Implementing Network Software

Outline
- Sockets
- Example
- 1st Programming Assignment

Communication Paradigms

Stream Paradigm	Message Paradigm
Connection-oriented	Connectionless
1-to-1 communication	Many-to-many communication
Sequence of individual bytes	Sequence of individual messages
Arbitrary length transfer	Each message limited to 64 Kbytes
Used by most applications	Used for multimedia applications
Built on TCP protocol	Built on UDP protocol

Figure 3.1 The two paradigms that Internet applications use.

Stream Paradigm

- Stream denotes a paradigm in which a sequence of bytes flows from one application program to another
- Internet's mechanism arranges two streams between a pair of communicating applications, one in each direction
- The network accepts input from either application, and delivers the data to the other application
- The stream mechanism transfers a sequence of bytes without attaching meaning to the bytes and without inserting boundaries
- A sending application can choose to generate one byte at a time, or can generate blocks of bytes
- The network chooses the number of bytes to deliver at any time
  - the network can choose to combine smaller blocks into one large block or can divide a large block into smaller blocks

Message Paradigm

- In a message paradigm, the network accepts and delivers messages
- Each message delivered to a receiver corresponds to a message that was transmitted by a sender
  - the network never delivers part of a message, nor does it join multiple messages together
  - if a sender places exactly *n* bytes in an outgoing message, the receiver will find exactly *n* bytes in the incoming message
- Message service does not make any guarantees; messages may be
  - lost (i.e., never delivered)
  - duplicated (more than one copy arrives)
  - delivered out-of-order
- A programmer who uses the message paradigm must insure that the application operates correctly
  - even if packets are lost or reordered

Sockets

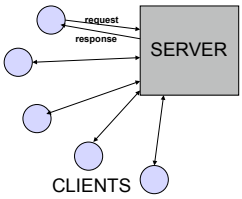
- Application Programming Interface (API)
- Socket interface
- "socket": point where an application attaches to the network
- What operations are supported on a socket?
  - specify local/remote communication endpoints
  - initiate a connection
  - wait for a connection
  - send/receive data
  - handle incoming data
  - terminate connection gracefully
  - handle connection termination from remote site
  - abort communication
  - handle error conditions or connection abort
  - allocate and release local resources

Socket Communication

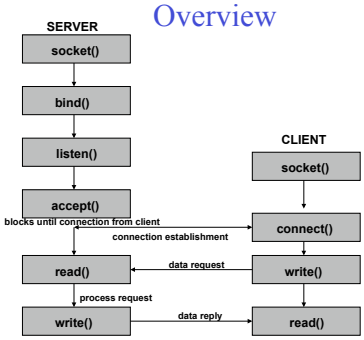
- File I/O:
  - open, close, read, write, seek, fcntl, ...
- Network communication:
  - developers extended set of file descriptors to include network connections.
  - extended read/write to work on these new file descriptors.
  - but other required functionality did not fit into the 'open-read-write-close' paradigm.
- -> Socket API

Client-Server Model

- Server listens for requests from clients
- Server: passive open
- Client: active open
- Example:
  - file server
  - web server
  - print server
  - mail server
  - name server
  - X window server



Overview



Socket Creation

int socket(int family, int type, int protocol);

- return value: socket descriptor (like file descriptor)

```
#include <sys/types.h>
#include <sys/socket.h>
int s;
s = socket (PF_INET, SOCK_STREAM, 0);
```

Socket Creation

- Domain: PF_INET, PF_UNIX, ...
- Type:
  - datagram UDP SOCK_DGRAM
  - reliable stream TCP SOCK_STREAM
  - raw IP SOCK_RAW
- Protocol: typically 0 = default protocol for type

Endpoint Addresses

- Different protocols may have different representations.
- TCP/IP uses combination of IP address and port.
- Sockets offer different 'address families', TCP/IP uses a single address representation (AF_INET).
- PF_INET, AF_INET often confused, use same numeric value (2).
- Now: #define PF_INET AF_INET (Linux)

Socket Addresses

```
struct sockaddr {               /* sys/socket.h */
    sa_family_t sa_family;      /* address family */
    char sa_data[14];           /* addressing information */
}

struct sockaddr_in {
    short sin_family;           /* AF_INET */
    u_short sin_port;           /* network byte order */
    struct sin_addr sin_addr;   /* network address */
}
```

Binding the Local Address

*int bind(int s, struct sockaddr *addr, int addresslen);*

- Socket has no notion of endpoint addresses (neither local nor remote).
- bind(): specify local endpoint.

Binding the Local Address

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int s;
struct sockaddr_in sin;

s = socket(PF_INET, SOCK_DGRAM, 0);
memset((char *)&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (6000); /* 0 -> let system choose */
sin.sin_addr.s_addr = htonl(INADDR_ANY); /* allow any interface */
bind (s, (struct sockaddr *)&sin, sizeof(sin));
```


setsockopt()/getsockopt()

- A server waits 2 MSL (maximum segment lifetime) for old connection.
- If not properly terminated, bind() will return EADDRINUSE.

*setsockopt(int s, int level, int optname, const void *optval, int optlen);*
*getsockopt(int s, int level, int optname,void *optval, int *optlen);*

- Before bind: *setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)&opt, sizeof(opt));*
- Allows re-use of port in use.

More Socket Options

- SO_ERROR — get error status
- SO_KEEPALIVE — send periodic keep-alives
- SO_LINGER — close() + non-empty buffer
- SO_SNDBUF — send buffer size
- SO_RCVBUF — receive buffer size

Linger

- Closing:

```
struct linger {
  int l_onoff;    /* 0:off, else on */
  int l_linger;   /* seconds */
}
```

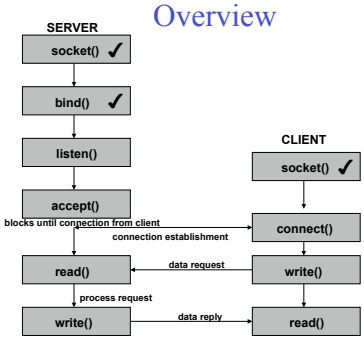
l_onoff	*l_linger*	*behavior*
0	*ignored*	*graceful shutdown*
1	*0*	*abort & flush, send RST*
1	*>0*	*sleep until data ack'ed*

shutdown

shutdown (int s, int how);

- how = 0: no more receives
- how = 1: no more sending
- how = 2: neither (close();)

Overview



connect()

*int connect(int s, struct sockaddr *addr, int namelen);*

- Client issues connect() to
  - establish remote address and port
  - establish connection
- Fails if host not listening to port.

listen()

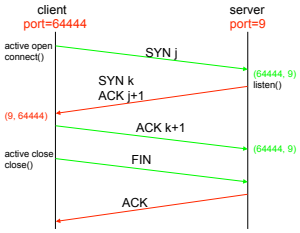
int listen(int s, int backlog);

- Only for stream sockets.
- Socket being listened to can't be used for client.
- listen() does not wait for connections, but needed to receive connection attempts.
- Completes 3-way handshake.

listen()

- Allows backlog pending connection requests (SYN, completed 3WH) while waiting for accept().
- Queue full: SYN requests are silently dropped.
- "SYN-flood": send fake TCP SYN requests to fill queue.

TCP Preview



accept()

*int accept(int s, struct sockaddr *addr, int *addrlen);*

- By connection-oriented server, after listen().
- Can't preview connection: accept and close.
- Returns a new socket.
- Returns address of client.

Sending/Receiving

- read/write: send/receive, no explicit address
- send/recv: send/receive, flags
- sendto: specify destination explicitly
- recvfrom: also receive address of peer
- sendmsg: msghdr data structure, scatter/gather
- recvmsg: msghdr data structure, scatter/gather

Example: TCP Client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int s, n;
  struct sockaddr_in sin;
  char msg[80] = "Hello, World!";

  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket"); return(-1);
  }
```


Example: TCP Client

```
sin.sin_family = AF_INET;
sin.sin_port = htons(atoi(argv[2]));
sin_addr.s_addr = inet_addr(argv[1]);
if (connect(s, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
  perror("connect"); return -1;
}
if (write(s, msg, strlen(msg)+1) < 0) {
  perror("write"); return -1;
}
if ((n = read(s, msg, sizeof(msg))) < 0) {
  perror("read"); return -1;
}
printf("%d bytes: %s\n", n, msg);
if (close(s) < 0) { perror("close"); return -1; }
return 0;
}
```

Single-Threaded Server

```
int server;

server = initialize();
while (1) {
  wait_for_request(server);
  read_request;
  send_answer;
}
```


Example: TCP Server

```
int main(int argc, char *argv[]) {
  int s, t, n;
  struct sockaddr_in sin;
  char *r;
  char buf[100];
  int sinlen;

  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket); return -1;
  }
  sin.sin_family = AF_INET;
  sin.sin_port = htons(13333);
  sin.sin_addr.s_addr = INADDR_ANY;
  if (bind (s, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
    perror("bind"); return -1;
  }
  if (listen(s, 5) < 0) { perror("listen"); return -1;}
```


Example: TCP Server

```
 for (;;) {
   sinlen = sizeof(sin);
   if ((t = accept(s, (struct sockaddr *) &sin, &sinlen)) < 0) {
     perror("accept"); return -1;
   }
   if (read(t, &buf, 100) < 0) { perror ("read"); return -1;}
   r = gettime();
   if (write(t, r, strlen(r)) < 0) { perror ("write"); return -1; }
   if (close(t) < 0) { perror("close"); return -1; }
 }
 if (close(s) < 0) { perror("close"); return -1; }
}
```

Message Paradigm

- The socket functions used to send and receive messages are more complicated than those used with the stream paradigm because many options are available
  - for example, a sender can choose whether to store the recipient's address in the socket and merely send data or to specify the recipient's address each time a message is transmitted
  - furthermore, one function allows a sender to place the address and message in a structure and pass the address of the structure as an argument, and another function allows a sender to pass the address and message as separate arguments
- You are "encouraged" to discover details yourself:
  - sendto and sendmsg socket functions
  - recvfrom and recvmsg socket functions

Other Socket Functions

- The socket API contains a variety of support functions
  - *getpeername*
  - *gethostname*
  - *setsockopt*
  - *getsockopt g*
  - *gethostbyname*
  - *gethostbyaddr*

What About Threaded Servers?

- The socket API works well with concurrent servers
- Implementations of the socket API adhere to the following inheritance principle:
  - each new thread that is created inherits a copy of all open sockets from the thread that created it
  - the socket implementation uses a reference count mechanism to control each socket
  - when a socket is first created
    - the system sets the socket's reference count to 1
    - and the socket exists as long as the reference count remains positive.
  - when a program creates an additional thread
    - the thread inherits a pointer to each open socket the program owns
    - and the system increments the reference count of each socket by 1
  - when a thread calls close
    - the system decrements the reference count for the socket
    - if the reference count has reached zero, the socket is removed

First Assignment

- Must be written in C or C++
- Labs: 1st floor Fitzpatrick, DARTS Lab
- IP Addresses: ifconfig, hostname
- Port numbers: 9000-10000
- Compiling: **gcc –g server.c –o server**
- To run server in background: &
- Test code between machines in the same lab