

CMPT 300

Operating System I

Socket

Dr. Hazra Imran

Summer 2022

The Unix philosophy

- “Everything is a file”
- This is widely regarded to be the central Unix philosophy
- We’ve seen examples of this in previous classes
 - The /dev and /proc logical filesystems
 - Terminal windows are files, your sound card is a file
 - Even network connections can be treated as files

File Descriptors

- Unix keeps track of open files through integers called *file descriptors* (fds).
- The first three integers are standard
 - 0 — standard input
 - 1 — standard output
 - 2 — standard error
- After fd 2, there are various ways of getting a new file descriptors.

File Descriptors

- Unix perfected a lot of the basic principles we take for granted in operating system design (the illusions we provide to processes, the filesystem model)
- The read, write and close system calls work on all file descriptors.

Getting file descriptors

- The open system call is the most straightforward way to get a new file descriptor: you open a file (based on the filename).
- Note that, in Unix, not every file is stored on disk (it may be a device), but open is still how you open files
- Read and write and change the contents of the file;
- close closes the file
- The pipe system call is another way to get a file descriptor
- Sockets have other system calls. . . .

Networking

- In Unix, communicating over a network is also done through file descriptors.
- The read and write system calls will still work with networking.

Sockets

- In Unix (and most operating systems), a *socket* is a file descriptor which corresponds to a network connection.
- Usually sockets are bi-directional: we can both read (using either the read or recv system calls) and write (using either the write or send system calls).
 - send/recv system calls are used for datagram transport protocols (UDP) where we might want to control the framing.
 - write/read is used for any sockets (more common in stream protocols like TCP) where framing doesn't matter.

Client-Server model

- Clients normally communicate with one server at a time.
- From a server's perspective, at any point in time, it is not unusual for a server to be communicating with multiple clients.
- Client needs to know the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.
- Client and servers communicate by means of multiple layers of network protocols.

Client-Server model

- There are two major transport layer protocols to communicate between hosts: TCP and UDP.

Addresses, Ports and Sockets

- Like apartments and mailboxes
 - You are the application
 - Your apartment building address is the address
 - Your mailbox is the port
 - The post-office is the network
 - The socket is the key that gives you access to the right mailbox

User Datagram Protocol (UDP)

- In UDP, the application writes a message to a UDP socket, which is then encapsulated in a UDP datagram, which is further encapsulated in an IP datagram, which is sent to the destination.
- UDP provides a connection-less service.
 - The client does not form a connection with the server like in TCP and instead just sends a datagram.
 - The server need not accept a connection and just waits for datagrams to arrive.
 - Datagrams upon arrival contain the address of the sender which the server uses to send data to the correct client.

User Datagram Protocol (UDP)

- The problem of UDP is its lack of reliability: if a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not automatically retransmitted.
- Each UDP datagram is characterized by a length. The length of a datagram is passed to the receiving application along with the data.

Transmission Control Protocol (TCP)

- TCP provides a *connection-oriented service* since it is based on connections between clients and servers.
- TCP provides reliability.
 - When a TCP client sends data to the server, it requires an acknowledgement in return. If an acknowledgement is not received, TCP automatically retransmits the data and waits for a longer period of time.
- UDP datagrams are characterized by a length. TCP is instead a byte-stream protocol, without any boundaries at all.

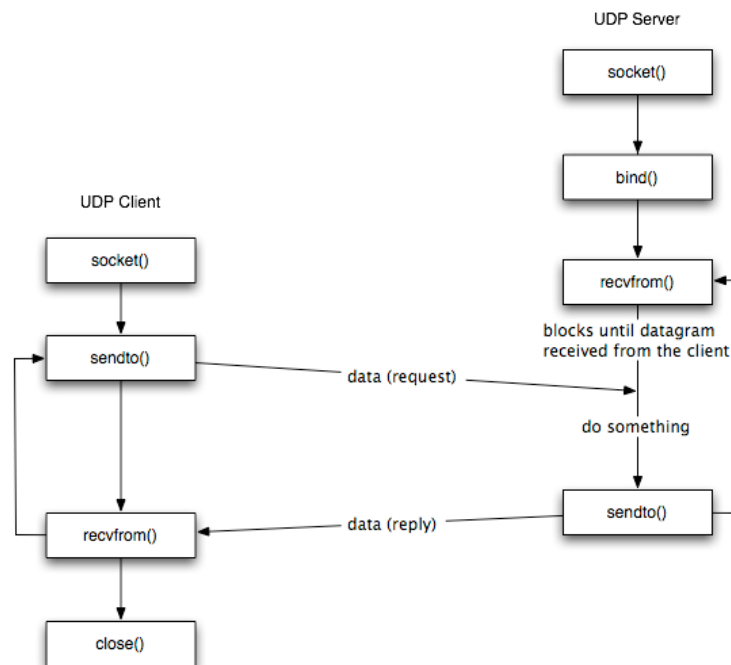
UDP vs TCP

- UDP and TCP sockets work similarly.
- Using UDP requires you to worry about reliability (e.g., dropped packet detection, retransmission, reordering) yourself.
- Also requires you to do your own framing (where does one packet end and the next begin?).

UDP Socket

Steps on the client-side:

- Create a socket using the `socket()` function.
- Send and receive data by means of the `recvfrom()` and `sendto()` functions.

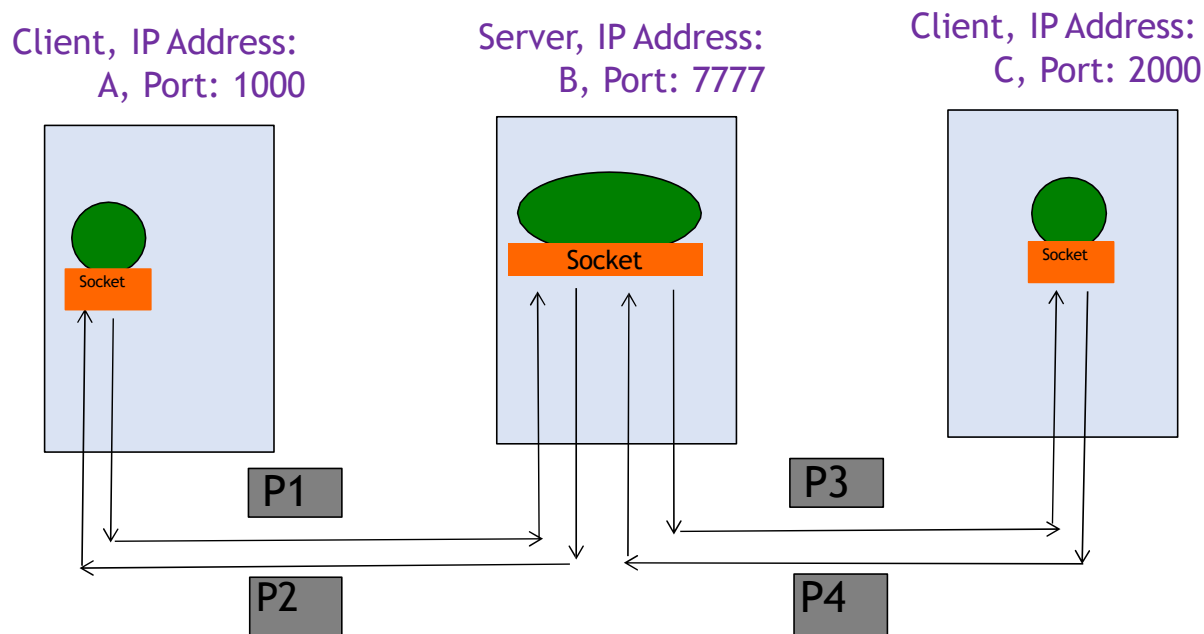


Steps on the server-side

- Create a socket with the `socket()` function.
- Bind the socket to an address using the `bind()` function;
- Send and receive data by means of `recvfrom()` and `sendto()`.

UDP Socket

- A single socket at server end
- It is the responsibility of server to differentiate messages from different clients



UDP

UDP Client

1. Create a UDP socket.
2. Send a message to the server.
3. Wait until a response from the server is received.
4. Close socket descriptor and exit.


UDP Server

1. Create a UDP socket.
2. Bind the socket to the server address.
3. Wait until the datagram packet arrives from the client.
4. Process the datagram packet and send a reply to the client.

Creating Sockets

- Sockets are created with the `socket` system call
- We tell it what type of socket to create, and it gives us back a file descriptor

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



family specifies the protocol family (AF_INET for the IPv4 protocols),

Specifies the type of socket to be created.

SOCK_STREAM for stream sockets and SOCK_DGRAM for datagram sockets.

- Specifies a particular protocol to be used with the socket.
- Specifying a protocol of 0 causes `socket()` to use an unspecified default protocol appropriate for the requested socket type.

The function returns a non-negative integer number, similar to a file descriptor, that we define *socket descriptor* or -1 on error.

Example

```
int main() {  
    int sockfd;  
    // Creating socket file descriptor  
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {  
        perror("socket creation failed");  
        exit(EXIT_FAILURE);  
    }
```

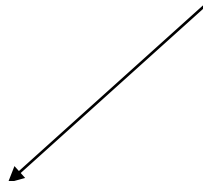
`int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)` Assigns address to the unbound socket.

Binding a listening socket

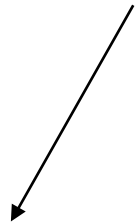
- Every listening socket has to be bound before it can start listening.
- `Bind()` assigns an address to the unbound socket.

```
#include <sys/socket.h>
```

```
int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```



sockfd - File descriptor of a socket to be bonded



servaddr is a pointer to a protocol-specific address



addrlen - Size of *address* structure

`bind()` returns 0 if it succeeds, -1 on error.

Binding

- This use of the generic socket address `sockaddr` requires that any calls to these functions must cast the pointer to the protocol-specific address structure.

```
struct sockaddr_in serv;  
bind(sockfd, (struct sockaddr*) &serv, sizeof(serv))
```

sendto()

- Send a message on the socket

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buff, size_t nbytes,
               int flags, const struct sockaddr *to,
               socklen_t addrlen);
```

- sockfd is the socket descriptor,
- buff is the pointer to write from
- nbytes is number of bytes to
- The additional argument flags is used to specify how we want the data to be transmitted.
- to argument is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is sent.
- addrlen specified the size of this socket.

The function returns the number of bytes written if it succeeds, -1 on error.

recvfrom()

- Receive a message from the socket.

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void* buff, size_t nbytes,
                  int flags, struct sockaddr* from,
                  socklen_t *addrlen);
```

- sockfd is the socket descriptor
- buff is the pointer to read into
- and nbytes is number of bytes to read.
- The recvfrom function fills in the socket address structure pointed to by from with the protocol address of who sent the datagram.
- The number of bytes stored in the socket address structure is returned in the integer pointed by addrlen.

The function returns the number of bytes read if it succeeds, -1 on error.

Close()

- Close a file descriptor

```
#include <unistd.h>
int close(int sockfd);
```

The normal close() function is used to close a socket and terminate a socket.

It returns 0 if it succeeds, -1 on error.

Example

```
// Filling server information
```

```
struct sockaddr_in server_addr;  
server_addr.sin_family = AF_INET;  
server_addr.sin_port = htons(2000);  
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

htons and htonl

- When building sockets earlier, we saw usages of the htons and htonl functions.
- The first stands for “convert short integer from **host** order to **network** order”
- When dealing with port numbers and IP addresses, we *have* to use htons/htonl to use them properly.

htobe?? and betoh??

- A more flexible series of functions has been introduced to Unix operating systems over the past years.
- hto64, for instance, stands for “convert **64**-bit integer from **host** order **to big-endian**”
- betoh64 does the reverse

Resources

Beej's Guide to Network Programming —
<https://beej.us/guide/bgnet/>



Lets put together

