Tutorial 10

NWEN241

Socket Programming

Kirita-Rose Escott

kirita-rose.escott@ecs.vuw.ac.nz

Content

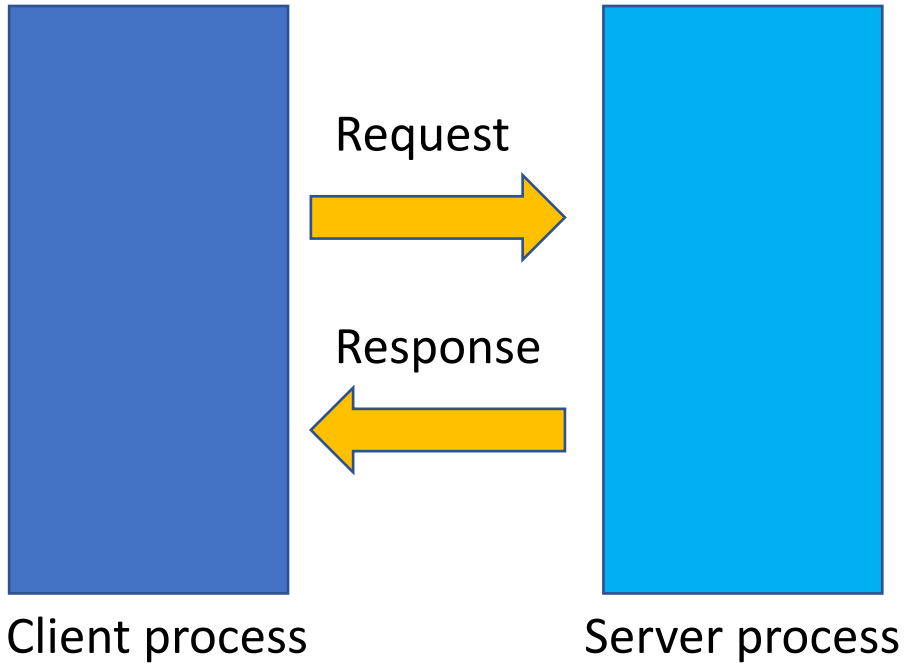
- Socket programming problem

Socket Programming Problem

- We have a user who wants to be able to enter an input which will be reversed and sent back to them by a service
- Example:
  - User enters the word “HELLO”
  - Service will return “OLLEH”

Solution: Client-server model

- The user enters input to the client
  - eg. ‘HELLO’
- The Client process requests the server to perform logic to reverse the input
- The Server process handles the request and sends the response (result) back to the client
  - eg. ‘OLLEH’



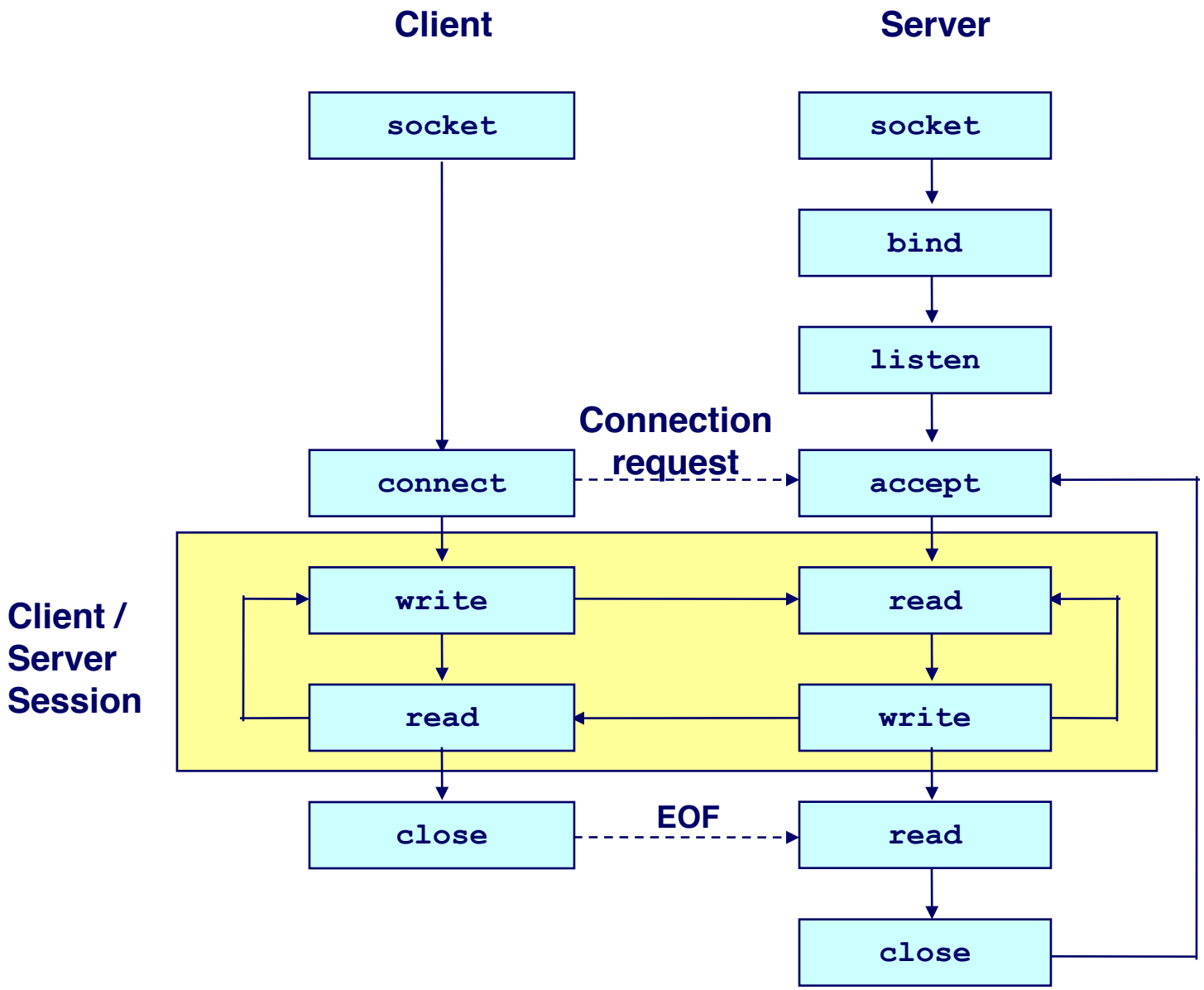
Server overview

1) Create a socket with the `socket()` system call
2) Bind the socket to an address using the `bind()` system call
3) Listen for connections with the `listen()` system call
4) Accept a connection with the `accept()` system call
5) Send and receive data

Client overview

1) Create a socket with the `socket()` system call
2) Connect the socket to the address of the server using the `connect()` system call
3) Send and receive data

Client-server communication overview



Server: Step 1

- Create a socket with the `socket()` system call

```
int socket(int domain, int type, int protocol);
```

  - *domain* can either be AF_INET (IPv4) or AF_INET6 (IPv6)
  - *type* can either be SOCK_STREAM (TCP) or SOCK_DGRAM (UDP)
  - *protocol* specifies the protocol, usually 0.

  - If successful, returns **socket file descriptor**, otherwise, returns -1

Server Step 1 - socket()

```
int sockfd;

/* Step 1 : create a socket with the socket() system call */
sockfd = socket(AF_INET, SOCK_STREAM, 0);

if (sockfd < 0) {
    printf("Error: Unable to open socket\n");
    exit(1);
}
```

Server: Step 2

- Bind the socket to an address using the `bind()` system call
  - Binding a socket means associating and reserving a port number for use by the socket

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

  - *sockfd* is the socket file descriptor (returned by socket())
  - *addr* is a pointer to the structure `struct sockaddr` which contains the host IP address and port number to bind to
  - *addrlen* is the length of what `addr` points to

  - If successful, returns 0, otherwise, returns -1

Server Step 2 - bind()

```
int sockfd, bindret;
struct sockaddr_in serv_addr;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(SERVER_PORT);

/* Step 2 : Bind the socket to an address using the bind() system call */
bindret = bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

if (bindret < 0) {
    printf("Error: Unable to bind\n");
    exit(1);
  }
```

Server: Step 3

- Listen for connections with the `listen()` system call

```
int listen(int sockfd, int backlog);
```

  - *`sockfd`* is the socket file descriptor (returned by `socket()`)
  - *`backlog`* is the maximum number of pending connections to this socket
    - `SOMAXCONN` is defined as the number of maximum pending connections allowed by the operating system
  - If successful, returns 0, otherwise, returns -1

Server Step 3 - listen()

```
 /* Step 3 : Listen for connections with the listen() system call */

if (listen(sockfd, SOMAXCONN) < 0){
    printf("Error: Unable to bind\n");
    exit(1);
}
```

Server: Step 4

- Accept a connection with the `accept()` system call

```
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

  - *`sockfd`* is the socket file descriptor (returned by socket())
  - *`addr`* is a pointer to the structure `struct sockaddr` which will contain the details of the peer socket
  - *`addrlen`* is a pointer to the length of what `addr` points to

  - If successful, returns non-negative **socket file descriptor**, otherwise, returns -1

Server Step 4 - accept()

```
struct sockaddr_in cli_addr;
socklen_t clilen;

clilen = sizeof(cli_addr);

/* Step 4 : Accept a connection with the accept() system call */
clientfd = accept(sockfd, (struct sockaddr *)&cli_addr, &clilen);

if (clientfd < 0) {
   printf("Error: Unable to accept\n");
   exit(1);
}
```

Server: Step 5

- **Send** and receive data

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

  - *sockfd* is the socket file descriptor (returned by socket())
  - *buf* is a pointer to buffer to be sent
  - *len* is the length of buffer to be sent
  - *flags* is bitwise OR of zero or more options
  - *dest_addr* is a pointer to the structure `struct sockaddr` which will contain the details of the peer socket
  - *addrlen* is a pointer to the length of what `dest_addr` points to

  - If successful, returns number of characters sent, otherwise, returns -1

Server: Step 5

- **Send** and receive data

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

- `send()` is used in connection-oriented sockets (TCP)
- `sendto()` is used in non-connection-oriented sockets (UDP)
- `send(sockfd, buf, len, flags);` is equivalent to `sendto(sockfd, buf, len, flags, NULL, 0);`
- `send(sockfd, buf, len, 0);` is equivalent to `write(sockfd, buf, len);`

Server Step 5 - send()

```
char buffer[DEFAULT_STRLEN];
int s;

/* Step 5 : Send and receive data */
s = send(clientfd, buffer, strlen(buffer), 0);

if (s < 0) {
  printf("Error: Unable to write to socket\n");
  exit(1);
}
```

Server: Step 5

- Send and **receive** data

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

  - *sockfd* is the socket file descriptor (returned by socket())
  - *buf* is a pointer to buffer to be sent
  - *len* is the length of buffer to be sent
  - *flags* is bitwise OR of zero or more options
  - *dest_addr* is a pointer to the structure `struct sockaddr` which will contain the details of the peer socket
  - *addrlen* is a pointer to the length of what `dest_addr` points to

  - If successful, returns number of characters received, otherwise, returns -1
  - If peer socket is shutdown/closed, will return 0

C and C++

Server: Step 5

- Send and **receive** data

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

  - `recv()` is used in connection-oriented sockets (TCP)
  - `recvfrom()` is used in non-connection-oriented sockets (UDP)
  - `recv(sockfd, buf, len, flags);` is equivalent to `recvfrom(sockfd, buf, len, flags, NULL, 0);`
  - `recv(sockfd, buf, len, 0);` is equivalent to `read(sockfd, buf, len);`

Server Step 5 - recv()

```
char buffer[DEFAULT_STRLEN];
int r;

/* Step 5 : Send and receive data */
r = recv(clientfd, buffer, strlen(buffer) , 0);

if (r < 0) {
  printf("Error: Unable to read from socket");
  exit(1);
}
```

Extra Sever Step : Reverse Logic

- We need a funciton for the server to reverse the input given by the user
- We create a separate function which will be called once an input has been received from the client
- We are able to call the function using the following:

```
/* call service function to reverse the input */
  reverse_input(buffer, 0, strlen(buffer)-1);
```

Extra Server Step – reverse_input()

```
void reverse_input(char *word, int begin, int end){
  char c;

  if (begin > end){
     return; }

     c = *(word+begin);
     *(word+begin) = *(word+end);
     *(word+end) = c;

     reverse_input(word, ++begin, --end);
}
```

Client: Step 1

- Create a socket with the `socket()` system call

- Same as server step 1

Client Step 1 - socket()

```
int sockfd;

/* Step 1 : create a socket with the socket() system call */
sockfd = socket(AF_INET, SOCK_STREAM, 0);

if (sockfd < 0) {
    printf("Error: Unable to open socket");
    exit(1);
}
```

Client: Step 2

- Connect the socket to the address of the server using the `connect()` system call
  - This step is only required for connection-oriented sockets (TCP)

```
int connect(int sockfd, const struct sockaddr *addr,
socklen_t addrlen);
```

  - *`sockfd`* is the socket file descriptor (returned by socket())
  - *`addr`* is a pointer to the structure `struct sockaddr` which will contain the details of the server socket
  - *`addrlen`* is a pointer to the length of what `addr` points to

  - If successful, returns 0, otherwise, returns -1

Client Step 2 - connect()

```
 int sockfd;
struct sockaddr_in serv_addr;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(SERVER_PORT);

/* Step 2 : Connect the socket to the address of the server using the
                                                        connect() system call */

if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
    printf("Error: Unable to connect\n");
    exit(1);
}
```

Client: Step 3

- Send and receive data

- Same as server step 5

Client Step 3 - send()

```
/*reset the buffer*/
memset(buffer, 0, DEFAULT_STRLEN);

printf("Please enter the message: ");
scanf("%[^\n]%*c", buffer);          // allows us to enter multiple words with whitespace

/* Step 3 : Send and receive data */
s = send(sockfd,buffer,strlen(buffer), 0);

if (s < 0) {
    printf("Error: Unable to write to socket\n");
    exit(1);
}
```

Client Step 3 - recv()

```
int r;

/* Step 3 : Send and receive data */
r = recv(sockfd, buffer, strlen(buffer), 0);

if (r < 0) {
    printf("Error: Unable to read from socket\n");
    exit(1);
}
```

Example : NC

- Run just the server
  - ./server
- NC hostname port number
  - nc localhost 5001
- Why can’t we exit?
  - Our server needs to deal with ‘\n’ for NC
- gethostbyname()
  - Alvin will go over next week