CSC 4304 - Systems Programming
Fall 2008

LECTURE - XXII
NETWORK PROGRAMMING

Tevfik Koşar

Louisiana State University
December 2nd, 2008

The Fundamentals

- The Computer Systems Research Group (CSRG) at the University of California Berkeley gave birth to the Berkeley Socket API (along with its use of the TCP/IP protocol) with the 4.2BSD release in 1983.
  - A Socket is comprised of:
    - a 32-bit node address (IP address or FQDN)
    - a 16-bit port number (like 7, 21, 13242)
  - Example: 192.168.31.52:1051
    - The 192.168.31.52 host address is in "IPv4 dotted-quad" format, and is a decmial representation of the hex network address 0xc0a81f34

Port Assignments

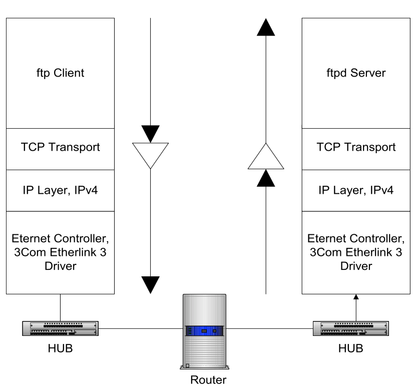
- Ports 0 through 1023 are reserved, *privileged* ports, defined by TCP and UDP well known port assignments
- Ports 1024 through 49151 are ports *registered* by the IANA (Internet Assigned Numbers Authority), and represent second tier common ports (socks (1080), WINS (1512), kermit (1649))
- Ports 49152 through 65535 are *ephemeral* ports, available for temporary client usage

Common Protocols

Application	ICMP	UDP	TCP
Ping	✓		
Traceroute	✓		
DHCP		✓	
NTP		✓	
SNMP		✓	
SMTP			✓
Telnet			✓
FTP			✓
HTTP			✓
NNTP			✓
DNS		✓	✓
NFS		✓	✓
Sun RPC		✓	✓

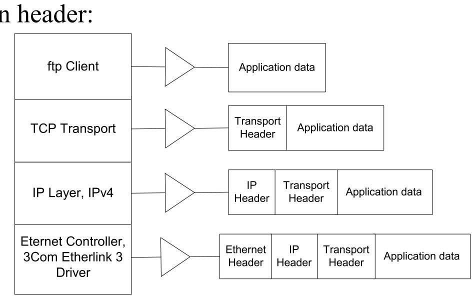
ICMP: Internet Control Message Protocol
UDP: User Datagram Protocol
TCP: Transmission Control Protocol

Protocol Communication



Data Encapsulation

- Application puts data out through a socket
- Each successive layer wraps the received data with its own header:



The Hardware (Ethernet) Layer

- Responsible for transferring frames (units of data) between machines on the same physical network

Preamble (bit sequence)	Destination Address (192.32.65.1)	Source Address (192.32.63.5)	Packet type (magic number for protocol: 0x800 = IP, 0x6003 = Decnet, 0x809B = Appletalk)	Datagram (THE DATA) (up to 12k bits)	Cyclic Redundancy Check
64 bits	48 bits	48 bits	16 bits	variable	32 bits

The IP Layer

- The IP layer allows packets to be sent over gateways to machines not on the physical network
- Addresses used are IP addresses, 32-bit numbers divided into a network address (used for routing) and a host address
- The IP protocol is connectionless, implying:
  - gateways route discrete packets independently and irrespective of other packets
  - packets from one host to another may be routed differently (and may arrive at different times)
  - non-guaranteed delivery

IP Datagram Format

- Packets may be broken up, or *fragmented*, if original data is too large for a single packet (Maximum Transmission Unit is currently 12k bits, or 1500 Bytes)
- Packets have a Time To Live, number of seconds/ rounds it can bounce around aimlessly among routers until it's killed

Preamble	Length of data	Fragmentation Information (if it's too big for an ethernet frame buffer)	Time To Live	Protocol (TCP, UDP)	Checksum	Source Address (192.32.63.5)	Destination Address (192.32.65.1)	Options	Datagram (THE DATA) (up to 12k bits)
2 bytes	2 bytes	4 bytes	1 byte	1 byte	2 bytes	4 bytes	4 bytes	variable	variable

The Transport Layer

- Unix has two common transports
  - User Datagram Protocol (UDP)
    - record protocol
    - connectionless, broadcast
    - *Metaphor*: Postal Service
  - Transmission Control Protocol (TCP)
    - byte stream protocol
    - direct connection-oriented

Transport Layer: UDP

- Connectionless, in that no long term connection exists between the client and server. A connection exists only long enough to deliver *a single packet* and then the connection is severed.
- No guaranteed delivery ("best effort")
- Fixed size boundaries, sent as a single "fire and forget message". Think *announcement*.
- No built-in acknowledgement of receipt

Transport Layer: UDP

- No built-in order of delivery, random delivery
- Unreliable, since there is no acknowledgement of receipt, there is no way to know to resend a lost packet
- Does provide checksum to guarantee integrity of packet data
- *Fast* and Efficient

Transport Layer: TCP

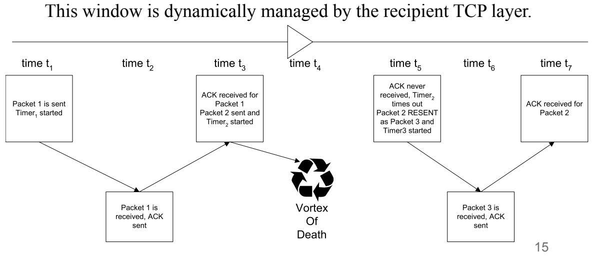
- TCP *guarantees delivery* of packets in order of transmission by offering acknowledgement and retransmission: it will automatically resend after a certain time if it does not receive an ACK
- TCP promises *sequenced delivery* to the application layer, by adding a sequence number to every packet. Packets are reordered by the receiving TCP layer before handing off to the application layer. This also aides in handling "duplicate" packets.

Transport Layer: TCP

- Pure stream-oriented connection, it does not care about message boundaries
- A TCP connection is full duplex (bidirectional), so the same socket can be read and written to (cf. half duplex pipes)
- Provides a checksum that guarantees packet integrity

TCP's Positive Acknowledgement with Retransmission

- TCP offers acknowledgement and retransmission: it will automatically resend after a certain time if it does not receive an ACK
- TCP offers *flow control*, which uses a "sliding window" (in the TCP header) will allow a *limited* number of non-ACKs on the net during a given interval of time. This increases the overall bandwidth efficiency. This window is dynamically managed by the recipient TCP layer.



Reusing Addresses

- Local ports are locked from rebinding for a period of time (usually a couple of minutes based on the TIME_WAIT state) after a process closes them. This is to ensure that a temporarily "lost" packet does not reappear, and then be delivered to a *reincarnation* of a listening server. But when coding and debugging a client server app, this is bothersome. The following code will turn this feature off:

```
int yes = 1;
server = socket(AF_INET, SOCK_STREAM, 0);

if (setsockopt(server, SOL_SOCKET,        SO_REUSEADDR,
&yes, sizeof(int)) < 0)
{
  perror("setsockopt SO_REUSEADDR");
  exit(1);
}
```


TCP Header Format

- Source and Destination addresses
- Sequence Number tells what byte offset within the overall data stream this segment applies
- Acknowledgement number lets the recipient set what packet in the sequence was received OK.

Source Port	Destination Port	Sequence Number	Acknowledgement Number	Flags	Window Size	Checksum	Urgent Pointer	Options	Datagram (THE DATA) (up to 12k bits)
2 bytes	2 bytes	4 bytes	4 bytes	2 bytes	2 bytes	2 bytes	2 bytes	variable	variable

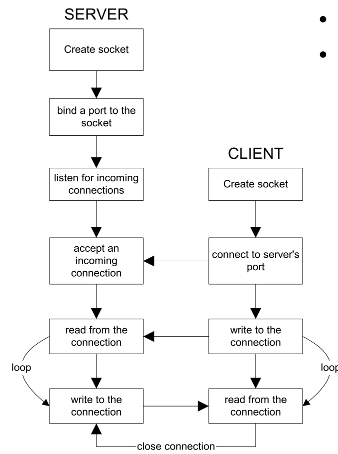
Creating a Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- domain is one of the *Address Families* (AF_INET, AF_UNIX, etc.)
- type defines the communication protocol semantics, usually defines either:
  - SOCK_STREAM: connection-oriented stream (TCP)
  - SOCK_DGRAM: connectionless, unreliable (UDP)
- protocol specifies a particular protocol, just set this to 0 to accept the default (PF_INET, PF_UNIX) based on the domain

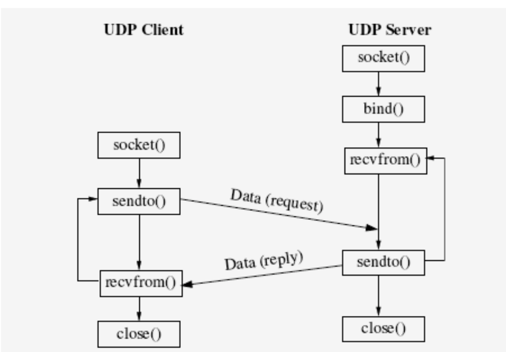
UDP Clients and Servers

- Connectionless clients and servers create a socket using SOCK_DGRAM instead of SOCK_STREAM
- Connectionless servers do not call listen() or accept(), and *usually* do not call connect()
- Since connectionless communications lack a sustained connection, several methods are available that allow you to *specify a destination address with every call*:
  - sendto(sock, buffer, buflen, flags, to_addr, tolen);
  - recvfrom(sock, buffer, buflen, flags, from_addr, fromlen);
- *Examples: daytimeclient.c, mytalkserver.c, mytalkclient.c*



- TCP Client-Server view
- Connection-oriented socket connections

UDP Socket Functions



Creating UDP Sockets

- To create a UDP socket on port 1234:

```
int fd, err;
struct sockaddr_in addr;

fd = socket(AF_INET,SOCK_DGRAM,0);
if (fd<0) { ... }

addr.sin_family      = AF_INET;
addr.sin_port        = htons(1234);
addr.sin_addr.s_addr = htonl(INADDR_ANY);

err = bind(fd, (struct sockaddr *) &addr, sizeof(struct sockaddr_in));
if (err<0) { ... }
```

- For historic reasons, you are obliged to explicitly cast your `struct sockaddr_in *` into a `struct sockaddr *`

Sending a UDP Datagram

- `sendto()` is used to send a UDP datagram:

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int fd, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

  - `fd`: the socket descriptor
  - `msg`: the message to be sent
  - `len`: the length of the message
  - `flags`: options, usually set to 0
  - `to`: the destination address (IP address **and** port number!)
  - `tolen`: the size of a `struct sockaddr_in`
  - Return value: the number of characters sent, or -1 in case of an error

sendto () example

```
int fd;
char msg[64];
int err;
struct sockaddr_in dest;

strcpy(msg,"Hello, world!");

fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd<0) { ... }

dest.sin_family      = AF_INET;
dest.sin_port        = htons(1234);
dest.sin_addr.s_addr = inet_addr("130.37.193.13");

err = sendto(fd, msg, strlen(msg)+1, 0, (struct sockaddr*) &dest,
             sizeof(struct sockadddr_in));
if (err<0) { ... }
```

Receiving a UDP Datagram

- `recvfrom()` **blocks the program** until a UDP datagram is received

```
#include <sys/types.h>
#include <sys/socket.h>
int  recvfrom(int fd, void  *buf, size_t len, int flags,
            struct sockaddr *from, socklen_t *fromlen);
```

  - `fd`: the socket descriptor
  - `buf`: a buffer where the message will be copied
  - `len`: the size of the buffer
  - `flags`: usually set to 0
  - `from`: a structure where the origin address of the datagram will be copied
  - `fromlen`: a pointer to an integer **containing the size of** `from`
  - Return value: number of bytes received, or -1 in case of an error

recvfrom() example

```
/* the socket is created and bound to a well-known port */

char msg[64];
int len, flen;
struct sockaddr_in from;

flen = sizeof(struct sockaddr_in);
len  = recvfrom(fd, msg, sizeof(msg), 0,
            (struct sockaddr*) &from, &flen);
if (len<0) { ... }
printf("Received %d bytes from host %s port %d: %s", err,
       inet_ntoa(from.sin_addr), ntohs(from.sin_port), msg);
```


How to handle timeouts?

- All `recvfrom()`-like functions are blocking
  - Once you started reading, you cannot return until some data has been read
  - (or you can read in non-blocking mode, but that's another story)
- Imagine that you expect a datagram, but you want to set a timeout
  - If you call `recvfrom`, you will be blocked until a packet arrives
- You need a way to wait until some data arrives **or** the timeout expires
  - The `select()` function can do that for you

select()

- `select()` monitors one (or more) file descriptor(s)
  - It blocks the program until one of them is ready for reading or writing, or a timeout expires

```
#include <sys/select.h>
int select(int n, fd_set *readfds, fd_set *writefds,
          fd_set *exceptfds, struct timeval *timeout);
```

  - `n`: the highest-numbered file descriptor, plus 1
  - `readfds`: a list of file descriptors to monitor for reading
  - `writefds`: a list of file descriptors to monitor for writing
  - `exceptfds`: a list of file descriptors to monitor for exceptions
  - `timeout`: a duration after which `select()` returns anyway. Pass a NULL pointer for no timeout.
  - Return value: the number of (i.e., **how many**) descriptors ready for I/O, or 0 in case of timeout, or -1 in case of an error

select() example

```
int            fd, nb;
fd_set         read_set;
struct timeval timeout;

FD_ZERO(&read_set);        /* Clear the read_set */
FD_SET(fd, &read_set);     /* Wait until fd is ready for reading */
timeout.tv_sec  = 0;       /* 0 seconds */
timeout.tv_usec = 500000;  /* 500000 micro-seconds = 0.5 second */

nb = select(fd+1, &read_set, NULL, NULL, &timeout);
if (nb<0)  { /* Error */  }
if (nb==0) { /* Timeout */ }
if (FD_ISSET(fd,&read_set)) {
  recvfrom(fd, ...);
}
```


Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J.Kim (KAIST), and J. Schaumann (SIT).