Lecture 3: UNIX Socket API

Network application software uses the *socket* interface.
Funded by ARPA, developed at Berkeley

Introduction
Goal: transport TCP/IP software to Unix and develop an application-level interface.

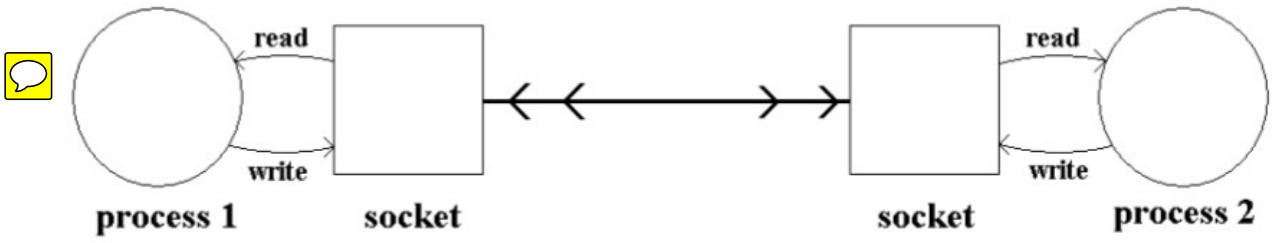
Result: Socket API in Berkeley Unix (BSD 4.2, 1983)

- used a combination of existing Unix system calls and some new ones
- integrated with Unix file system
- implemented within the Unix kernel
- eventually adapted as a standard by Unix vendors (SUN, HP, linux, etc.) and non-Unix systems (Windows, Java)
- also developed early applications (telnet, ftp)
- client/server model

The interface is general enough to support a variety of protocols ("domains"), but on most systems, the only ones supported are:

- stream sockets (TCP)
- datagram sockets (UDP)
- Unix domain sockets (for interprocess communication on a single Unix system)
- raw sockets (direct access to IP packets)

Once a socket connection is established, it supports symmetric, two-way communication.
Conceptually, the socket is the endpoint of a communication link, similar to a telephone receiver:



Each process has a socket.
process 1 writes to its socket, process 2 reads from its socket
process 2 writes to its socket, process 1 reads from its socket

Reading is sequential (stream)

At each end, the socket appears to be two byte streams, one for reading and one for writing. It is represented by a Unix file descriptor, so normal file I/O system calls (read, write) can be used.

The application designer can then design the application protocol, considering issues such as

- format of messages
- meaning of messages

- message sequences used to accomplish some goal (i.e., protocols)

On the other hand, establishing the connection is *asymmetric*. Client and server have specific roles, and there are certain system calls specifically used for client or server (but not both).

Establishing a connection:

1. A server initializes itself to listen for incoming requests for connection by clients, then lies dormant until a client request arrives.
2. The client initiates communication by sending a message to the server requesting connection, then waits for a reply.
3. The server accepts the connection, gets a file descriptor to represent the server-side socket.
4. The client connection request returns, passing to the client a file descriptor to represent the client-side socket.

Socket-related system calls

- socket
- bind
- listen
- accept
- connect

1. socket

- create a socket. (The created socket is not connected. This call just makes an entry in a table in the kernel.)

2. bind

- used by a server to bind a socket to an address
- an address consists of (host IP address, port number)
- common services use well-known port numbers, such as
    - http: 80
    - ftp: 21
    - telnet: 23
    - ssh 22

3. listen

- used by servers to make a socket into a listening socket that will listen for connection requests from clients.

4. accept

- used by servers to wait for connection requests on a listening socket.
- used only for sockets using a connection-oriented protocol

5. connect

- used by clients to request a connection to a remote server

Note: A client must create a socket before calling connect, but does not need to bind it to a local address. The client socket is assigned to an "ephemeral" port number by the TCP/IP software.

So, setting up a connection works like this:

Server:	Client:
create socket	
bind to local address	
convert to listening socket	
loop	
accept connection	
get request	
send response	
close connection	
end loop	
	create socket
	request connection
	get reply
	send request
	get response
	close connection

Once the connection is established, the request-response dialog may continue repeatedly.

More detail on system calls:

```
int socket(int protofamily, // PF_INET or PF_UNIX
           int type,        //  SOCK_STREAM or SOCK_DGRAM
           int protocol     //  IPPROTO_TCP or IPPROTO_UDP or 0 (use default)
);
```

- Creates a socket data structure within the Unix kernel.
- Returns a file descriptor if successful, -1 if error. The file descriptor is used to identify the socket in subsequent calls.

```
int connect(int socket, // file descriptor of a (local) socket
                struct sockaddr * saddr,      // pointer to a structure containing the address of
a remote socket
                int saddrlen // length (in bytes) of the sockaddr struct
);
```

Establishes a connection between a local socket and a remote socket.

- Returns 0 if OK, -1 if error.
- Possible errors:
  - time out
  - connection refused
  - already connected
  - bad file descriptor
  - not a socket descriptor

The sockaddr struct is very important. There is a generic version of the struct, defined as follows:

```
struct sockaddr {
   u_char   sa_len;          // length of the struct
   u_char   sa_family;       // address family
   char     sa_data[14];    // actual address; format depends on address family
};
```

For internet addresses, use the address family AF_INET. A specialized struct is defined for internet addresses:

```
struct sockaddr_in {
   u_char   sin_len;            //  length of the struct
   u_char   sin_family;         //  address family (AF_INET)
   u_short   sin_port;          //  port number
   struct in_addr sin_addr;    //  32-bit binary IP address
   char       sin_zero[8];      //  set to zero
};
```

Functions are supplied to generate IP addresses.

```
int bind(int sockfd,             //  file descriptor for a local socket
         struct sockaddr * saddr,  //  socket address
         int addrlen           //  length of sockaddr struct
);
```

Binds a socket to an address.

- Returns 0 if OK, -1 if error
- saddr may be INADDR_ANY to bind to any local address

```
int listen(int sockfd,     //  file descriptor for a local socket
           int backlog   //  length of queue for waiting clients
);
```

- Makes a socket into a listening socket, establishes a queue for incoming connection requests.
- Returns 0 if OK, -1 if error.

```
int accept(int sockfd, //  file descriptor for a listening socket
           struct sockaddr * saddr,  //address of local variable to be filled in with client's
socket address
           int * addrlen //address of local variable to be filled in with length of client's socket
address
);
```

Accept a connection request from a client.

- Returns a file descriptor for data socket if OK, else -1.

Also of interest are the low-level Unix system calls for reading and writing files (if you haven't seen these before):

```
int read(int fd, //  file descriptor to read
         char * buffer, //  address of a local variable to be filled with incoming data
         int len //number of bytes to read
);
```

- Read len bytes from fd to buffer.
- Return value is the number of bytes actually read, or -1 if error.

```
int write(int fd, //  file descriptor to write
        char * buffer, //  address of a local variable containing data to be written
```

```
        int len //  number of bytes to write
);
```

- Write len bytes from buffer to fd.
- Return value is the number of bytes actually written, or -1 if error.

TCP compares with a telephone system

socket() is like having a telephone to use.

bind() is like telling others your phone number so they can call you.

listen() is like turning on the ringer so you will hear when an income call arrives.

connect() is like we call someone whose phone number is known to us.

Having client's identity returned by accept() is like having a caller id feature.
Several additional system calls are available which are designed specifically for writing to and reading from sockets.

```
int send(int fd,
        char * buffer,
        int len,
        int flags);
```

Equivalent to a write operation with one additional parameter. The flags parameter may be used to indicate "out-of-band" data.

```
int recv(int fd,
        char * buffer,
        int len,
        int flags);
```

Equivalent to a write operation with one additional parameter. The flags parameter may be used to indicate "out-of-band" data or a "peek" operation which reads data from the socket but does not consume it. The data is available to read by subsequent recv calls.

The other two functions sendto and recvfrom can be used for datagram sockets. Their prototypes are

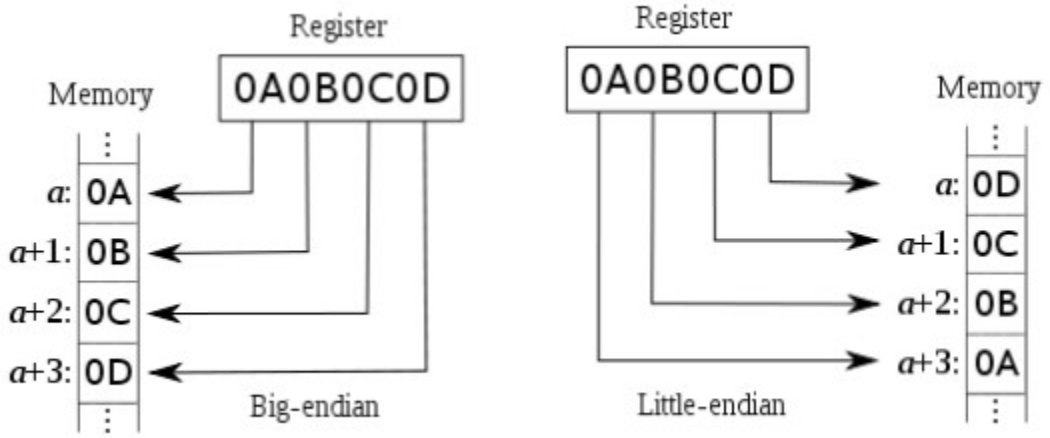
```
ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to,socklen_t
      tolen);
```

```
 ssize_t recvfrom(int s, void *buf, size_t len, int flags,struct sockaddr *from, socklen_t
      *fromlen);
```

Notes

1. network byte order and host byte order

To store an int, some computers store the most significant digit first in the member cells allocated for the int. This is called big-endian. The other way is called little-endian. The network sockets uses big-endian and give it another name called network byte order.



from wikiPedia

The c library provides some basic functions to convert between network and host orders

htons, htonl, ntohs, ntohl

Port numbers and addresses must be in network order!

2. How to manipulate IP addresses

```
in_addr_t inet_addr(const char *cp);
```

This function converts an IP address to a 4byte int to be stored in the sin_addr of the sockaddr_in struct. **(IPdemonstration.c)**