Sockets

Lecture 08.01

https://src-code.simons-rock.edu/git/mbarsky/socket_demo.git

Inter-process communication

- Wait for exit status (report when done)
  - Only short integer status
- Pipe (always open for communication)
  - Only between related processes
- Signals (send when you want, handle or ignore)
  - Just a poke

Inter-process communication

- Wait for exit status (report when done)
- Pipe (always open for communication)
- Signals (send when you want, handle or ignore)
- Sockets (open connection with the world)

Sockets

- We want two unrelated processes to talk with each other:
  - Created by different shells
  - Created by different users
  - Running on different machines

- Sockets are communication points on the same or different computers to exchange data
- Sockets are supported by Unix, Windows, Mac, and many other operating systems
- Now they are also supported by all modern browsers

Sockets use file descriptors to talk

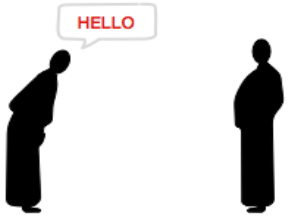
- Every I/O action is done by writing or reading to/from a stream using *file descriptor*
- To a programmer, a socket looks and behaves much like a low-level file descriptor: has read(), write(), close()
- Sockets are **full-duplex (2 way)** – as if opening a stream for both reading and writing

- The only difference – how we set up the socket

If 2 processes are unrelated – we need a *protocol* for communication

SYNTAX

- What to say



SEMANTICS

- In what context

TIMING

- When

Communication protocols

- TCP protocol – how to transfer and receive byte streams
- IP protocol – how to locate and connect to a machine on the internet
- HTTP protocol establishes rules of communication between browser and web server
- Application-level protocols: FTP, SMTP, and POP3

Socket protocols

- The two most common socket protocols:
  - TCP (Transmission Control Protocol)
  - UDP (User Datagram Protocol)

Stream sockets

Datagram sockets

Stream sockets (TCP)

- Message delivery is guaranteed. If delivery is impossible, the sender receives an error indicator
- If you send three items "A, B, C", they will arrive in the same order – "A, B, C"
- Data records do not have any boundaries

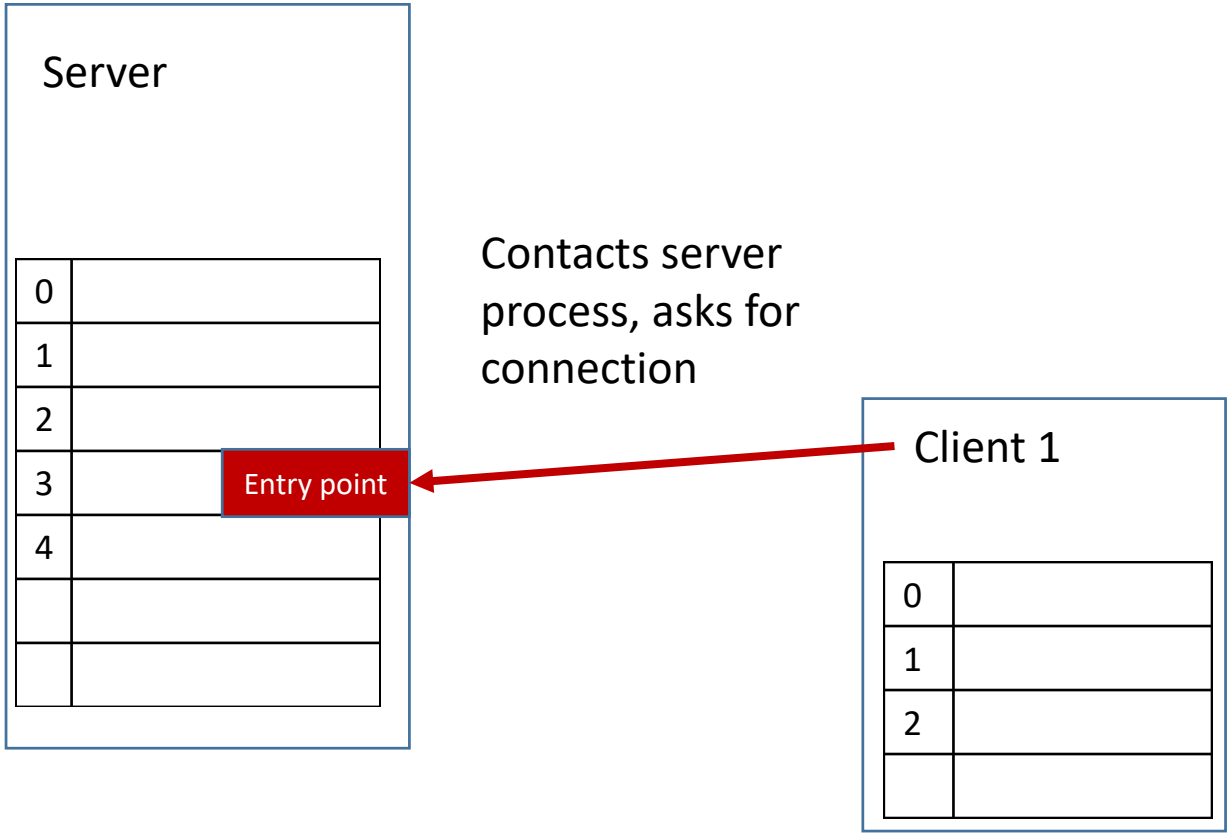
Datagram sockets (UDP)

- Order is not guaranteed
- Connectionless: you don't need to have an open connection as in Stream Sockets – you build a packet with the destination information and send it out
- Delivery is not guaranteed

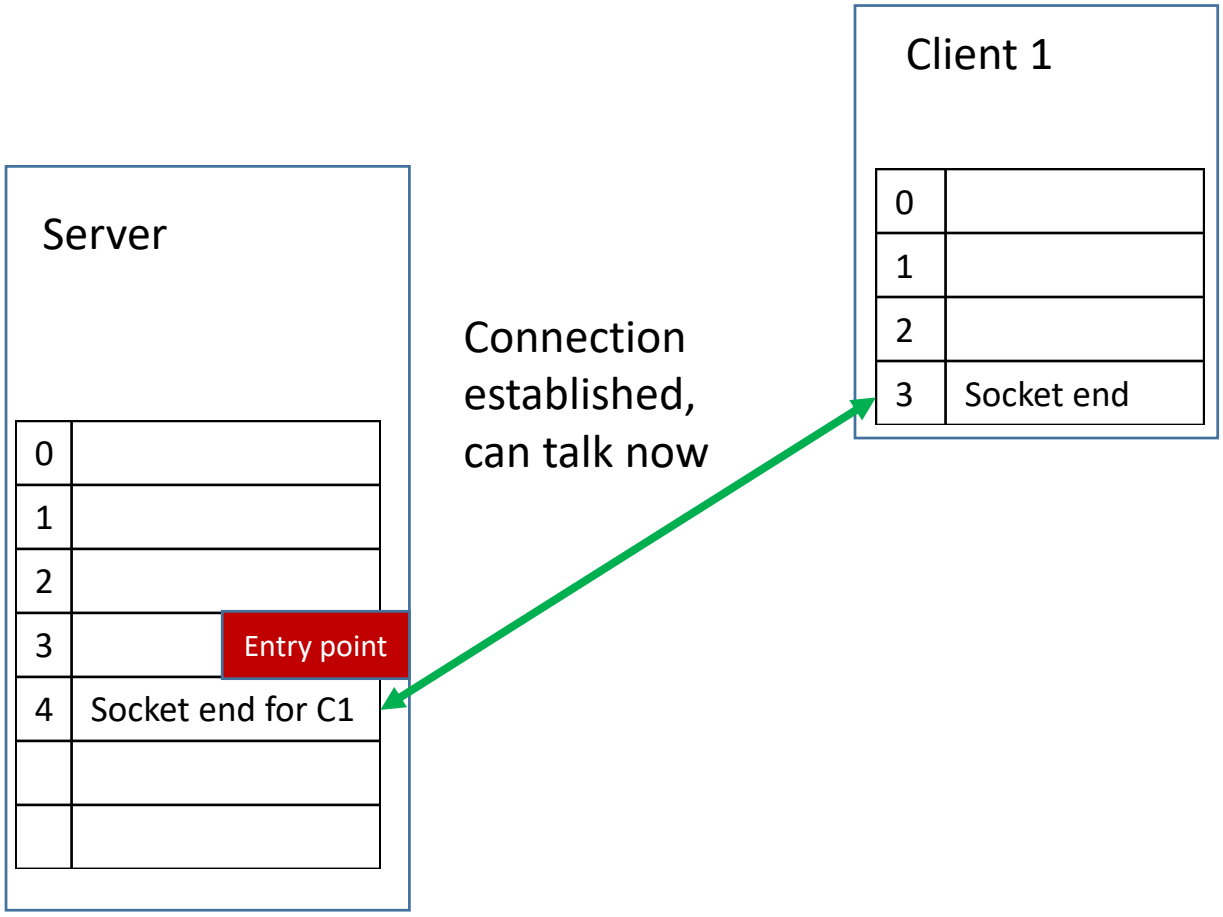
Server

- A *server* is a program that performs some functions on request from a client
- Server serves as a major switch in the phone company
- It is responsible for taking incoming calls from clients and then creating personal connection between a pair of file descriptors: one in the client and one in the server process

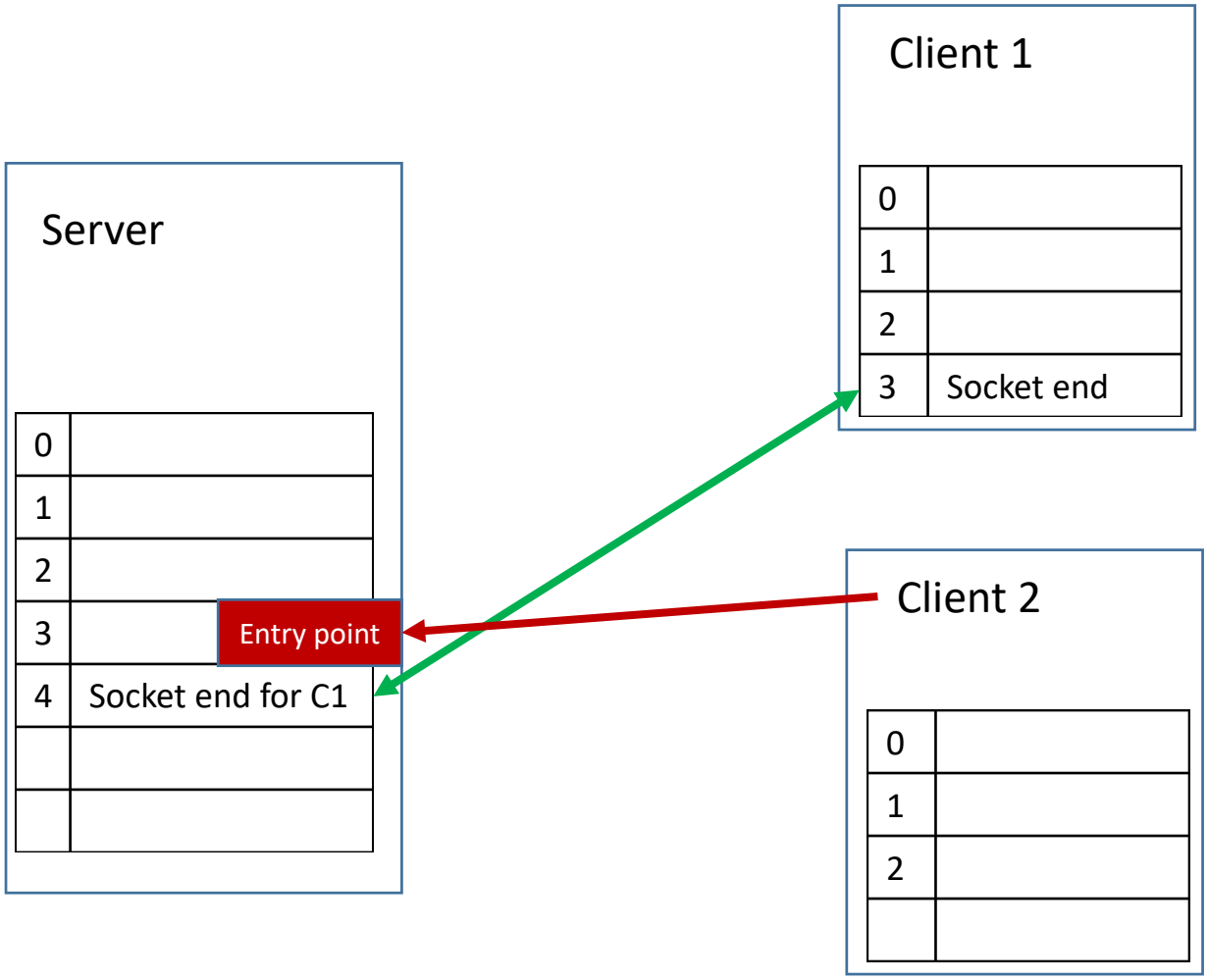
How does it work



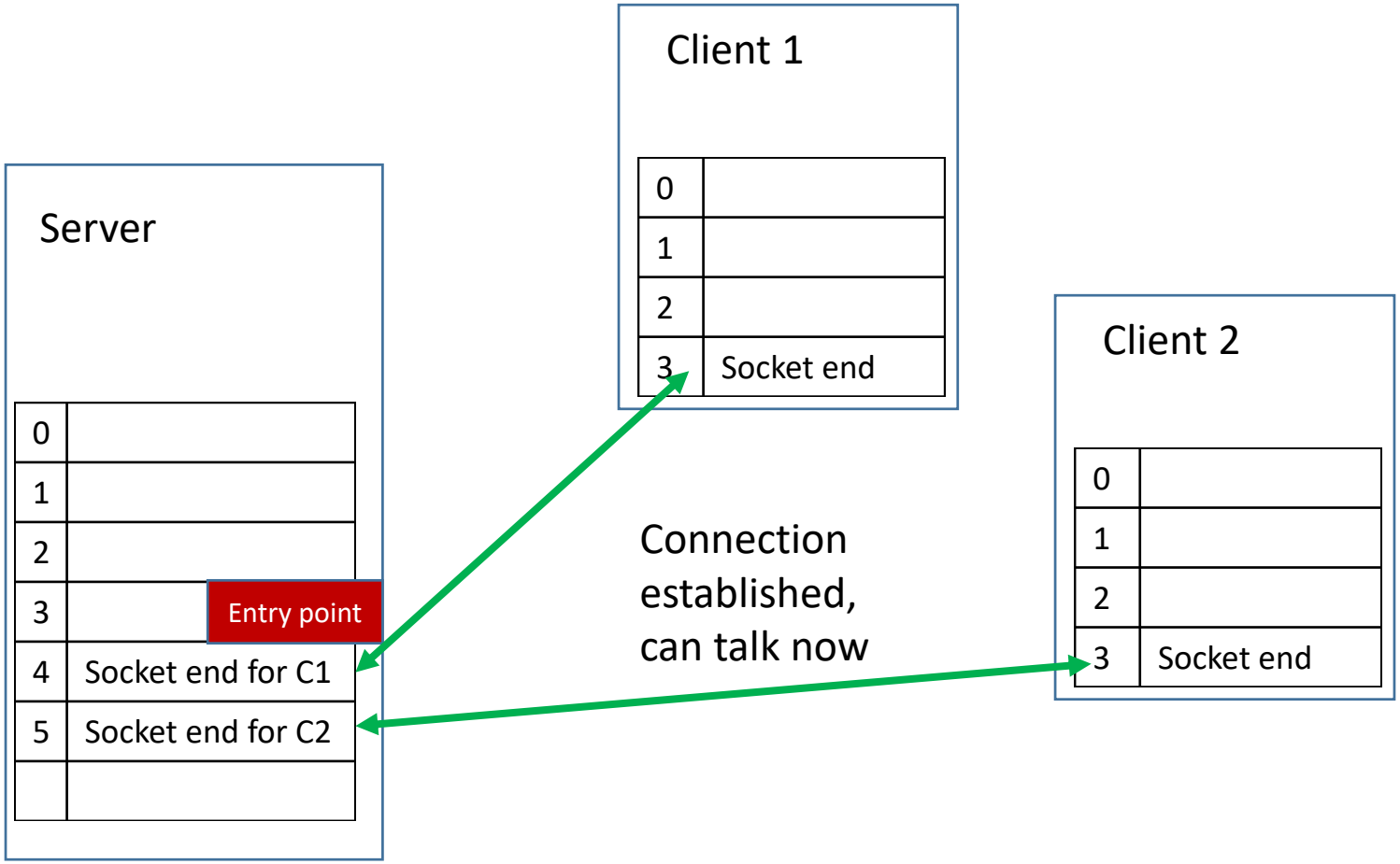
How does it work



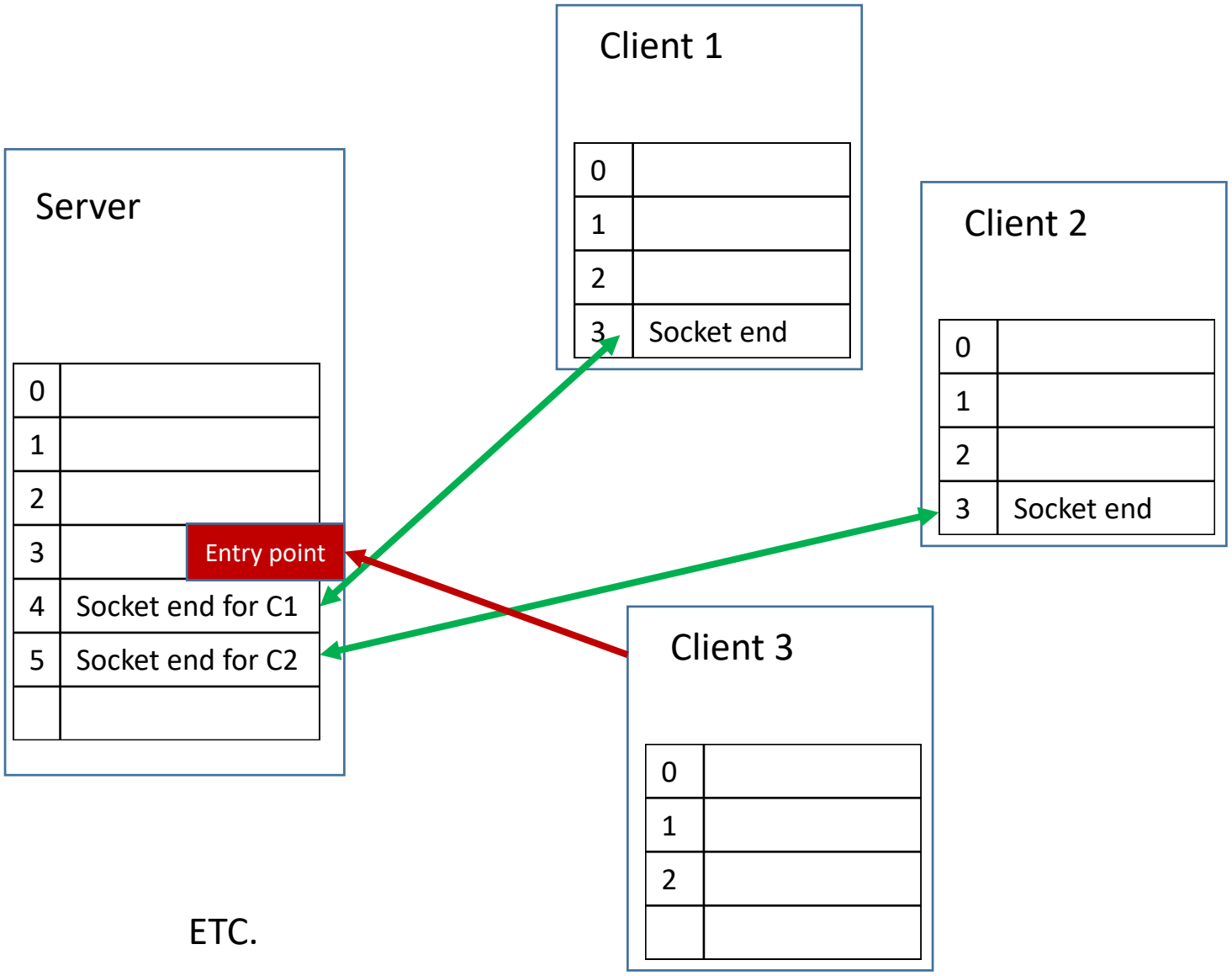
How does it work



How does it work



How does it work

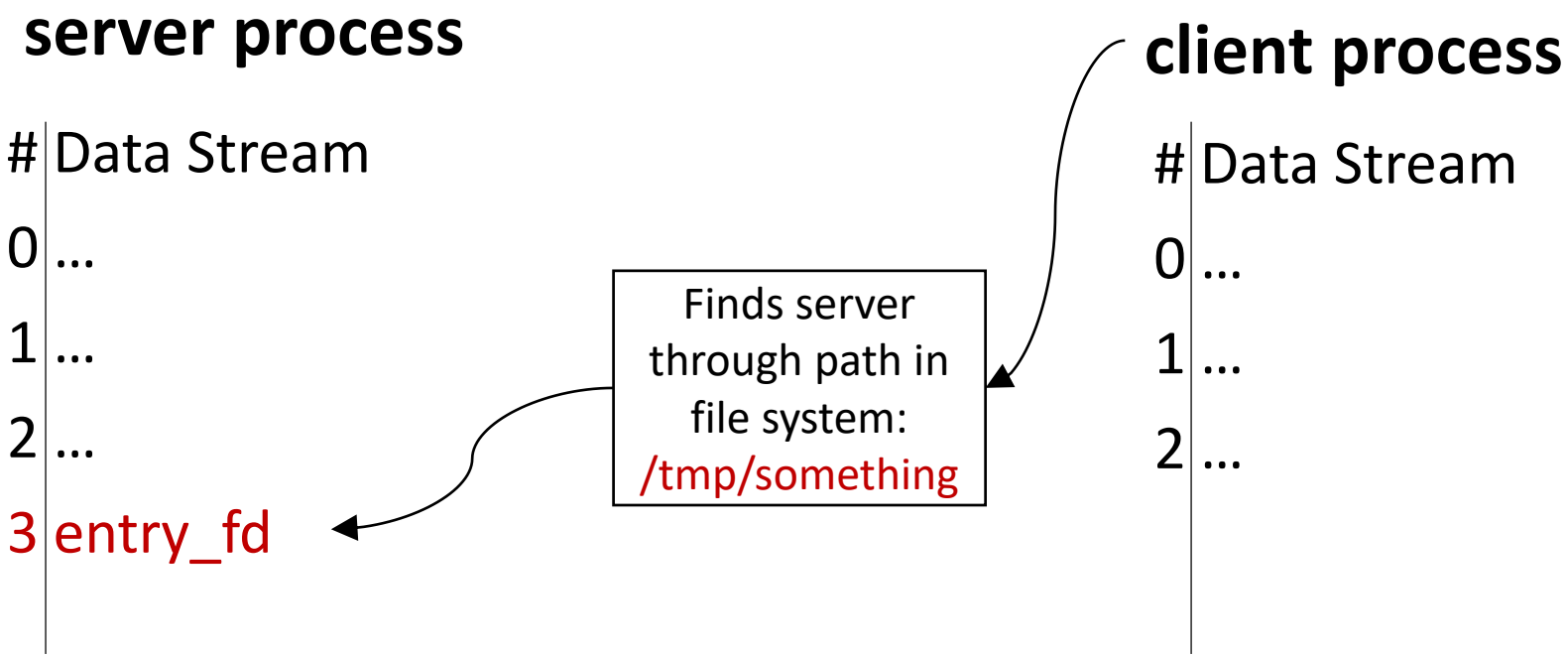


Unix domain socket server

Data communications endpoints for exchanging data between processes executing **on the same Unix host** system

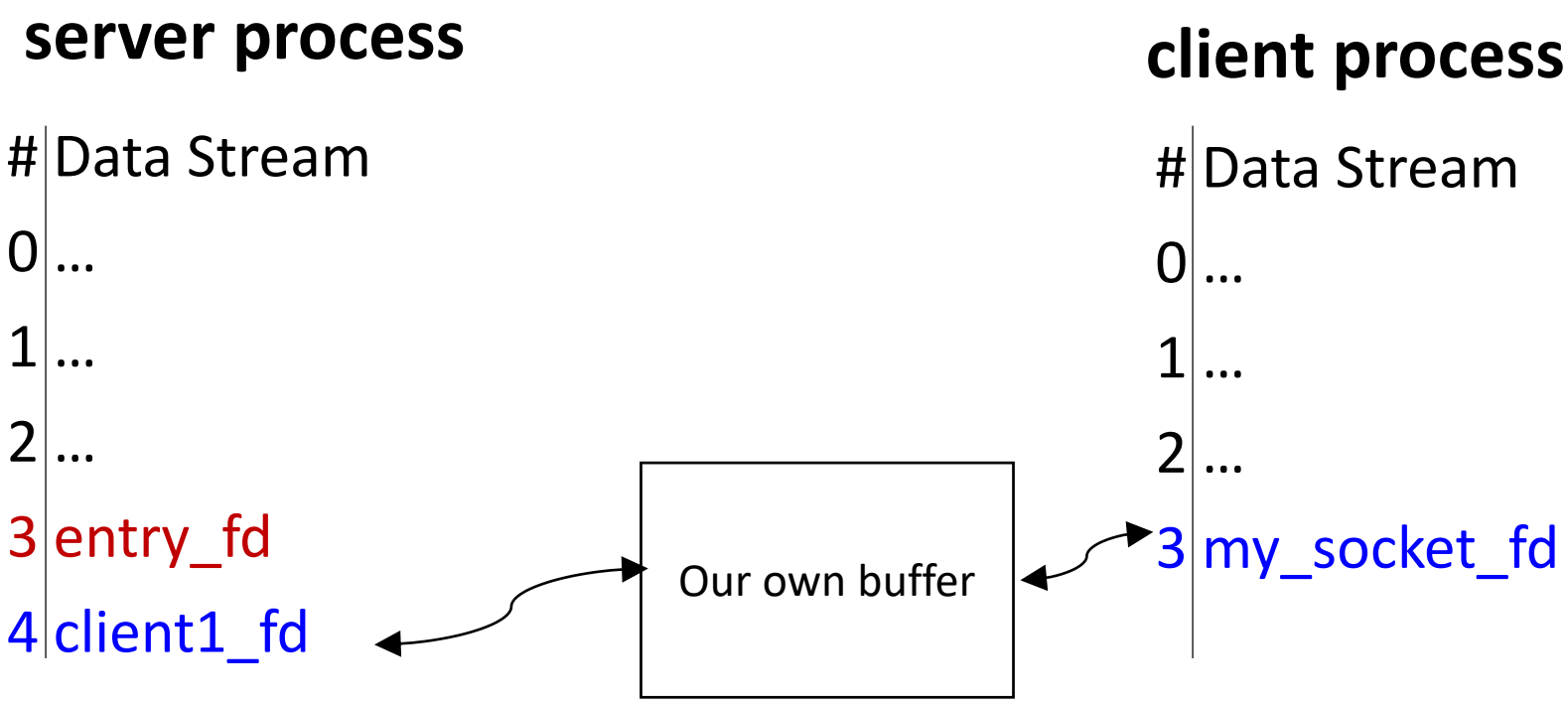
code in *server.c*

Unix domain sockets



Full-duplex communication between two unrelated processes through Unix inode

Unix domain sockets



Full-duplex communication between two unrelated processes through Unix inode

Define socket (like installing a phone socket)

```
int serv_fd;


if ((serv_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror ("socket");
        exit (1);
}
```

1 *Socket type – Unix socket*

2 *Protocol type – TCP*

3 *Protocol id – can be multiple protocols, but here only one*

socket() call does not specify where data will be coming from, nor where it will be going to –it just creates the socket prototype of a certain type and assigns it to the file descriptor - *serv_fd*

Assign address to socket (like assigning phone number)

- Socket descriptor prototype from the call to *socket* needs to be assigned an address
- For Unix sockets this address is a *file of a special type* in a file system
- Different processes can access these “files” as file system *inodes*, so two processes can establish communication

Define server socket address

```
struct sockaddr_un server_addr;
```

Declare variable of type *sockaddr_un*

```
memset(&server_addr, '\0', sizeof (server_addr));
```

Clear all bytes to zero

```
server_addr.sun_family = AF_UNIX;
```

Setup address family

```
strcpy(server_addr.sun_path, "/tmp/something");
```

Set file name through which server can be contacted

```
unlink(server_addr.sun_path);
```

Delete file with this name if already exists

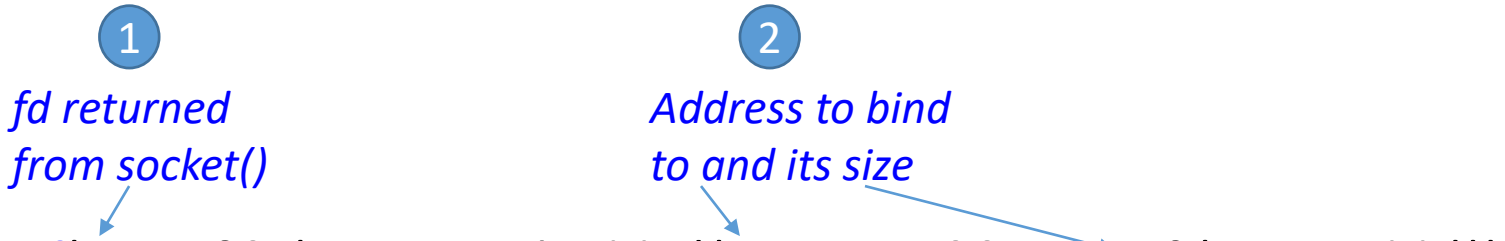
3 steps of socket server setup: BLA

1. Bind
2. Listen
3. Accept

1. Bind

- Bind socket to a connection resource - in this case the socket *inode* (a new kind of "special file") – to create an entry point

struct sockaddr_un **serv_addr**; //all set up – see above




```
if (bind(serv_fd, (struct sockaddr *)&serv_addr, sizeof (serv_addr)))
{
        perror("bind");
        return(1);
}
```

Returns zero on success

Some sort of “polymorphism”

- Because *bind* is designed to work with all kinds of sockets, and the size of the addresses may be different, the second argument of *bind*() is of a general type struct sockaddr*
- We need to cast our Unix socket address of type sockaddr_un to this general type
- Third parameter tells how much space to consider for reading an actual address from a given memory location (different types of address structs have different length)

struct sockaddr_un serv_addr; //all set up above

bind(**serv_fd**, (struct sockaddr *)&**serv_addr**, sizeof (serv_addr)) ;

2. Listen

- Listen — wait for incoming connections
- Also specifies the length of the queue for connections which have not yet been "accepted“
- It is not a limit on the number of people you can talk to - it's just how many can do a connect() before you accept() them

1 *fd returned from socket()*

Backlog for incoming connections 2

```
if (listen(serv_fd, 5)) {
    perror("listen");
    return(1);
}
```

3. Accept

- Accept processes client requests (usually in a loop)
- It returns a new socket file descriptor for talking to that particular client

```
struct sockaddr_un client _addr;
int len = sizeof (client_addr);
```

1 *fd returned from socket()*

2 *Address of a client and the length of this address*

```
if ((client_fd = accept(serv_fd, (struct sockaddr *)&client_addr,
                                                    &len)) < 0) {
    perror("accept");
    return(1);
}
```

Client address is recorded into variable *client_addr*

- When accept() returns, the *client_addr* variable will be filled with the remote side's struct *sockaddr_un*, and *len* will be set to its length
- The new file descriptor *client_fd* is created, and is ready for sending and receiving data for this particular client

```
struct sockaddr_un client _addr;
int len = sizeof (client_addr);

if ((client_fd = accept(serv_fd, (struct sockaddr *)&client_addr, &len)) < 0) {
        perror("accept");
        return(1);
}
```

Read data from a client: example

```
char buf[BUF_SIZE+1];
if ((len = read(client_fd, buf, BUF_SIZE))) < 0) {
      perror("read");
      return(1);
}

// The read is raw bytes.  This turns it into a C string.
buf[BUF_SIZE] = '\0';
printf("The other side said: %s\n", buf);
```

Write data to a client: example

```
//echo data back
if (write(client_fd, buf, strlen(buf)) != strlen(buf) ) {
      perror("write");
      return(1);
}
```

Close

- Closing the client_fd makes the other side see that the connection is closed

close(client_fd);

- Unix domain socket binding is reclaimed upon process exit, but the inode is not. You have to explicitly unlink (delete) it

close(server_fd);

unlink("/tmp/something");

- If you run the server, you can connect to it in a different terminal using a general client program netcat (nc):

```
nc -U /tmp/something
```

- You can see all open unix sockets using netstat:

```
netstat -lxp
```

- You can see that socket in Unix sockets is a ‘file’:

```
ls -l  /tmp/something
```

Unix domain socket client

code in *client.c*

Client program: socket setup

- Create a socket interface of type Unix domain socket:

Client has only one file descriptor used to connect to a remote process and if successfully connected – this will be the fd of the communication

```
if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return(1);
}
```

Connect to known address

- The **client** does **connect**(), the **server** does **accept**()
- Fill-in fields of server address – to which to connect:

```
struct sockaddr_un serv_addr;
memset(&serv_addr, '\0', sizeof (serv_addr));
serv_addr.sun_family = AF_UNIX;
strcpy (serv_addr.sun_path, "/tmp/something");
```

```
if (connect(fd, (struct sockaddr *)&serv_addr, sizeof (serv_addr)))
{
    perror("connect");
    return(1);
}
```

Descriptor created with *socket()*

Now client can write and read

```
if ((len = write(fd, "Hello", 5)) != 5) {
      perror("write");
      return(1);
}

if ((len = read(fd, buf, MAX_LINE)) < 0) {
      perror("write");
      return(1);
}
buf[MAX_LINE] = '\0';
```