Internet Programming & Protocols Lecture 4

UDP

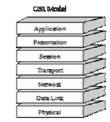
BSD Sockets

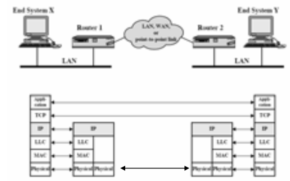
Client/servers

Assignment 1

The Internet protocols

- Physical/data link layer: Ethernet
- Network layer: IP
- Transport layer: ICMP/UDP/TCP
- Session/presentation: sockets/XDR
- Application: http/mail/ssh





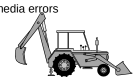
Ethernet	IP	TCP/UDP	Application

Traceroute anomalies

- Some routers (firewalls) may not send back the ICMP
- You may see RTT shorter to N+1 than to N
  - Routers may treat TTL expiration and ICMP at lower "priority" than forwarding packets
- Route could be changing while you're running traceroute
- Traceroute doesn't say anything about the return path
  - See "traceroute servers" on the internet to explore reverse paths

Things that slow us down …

- **Physical layer**
  - Loose connectors
  - RF interference
  - Collisions
  - Slow media or media errors
  - Speed of light
  - Backhoe
- **Link layer**
  - Half/full duplex mismatch
  - CRC errors
  - Exponential backoff
  - Packet reordering
  - NIC queues (txquelen)
  - Device (NIC) Driver software
    - interrupts



- **Network layer**
  - Fragmentation
  - Long routes
  - Slow links
  - Congestion
  - queue overflows (drops)
  - Synchronous routing updates?
  - Packet reordering (route/Juniper)
  - Software implementations/bugs
  - Firewalls/encryption
    - Block ports, ICMP
    - Examine/modify packets

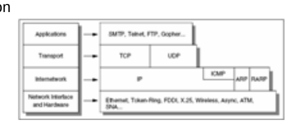
Encapsulation overhead:
just handling all the layering
extra bits in headers

Concept Collection

- **Best effort**
- **Bit error rate**
- **Checksums**
- **CIDR**
- **CSMA/CD**
- **fragmentation**
- **Packet switching vs circuit-based**
- **Layers/encapsulation**
- **MTU**
- **Network mask**
- **promiscuous**
- **Routing**
- **RTT**
- **Subnets/supernets**
- **Switch vs hub**
- **TTL**

Transport layer

- end-to-end services to application
- API (BSD sockets, TLI)
- flow control
- error recovery
- ICMP, UDP, TCP
  - ICMP ping, traceroute
  - TCP ssh, www, ftp, mail, telnet, chat, print, finger, X…
  - UDP ntp/time, NFS, DNS, audio/video, RPC, snmp, DHCP

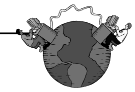


Ethernet	IP	TCP/UDP	Application

User Datagram Protocol (UDP)

- Defined in RFC 768
- connectionless (datagram)
- Lightweight – good for query/response
- 16-bit port (service number)
  - echo(7), DNS(53), bootp(68),ntp(123), snmp(160), NFS, RPC,netbios(137)
- unreliable (lost, damaged, duplicated, delayed, out of sequence) ☹
  - Same reliability as IP
  - If you want reliable UDP, application (YOU) must provide it!
- optional checksum
- supports broadcast and multicast (audio/video streaming)
  - Broadcast only within local ‘subnet’
  - Multicast local and wide area (awkward)
    - Uses IP class D addresses (map to special Ethernet addresses)
    - Ether NIC can be told which multicast addresses to “accept”

Why the net is unreliable



- Packets may be lost
  - Routing loops (TTL expires)
  - Insufficient buffers (routers, receiving host, switch)
  - Errors on the wire (link layer drops packet, e.g. CRC failure)
- Packets may be corrupted
  - Some app’s (NFS) don’t use UDP checksum for speed
  - Usually link layer CRC will catch mangled bits
- Packets may be delayed or arrive out of order
  - Each packet could go by a different route
  - Delays due to queuing at routers
- Packets may be duplicated
  - Rare but possible – retransmissions, routing loops

Recall that IP is a best-effort protocol … no guarantees

Network programming

- API needs to provide a way to “address” the remote application
  - For IP this means providing an IP address and a “port” number
  - Converting host “names” to an IP address is provided by the API (DNS)
- API needs to provide a way to send and receive a “message”
  - UDP is message based (datagram), connectionless
  - TCP is stream based (continuous stream of bytes)
    - You may not receive as many bytes as you request!
    - Connection-oriented and reliable
- Various programming paradigms
  - BSD sockets (ugly, so wrapper routines are often provided)
  - Classes/methods for C++/Java/Perl/Python
  - Incompatible data representations? (integer, float, byte-order)
  - More abstract: RPC or JAVA RMI
- Classical client/server coding

netfd = netopen(“host.com”,port)
read(netfd,buffer,length)
netfd = netlisten(port)

Client/Server

client

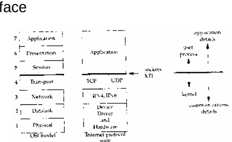
- user activated
- connects to well-known address
- sends/receives data
- closes connection
- non-privileged
- concurrency provided by OS

server

- activated by system
- runs forever (awaits requests)
- usually privileged
- worry about security
- handle multiple requests either **iteratively** or **concurrently**
- iterative servers for fast, single-response requests (e.g., time)
- concurrent servers usually *fork*() (e.g. httpd), or asynchronous I/O (*select*())
- On UNIX ***inetd*** is master server

BSD sockets

- UNIX (and Windows) transport layer interface
- API, subroutine library
- no standards (BSD is de facto)
- supports multiple protocol families
  - TCP/IP, XNS, UNIX, OSI, IPv6, ATM, raw
  - flexibility is paid for in complexity



- mixture of filling data structures and function calls ☹
  - Data structures contain data in network byte order
  - Funky struct’s require casts and length
- supports I/O abstraction
  - like reading/writing to file
  - but can't read what you write
  - read’s can block if no data is available!
  - write can complete, but it doesn’t mean receiver has read data
  - Full duplex – both sides can be reading and writing
  - Concurrency via forks, threads, select, asynch I/O

socket calls

- **socket()** get a socket descriptor for given protocol family and type
- **bind()** associate name (address/port, etc.) with a server (usually) socket
- **connect()** client establishes a connection to a server
- **listen()** connection-oriented server tells system it’s going to be passive.
- **accept()** server accepts incoming connection request and creates a new socket
- **close()** will try to deliver any unsent data
- Data transfers with **read(), write(), send(), recv()** or connectionless **sendto(), recvfrom()**

More functions

- Functions to handle integer byte order
  - ntohl() htonl() ntohs() htons() (sparc vs intel)
  - Your application may need to worry about other data (floating point)
- Functions to handle address-hostname conversions
- Functions to modify connection behavior (setsockopt())
- Functions to manage error reporting (perror())
- Functions to manage timeouts (alarm())
- Functions to measure performance (wall clock time)
- Functions to manage asynchronous IO (select()/fork()/threads)
- Functions to manage asynchronous events (signal())

converting IP addresses

- Convert IP address to/from ASCII

```
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in)
int inet_aton(const char *s, struct in_addr *a)
in_addr_t inet_addr(char *string)
```

- **inet_ntoa() not re-entrant**
- **inet_aton() replaced inet_addr() because -1 is legit (255.255.255.255)**

hostname to IP address

```
#inlclude <netdb.h>
struct  hostent *him;          /* host table entry */
char *host;  /* ascii IP address or hostname */
 …
if ( (inaddr = inet_addr(host)) != -1) {
              sin.sin_family = AF_INET;
              bcopy((char *) &inaddr, (caddr_t)&sin.sin_addr,sizeof(inaddr));
       }else {
              if ((him = gethostbyname(host)) == NULL) {
              fprintf(stderr, "uvdelay: Unknown host %s\n", host);
              return(-1);
              }
              sin.sin_family = him->h_addrtype;
              bcopy(him->h_addr, (caddr_t)&sin.sin_addr, him->h_length);
       }
```

•Probably should use memcpy() instead of older bcopy()

•gethostbyname() often results in DNS packets (more later)

hostent struct

Filled in by gethostbyname() with info from name server

```
struct hostent
{
    char *h_name;              /* Official name of host.  */
    char **h_aliases;          /* Alias list.  */
    int h_addrtype;            /* Host address type.  */
     int h_length;             /* Length of address.  */
    char **h_addr_list;        /* List of addresses from name server.  */
#define h_addr  h_addr_list[0]  /* Address, for backward compatibility.  */
};
```

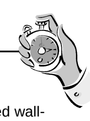

UNIX signals

- Asynchronous event handling (software “interrupts”)
- Messy – OS variations, BSD semantics, POSIX semantics
  - Are interrupted system calls restarted or terminated with an error?
  - What is default action for a given signal?
  - What if more signals occur while I’m handling a signal …?
- For assignment 3, you must handle alarm() signal and ctrl-C
- signal() establishes a handler (function) for specified signal

```
#include <signal.h>
void ding() { return;}
void ctrlc()
{
    printf(“this is goodbye\n”);
    exit(0);
}
…
    signal(SIGINT, ctrlc);  /* handle ctrl-c */
    signal(SIGALRM, ding);   /* handle alarm */
```

SIGINT
SIGALRM
SIGCHLD
SIGSEGV
SIGPIPE
SIGFPE
SIGUSR1
…

timing



- You can measure CPU time (process time) or wall-clock time
- For network measurements, we’re mainly concerned with elapsed wall-clock time

```
#include <sys/time.h>
double secs()
{
       struct timeval ru;
       gettimeofday(&ru, (struct timezone *)0);
       return(ru.tv_sec + ((double)ru.tv_usec)/1000000);
}

…
         double start, elapsed;
         start = secs();
         …
         elapsed = secs() – start;
```

socket()

int socket(family, type, protocol)

- returns a socket descriptor which is then used in read/write/close
- family: AF_UNIX, AF_INET, AF_NS, AF_INET6
  - (actually should be PF_UNIX etc.)
- type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- protocol: usually 0
- fails: bad args, no fd's/memory
- just sets up kernel data structures
- You need

#include <sys/types.h>

#include <sys/socket.h>

Socket data structures

```
/* sys/socket.h */
struct sockaddr {
    u_short sa_family;   /* address family */
    char    sa_data[14]; /* up to 14 bytes of direct address */
}


/* netinet/in.h */
struct sockaddr_in {
    short   sin_family;  /* AF_INET*/
    u_short sin_port;     /* network byte order */
    struct  in_addr sin_addr;     /* network byte order */
    char    sin_zero[8];
};
```

•sockaddr is generic struct used in function calls.

•sockaddr_in is internet socket struct that is filled by program

•Other socket struct's (of differing size) are PF_UNIX, PF_APPLETALK, PF_NETLINK, PF_IPX, PF_ATMPVC,…

bind()

int bind(sockfd, struct sockaddr *local, lth)

- binds local address and port to sockfd
- user fills struct sockaddr_in first providing port number
- required for server
- optional for client (usually not done by client)
- system will supply local address if client doesn't do bind
- lth of structure is required since struct sockaddr is different size for each protocol
- failures: bad args, port in use

port numbers

- Port numbers (UDP/TCP) provide a "process" address
  - Destination address, protocol (UDP or TCP), and port number define endpoint
  - Port number allows OS kernel to pass packets to appropriate process
  - For server process, bind() requests a port from the OS
  - In UNIX, ports < 1024 privileged
  - Well known (pre-defined) ports (services) listed in /etc/services
- bind() will fail if another server program on the machine is using the port
- bind() with port value of 0 tells OS to assign the port number
- bind() is optional for client (OS will assign a port number)

Well known UDP ports: echo (7), dns(53), bootp(67/68), ntp(123), netbios(138), snmp(167), ISAKMP(500)

getsockname()

Server can also let system assign port and use getsockname() to find out what port was assigned

```
struct sockaddr_in serv_addr;
serv_addr.sin_port        = htons(port);

if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
   perror("server: can't bind local address");
```

becomes

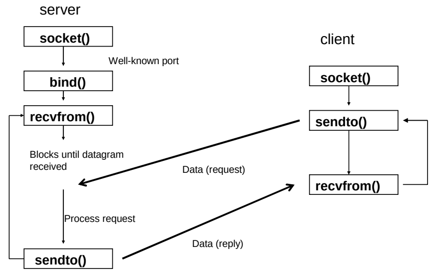
```
struct sockaddr_in cli_addr;

serv_addr.sin_port = 0;

if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
    perror("server: can't bind local address");
clilen = sizeof(cli_addr);
getsockname(sockfd, (struct sockaddr *) &cli_addr, &clilen);
port = ntohs(cli_addr.sin_port);
```


UDP data transfer calls

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int s, const void *msg, int len,  unsigned  int flags, const struct sockaddr
    *to, int tolen);
int recvfrom(int s, void *buf, int len, unsigned int flags, struct sockaddr *from, int
    *fromlen);
```

- can send and receive 0 bytes (no EOF)
- Data is moved in datagrams (messages)
- flags usually 0
- recvfrom() may never complete ☹
- recvfrom() returns length of packet accepted or -1
- If incoming packet is bigger than "int len" in recvfrom, you get only len bytes
- Sockaddr arg in sendto holds the destination address and port
- sockaddr arg in recvfrom will hold the "return address"
- Check return values from network functions!
  - failures: too big, buffs full, interrupted

UDP client/server



Example: client reads from tty and sends to server

server echos back whatever it receives

udpcli.c

```
main(argc, argv)
int   argc;
char  *argv[];
{
     int         sockfd;
     struct sockaddr_in   cli_addr, serv_addr;

     bzero((char *) &serv_addr, sizeof(serv_addr));
     serv_addr.sin_family      = AF_INET;
     serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
     serv_addr.sin_port        = htons(SERV_UDP_PORT);
     if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
          perror("client: can't open datagram socket");
     dg_cli(stdin, sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
     close(sockfd);
     exit(0);
}
```


udpcli.c

```
dg_cli(fp, sockfd, pserv_addr, servlen)
FILE      *fp;
int       sockfd;
struct sockaddr   *pserv_addr;   /* ptr to sockaddr_XX */
int servlen;   /* actual sizeof(*pserv_addr) */
{
     int n,fromlen;
     struct sockaddr from;
     char  sendline[MAXLINE], recvline[MAXLINE + 1];

     while (fgets(sendline, MAXLINE, fp) != NULL) {
        n = strlen(sendline);
        if (sendto(sockfd, sendline, n, 0, pserv_addr, servlen) != n)
          perror("dg_cli: sendto error on socket");
        fromlen = sizeof(from);  /* +++ */
        n = recvfrom(sockfd, recvline, MAXLINE, 0,&from,&fromlen);
        if (n < 0)perror("dg_cli: recvfrom error");
        recvline[n] = 0;  /* null terminate */
        fputs(recvline, stdout);
     }

     if (ferror(fp))perror("dg_cli: error reading file");
}
```


udpserv.c

```
main(argc, argv)
int   argc;
char  *argv[];
{
     int         sockfd;
     struct sockaddr_in   serv_addr, cli_addr;

     if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        perror("server: can't open datagram socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     serv_addr.sin_family      = AF_INET;
     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
     serv_addr.sin_port        = htons(SERV_UDP_PORT);

     if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
        perror("server: can't bind local address");

     dg_echo(sockfd, (struct sockaddr *)&cli_addr, sizeof(cli_addr));

}
```


dgecho.c

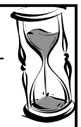
```
dg_echo(sockfd, pcli_addr, maxclilen)
int       sockfd;
struct sockaddr   *pcli_addr; /*  sockaddr_XX structure */
int       maxclilen;  /* sizeof(*pcli_addr) */
{
     int   n, clilen;
     char  mesg[MAXMESG];

     for ( ; ; ) {
        clilen = maxclilen;
        n = recvfrom(sockfd, mesg, MAXMESG, 0, pcli_addr, &clilen);
        if (n < 0) perror("dg_echo: recvfrom error");

        if (sendto(sockfd, mesg, n, 0, pcli_addr, clilen) != n)
          perror("dg_echo: sendto error");
    }
}
```


UDP and connect()

- If your UDP client is only going to one server, then connect() will cause your first read() to fail if service is not available. (ICMP port unreachable)
- use of connect() permits you to use read/write but have to modify use of send/recv/sendto/recvfrom

```
...
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = server->h_addrtype;
   bcopy(server->h_addr, (caddr_t)&serv_addr.sin_addr,server->h_length);
   serv_addr.sin_port        = htons(port);
   sockfd = socket(AF_INET, SOCK_DGRAM, 0);
#ifdef CONNECT
   connect(sockfd,&serv_addr,sizeof(serv_addr));
   sendto(sockfd,buff,lths[i],0,NULL,0);
#else
   sendto(sockfd,buff,lths[i],0,&serv_addr,sizeof(serv_addr));
#endif
```

UDP and timeouts



- Application (YOU) must “worry” about lost packets

```
#include <errno.h>
...
void ding(int signo) { return;}  /* handle alarm signal */
...
signal(SIGALRM, ding);   /* procedure to call when alarm goes off */

sendto(sockfd,buff,lths[i],0,NULL,0);

alarm(SECS);   /* signal me in SECS seconds */
if (recvfrom(sockfd,buff,lths[i],0,&from,&fromlen) <0){
    if (errno == EINTR) lost++;
      else perror("recvfrom");
}
alarm(0);   /* cancel timer */
```


rdate

```
/*  rdate.c   udp version */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <signal.h>
#define BASE1970  2208988800L /* difference between Unix time and net time */
#define SECS 3
int timedout;
void ding(int signo) { timedout=1; }
main (argc,argv)
int argc;
char *argv[];
{
    int i;
    if (argc == 1) {
          printf("usage: rdate <host1> <host2> <host...>\n");
          exit(1);
    }
    signal(SIGALRM,ding);
    for( i = 1; i < argc; i++ )
          RemoteData(argv[i]);
}
```



```
RemoteData(host)
char *host;
{
    struct   hostent *him;     /* host table entry */
    struct   servent *timeServ;   /* sevice file entry */
    struct   sockaddr_in sin;  /* socket address */
    struct sockaddr from;
    int fromlen = sizeof(from);
    int   fd;         /* network file descriptor */
    long  unixTime;      /* time in Unix format */
    u_char  netTime[4];     /* time in network format */
    int   i;       /* loop variable */
    char  *ctime();

    if ((him = gethostbyname(host)) == NULL) {
          fprintf(stderr, "rdate: Unknown host %s\n", host);
          return(-1);
    }
    if ((timeServ = getservbyname("time","udp")) == NULL) {
          fprintf(stderr, "rdate: time/udp: unknown service\n");
          return(-1);
    }
    if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
          perror("rdate");
          return(-1);
    }
```

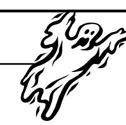


```
     sin.sin_family = him->h_addrtype;
     bcopy(him->h_addr, (caddr_t)&sin.sin_addr, him->h_length);
     sin.sin_port = timeServ->s_port;
     printf("[%s]\t", him->h_name);
     if (sendto(fd,netTime,0,0,&sin,sizeof(sin))<0){
        perror("rdate send");
        close(fd);
        return(-1);
    }
/* read in the response */
/**   for udp  need timeout here and verify size is right size **/
     timedout=0;
     alarm(SECS);
     i = recvfrom(fd,netTime,sizeof(netTime),0,&from,&fromlen);
     if (i != sizeof(netTime) || timedout) {
        if (timedout)printf(" no response\n");
        else  perror("rdate recv");
        close(fd);
        return(-1);
     }
     alarm(0);
     close(fd);
     unixTime = ntohl(* (long *) netTime) - BASE1970;
     printf("%s", ctime(&unixTime));
}
```


UDP details

- Your application may need to worry about reliability
  - Lost packets (timers/retransmission)
  - Packet ordering (sequence numbers)
  - Data conversion
- Maximum packet/datagram size?
  - OS dependent
  - Datagram can be larger than MTU, but then IP must fragment (NFS uses 8K datagrams or bigger!)
- Even though write() completes, packet may still be in OS buffer
- Packet may be lost/dropped, but sender will never know!
  - netstat –s can you tell if the OS is dropping UDP packets
  - netstat –a shows you what network ports are active
  - lsof (privileged) can tell you what processes have what ports

Things that go bump in the net



- UDP sendto and no server process
- UDP sendto with connect(), and no server process
- active UDP session, ctrl-c server
- inactive UDP session, server computer crashes and reboots and restarts server
- server tries to bind to port < 1024
- start 2nd copy of server
- A sends faster than B can receive (UDP)
- A sends faster than its interface (ENOBUFS)
- A sends 10000 byte datagram
- A sends 500-byte datagram, B recvfrom length is 100
- A sending to B, can C send to B?
- Does content of packet data (all zeros vs random) affect performance?

Tips

netstat –s may show overruns

Increase SO_RCVBUFF

Next time …

- UDP internals
- Some UDP applications