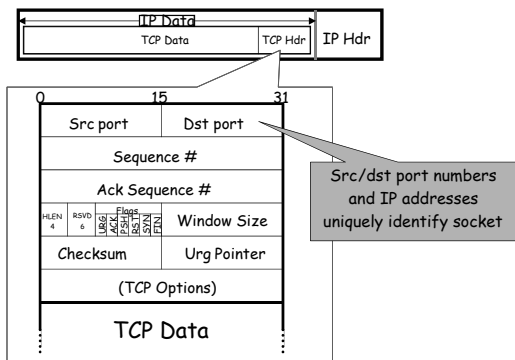
Establishing the Connection

- Client
 - Sequence # (x)
- Server
 - ACK (x+1)
 - Own sequence # (y)
- Client
 - ACK (y+1)
 - Sequence # (x+1)

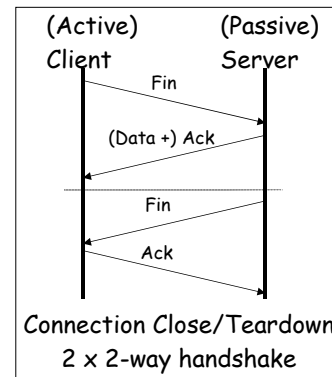


Connection Setup
3-way handshake

Maintaining the “Connection”

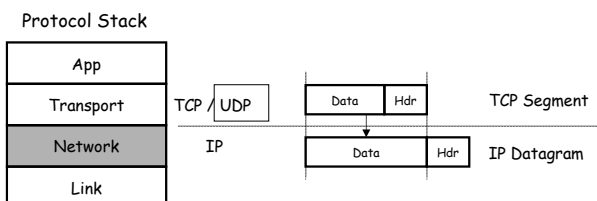


Terminating the Connection



Connection Close/Teardown
2 x 2-way handshake

Another Transport Layer Protocol: UDP



User Datagram Protocol (UDP)

- Characteristics
 - Connectionless, Datagram, Unreliable
- Adds only application multiplexing/demultiplexing and checksumming to IP
- Good for Streaming Media, Real-time Multiplayer Networked Games, VoIP

Summary

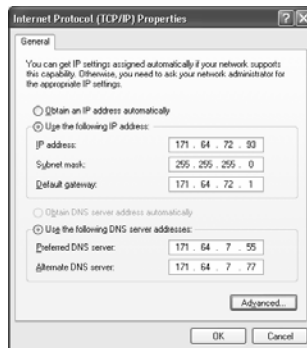
- IP is the basis of Internetworking
- TCP builds on top of IP
 - adds reliable, congestion-controlled, connection-oriented byte-stream.
- UDP builds on top of IP
 - allows access to IP functionality

Brief Return to Addressing

- Broadcast
 - Send message to everyone on LAN
- Address Resolution Protocol (ARP)
 - Broadcast “Who has IP address 171.64.64.250?”
 - Owner of IP address answers with LAN address

Addressing in Action

- IP Address
- Subnet Mask
 - Last 8 bits used for host
- Gateway
 - Local router to forward traffic to
- DNS server
 - Translates names to IP addresses



Domain Name Service (DNS)

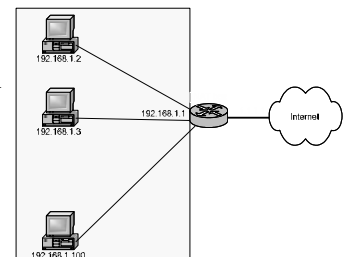
- Distributed database maps
 - host names --> numerical IP Address
- Hierarchical Namespace
- Top-level domain (root domain)
 - .com, .net, .org**
- Second-level domain
 - hotmail.com, stanford.edu**
- Sub domains
 - www.yahoo.com, movies.yahoo.com**

DNS and Routing

- Both are Hierarchical
- IP Routing Hierarchy is left to right
 - (128.12.132.29)
- DNS Hierarchy is right to left
 - (www.stanford.edu)
- Root name server & delegate name server
 - Backbone router & delegate router

Network Address Translation (NAT)

- Hosts share single IP address
- Hosts IP addresses in 192.168.*.*
- Router between LAN and Internet
 - IP address (1.1.1.1)
 - Translates to host address before forwarding



Five Minute Break

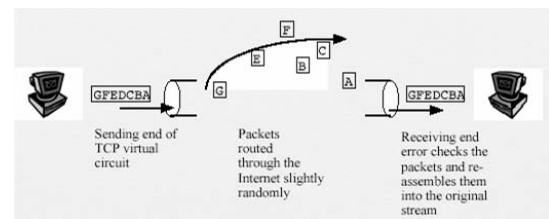
Sockets

- Basic Concepts & Important Issues
- Client Socket Code & Server Socket Code
- Client-Server Interaction

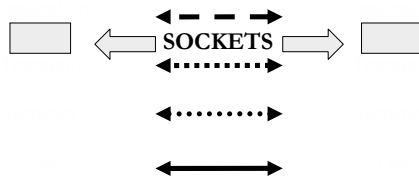
What is a Socket?

- Interface to the TCP Byte-Stream (2-way)
- Both sides think they are writing to Files!
- Example Code
 - `write(SOCK, "Hello\n");`
 - `print SOCK "Hello\n";`

TCP Byte Stream (Virtual Circuit)



Protocol Stack

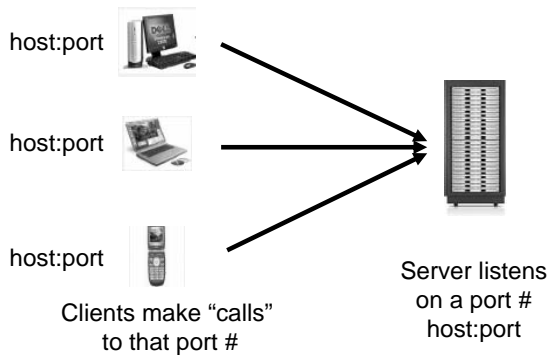


What is a Port?

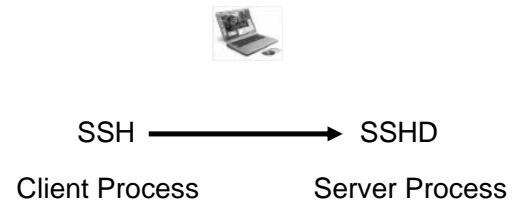
- Transport address to which processes can listen for connection requests
- Local to host: 1-65535 ports
- Well-known ports
 - Below 1024
 - Standard services
 - Only supervisor privileged enough to access

Examples
■ FTP: 21
■ HTTP: 80
■ TELNET: 23

Server and Client



Server and client can be the same machine!



Client Side

Establishing a Connection

- Establish Socket Connection
- Send & Receive Data
- Close Socket Connection

Remember how TCP is connection-oriented, and has three phases of a connection?

Once the connection is established...

- Protocol (RFC) dictates turn taking between Server and Client
- Write by saying in Perl:
`print SOCK "Hello\n";`
- Read by saying in Perl:
`$incoming_data = <SOCK>;`

Example Sockets in Perl

- `$ipaddr = inet_aton("www.yahoo.com");`
- `$sockaddr = sockaddr_in(80, $ipaddr);`
- `socket($sock, ...);`
- `connect($sock, $a);`
- `print $sock "Hi There!"`

Socket Example in C

- `struct sockaddr_in peer;`
`peer.sin_addr.s_addr = inet_addr("127.0.0.1");`
`peer.sin_port = htons(7500);`
`peer.sin_family = AF_INET;`
- `s = socket(AF_INET, SOCK_STREAM, 0);`
- `connect(s, (struct sockaddr *)&peer, sizeof(peer));`
- `send(s, "Hi There!", 9, 0);`

Demo Code in Java

- `myClient = new Socket("www.yahoo.com", 80);`
- `outputStream = new`
`PrintStream(myClient.getOutputStream());`
- `outputStream.println("Hello There!");`

Blocking vs. Non-Blocking

- Blocking Function Call
Waits if necessary
 - <SOCK> blocks if no data to read
OS will wake up the process to read
 - Also write blocks
(send faster than recipient can handle)
- Non-Blocking Function Call
Returns immediately
 - May need a while loop to check for data

Buffering

- Why is it good normally?
 - Batch the work for efficiency (Harddrives...)
 - Prevent Failures (CD Players, etc...)
- Concept of flushing the buffer (write to disk/network)
- Why is it a problem for networking?

Autoflush

- Automatically flush after every write
- use `FileHandle; # FileHandle module...`
....
`autoflush SOCK, 1; # set autoflush`

Other Issues

- Irregular Timing (CPU fast, mostly blocked)
- Irregular Sizing
- Line Endings
 - `\r\n` -- most common on Internet, oldest
 - `\n` -- Unix way (single char, nice)
 - `\n` may get remapped! Messes up portability.
 - In your code... really is `\015\012`

Setting Up a Socket Program

1. Hostname to IPAddr conversion
`$ip = inet_aton($hostname);`
2. Create Socket Address
`$sockaddr = sockaddr_in($port, $ip);`
3. Allocate Socket
`socket(SOCK, PF_INET, SOCK_STREAM, 0);`
4. Connect
`connect(SOCK, $sockaddr);`

Reading from a Socket

- `connect(SOCK, $sockaddr);`
`$line = <SOCK>;`
- `$line =~ s/\015\012//g;`
- Turn-taking via EOF
`while ($line = <SOCK>) {`

`}`
- OR `sysread(SOCK, str, length); ## efficient`

sysread vs <>

- Read bytes `sysread(SOCK, ...)`
- Read line by line `<SOCK>`
- Do things in larger chunks!
Just like in Buffering...

Writing to a Socket

- `print SOCK "Hello!\012";`
- OR `syswrite(SOCK, str, length);`

Close the Socket

- `close(SOCK); ## sends EOF to other end`
- Other side sees all data, then EOF... then quits
- Consider it an end “marker” that is sent across

Server Side

Server Listens on Port

- **Create** the Socket as before
- **Bind** socket
bind(...) associates port & socket
- **Listen** at a port
Does not block.
- **Accept** an incoming connection
(so server must sit on the Internet all day)
This call blocks! Returns when client connects...

Server Listens on Port

- Look at Client Address (caller id)
- Read & Write
- Close
- Loop and listen again...

Perl Server Setup Example

```
sub CreateServerSocket {  
    my($sock, $port) = @_;  
    socket($sock, PF_INET, SOCK_STREAM, 0);  
    setsockopt($sock, SOL_SOCKET,  
        SO_REUSEADDR, pack("l",1));  
    bind($sock, sockaddr_in($port, INADDR_ANY));  
    autoflush $sock, 1;  
    return("");  
}
```

Listen and Accept

- listen (SERVER, ...);
- accept (CLIENT, SERVER);
autoflush CLIENT, 1;

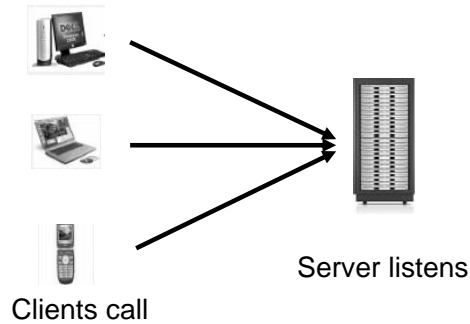
Server Pseudo-code

```
$serverport = 3456;  
CreateServerSocket(SERVER, $serverport);  
listen(SERVER,...);  
while($clientaddr = accept(CLIENT, SERVER))  
{  
    $incomingline = <CLIENT>;  
    print CLIENT "$outgoingline\n";  
    close(CLIENT);  
}
```

Telnet Trick

- Many Services just use ASCII text dialog between server and client
- > telnet host port
- Bottom Line: Text based protocols are **easier to debug** because their state is **visible** and **accessible**

Design of a Chat Room Server/Client



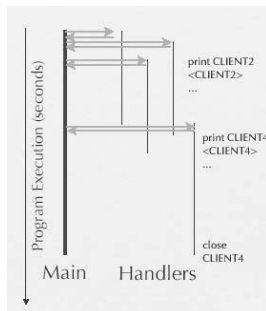
Chat Server Design

- **Server State**
 - Who is connected
 - Max clients connected
- **Server Abilities**
 - Accept a New Connection
 - Close a Connection
 - Send Message from A --> B
 - Send Message from A --> All

Chat Server Design

- **Main Thread**
 - Handles connections
 - Calls handler threads
- **Handler Threads**

```
<CLIENT>
print CLIENT "Hi!\n";
close CLIENT;
```



Chat Client Design

- **Client State**
 - Server's IP address : port
 - Who is connected to server
- **Client Abilities**
 - Connect to server
 - Send message to A (thru server handler)
 - Send message to all (thru server handler)
 - Leave server
 - (e.g. print SOCK "BYE", close SOCK)

Peer-to-Peer

- **No fixed Server & Client**
 - Anybody can be Server or Client at any time
- **Example P2P software**
 - File Sharing (Bittorrent, FastTrack, Gnutella)

Socket Summary

- **Client Socket**
 - Setup/Create, Connect, Read/Write, Close
- **Server Socket**
 - Setup/Create, Bind, Listen, Accept, Read/Write, Close
- **It's just a Programmer's Application Programming Interface (API) to TCP or UDP on top of IP**