Week 11 Lecture 2

NWEN 241
Systems Programming

Alvin C. Valera

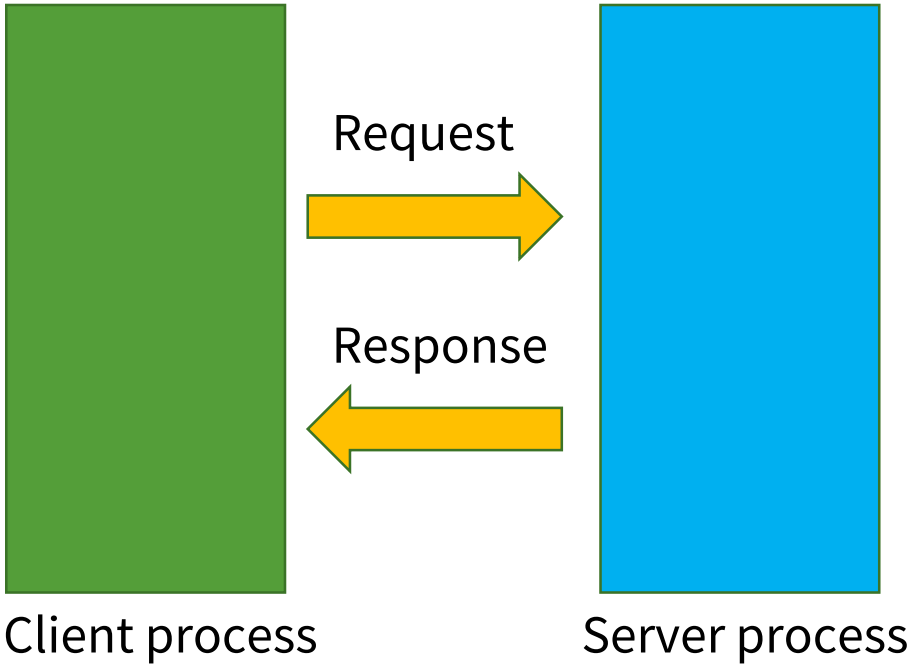
alvin.valera@ecs.vuw.ac.nz

Content

- Socket programming

Recap: Client-server model

- Based on the producer-consumer model of process cooperation
- Client makes the request for some resource or service to the server process
- Server process handles the request and sends the response (result) back to the client



Recap: Client-server model

- Client process needs to know the existence and the address of the server
- However, the server does not need to know the existence or address of the client prior to the connection
- Once a connection is established, both sides can send and receive information

Recap: Client-server communication

- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)
- **Sockets**

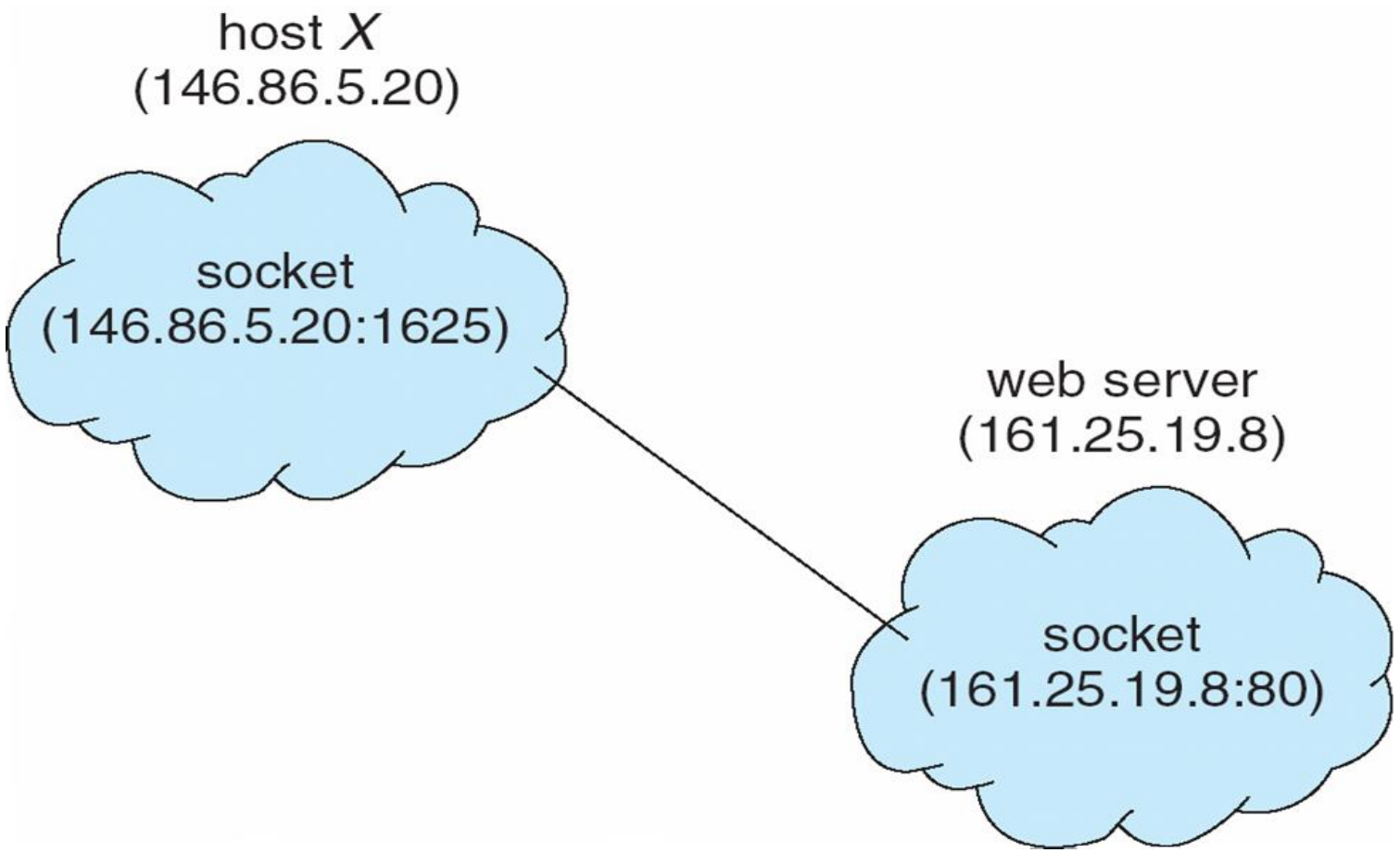
What is socket?

- A **socket** is defined as an endpoint for communication

- Concatenation of **IP address** and **port** – a number included at start of message packet to differentiate network services on a host

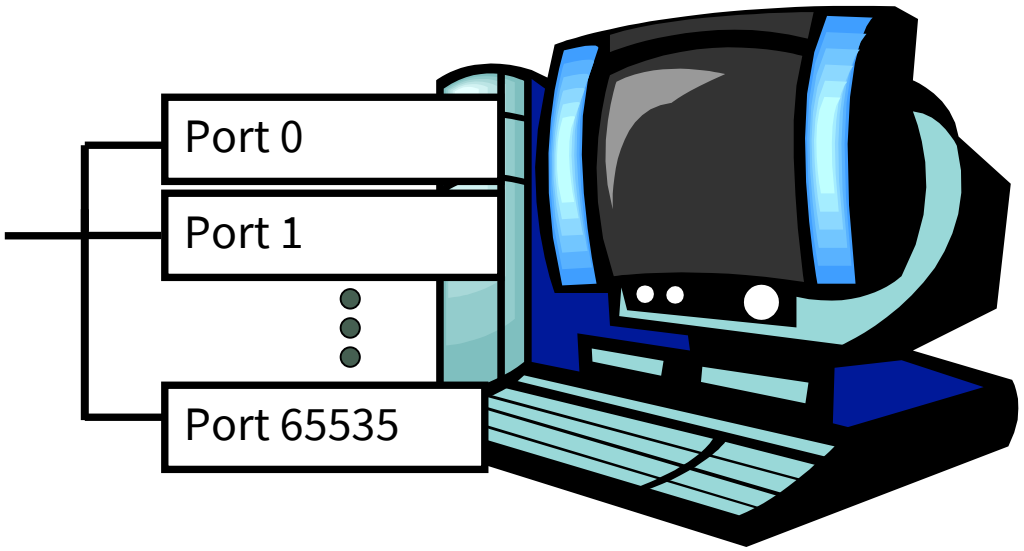
- Example:
  - The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**

Socket communication



Port numbers

- Each host has 65,536 ports
- Use of ports 1-1024 requires privileges
- Some ports are reserved for specific apps
  - 20, 21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



Sockets as programming interface

- An interface between application and network
  - The application creates a socket
  - The socket type dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless

Socket types

- `SOCK_STREAM`
  - a.k.a. **TCP**
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional

- `SOCK_DGRAM`
  - a.k.a. **UDP**
  - unreliable delivery
  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive

System calls

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `send() / sendto()`
- `recv() / recvfrom()`

Defined in

`sys/types.h`
`sys/socket.h`

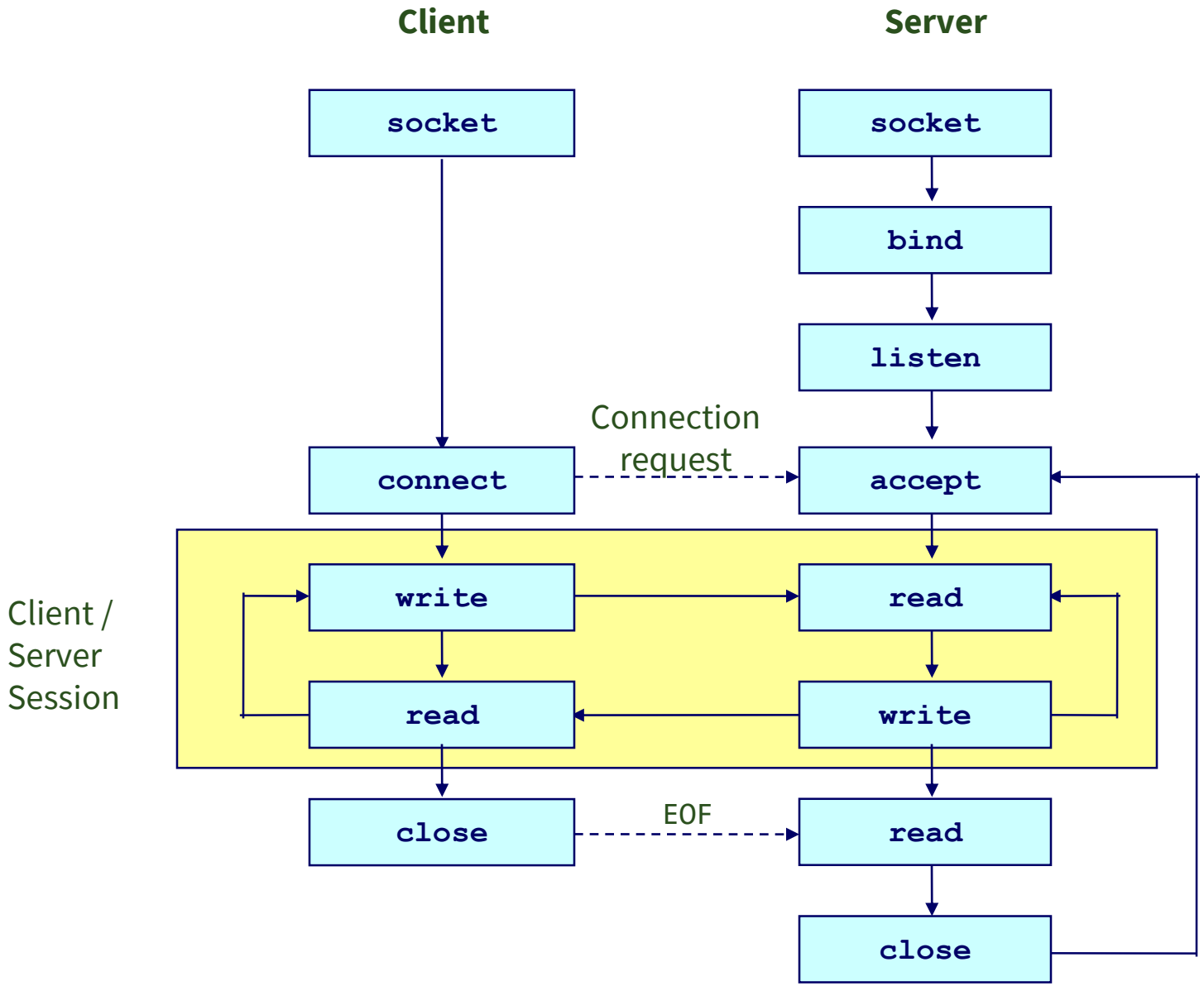
Server overview

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 example

- Create TCP socket

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd == -1) {
    printf("Error creating socket");
    exit(0);
}
```

- Create UDP socket

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd == -1) {
    printf("Error creating socket");
    exit(0);
}
```

Server: step 2

- Bind the socket to an address using the `bind()` system call
  - Binding 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

struct sockaddr

- Uses `struct sockaddr_in` in IPv4

```
struct sockaddr_in {
    short sin_family;          // e.g. AF_INET
    unsigned short sin_port;   // port number
    struct in_addr sin_addr;   // address
    char sin_zero[8];          // padding to be same size
                               // as struct sockaddr
};

struct in_addr {
    unsigned long s_addr;     // IPv4 address
};
```

Server: step 2 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; // any address
addr.sin_port = htons(12345);

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr))<0) {
    printf("Error binding socket");
    exit(0);
}
```

Host and network byte order

- Remember the little-endian and big-endian issue?

- Byte ordering also matters in network communication
  - Host and network may differ in byte ordering

- Functions for converting between host and network byte order:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

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 allow for 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 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

if(listen(fd, SOMAXCONN) < 0) {
    printf("Error listening for connections");
    exit(0);
}
```

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 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
struct sockaddr_in client_addr;
int addrlen = sizeof(client_addr);

int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                       (socklen_t*)&addrlen);
if(client_fd < 0) {
    printf("Error accepting connection");
    exit(0);
}
```

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 example using send()

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
int client_fd = accept(fd, (struct sockaddr *)& client_addr,
                           (socklen_t*)&addrlen);
…

char msg[] = "hello, world";
ssize_t r = send(client_fd, msg, strlen(msg), 0);
if(r < 0) {
    printf("Error sending message");
    close(client_fd);
    exit(0);
}
```

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

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 example using `recv()`

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
int client_fd = accept(fd, (struct sockaddr *)& client_addr,
                           (socklen_t*)&addrlen);
…

char incoming[100];
ssize_t r = recv(client_fd, incoming, 100, 0);
if(r <= 0) {
    printf("Error sending message");
    close(client_fd);
    exit(0);
}
// Do something with receiving message
printf("Received message: %s", incoming);
```

Client: step 1

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

- Same as server step 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 3

- Send and receive data
- Same as server step 5

Closing a socket

- Socket must be closed after its use

```
int shutdown(int sockfd, int how);
```

```
int close(int sockfd);
```

  - *sockfd* is the socket file descriptor (returned by socket())
  - *how* can either be SHUT_RD (further receptions disallowed), SHUT_WR (further transmissions disallowed), or SHUT_RDWR (further receptions and transmissions disallowed)

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

Next lecture

- More on socket programming