Client-Server Model

and Socket Programming

Miaoqing Huang
University of Arkansas
Outline

1. Client-Server Model
2. Socket Programming
Outline

1 Client-Server Model

2 Socket Programming
Client-Server Model

- A distributed application structure
  - Partition tasks between the providers of a resource or service, called servers, and service requesters, called clients
- One of the central ideas of network computing
  - Examples: *email exchange*, *web access*, and *database access*
Asymmetric Relationship

**Specialization**
- Clients specialize in user interface
- Servers specialize in managing data and application logic

**Sharing**
- Many clients can be supported by few servers
- Client predominately makes requests, server makes replies
Email Example

Client

- Email client sends message to server

Server

- Message is stored on POP server

Client

- Later, recipient’s email client retrieves message from server
Chat Room Example

Chat clients send user’s typing to server

Chat server receives typing from all users and sends to all clients

Other user’s clients display aggregated typing from chat server
Outline

1. Client-Server Model
2. Socket Programming
What is a socket? (socket ←→ telephone)
Socket vs. Telephone

**Socket (aka, Internet Socket)**
- An endpoint of a bidirectional inter-process communication flow across an Internet Protocol-based computer network
- A socket address consists of
  - An IP address: the location of the computer
  - A port: mapped to the application program process
- Communication protocol: TCP or UDP

**Telephone**
- An endpoint of a bidirectional inter-person communication flow across a telephone network
- An address of a telephone consists of
  - A telephone number
  - (Possibly) an extension
- Mobile phone standard: GSM or CDMA
IP Address and Port Number

- **Internet Protocol address**
  - A numerical label assigned to each device (e.g., computer, printer) participating in a computer network that uses the Internet Protocol for communication
  - Serve two principal functions
    - Host or network interface identification
    - Location addressing
  - Current IP version: IPv4

- **Port number**
  - A port is an application-specific or process-specific software construct serving as a communications endpoint
  - A specific port is identified by its number, i.e., the port number
  - A port number is a 16-bit unsigned integer, [0, 65535]
    - Well-known ports: 0-1,023
    - Registered ports: 1,024-49,151
    - Dynamic, private or ephemeral ports: 49,152-65,535
An IPv4 address (dotted-decimal notation)

172  .  16  .  254  .  1

10101100  .  00010000  .  11111110  .  00000001

One byte = Eight bits

Thirty-two bits ( 4 * 8 ), or 4 bytes
Two Types of Sockets

- Stream socket, aka, connection-oriented socket
  - Establish a connection before transferring data
    - Reliable, in-order
  - Use *Transmission Control Protocol (TCP)*

- Datagram socket, aka, connectionless socket
  - Each packet sent or received on a Datagram socket is individually addressed and routed
    - May arrive in any order
    - May get lost
  - Use *User Datagram Protocol (UDP)*
Inter-process Communication using Datagram Socket (UDP)

**Client**
1. Create a socket \((s1)\)
2. Bind the socket to any available port of the local computer
   - This step may be skipped when programming in C
3. Send message to a known socket address \((s2)\)
   - Receive message from \(s1\)

**Server**
1. Create a socket \((s2)\)
2. Bind the socket to an agreed socket address
3. Receive message from the socket \((s2)\)
   - Send message to \(s1\)
Client

1. Create a socket (s1)
2. Send a connection request to a known socket address (s2)
3. Start writing and reading through the connection between s1 and s3

Server

1. Create a socket (s2)
2. Bind the socket to an agreed socket address
3. Listen to and accept the connection request from s2
   - Create a new socket s3 for the connection to s1
4. Start reading and writing
Following system calls are used:

- **socket**: to create a socket and get a file descriptor for it
- **bind**: to bind a socket address to the file descriptor of a socket
- **sendto**: send a message through a bound socket to a socket address
- **recvfrom**: to receive a message through a socket
- **close**: to destroy the socket when it is no longer needed
Following system calls are used:

- **socket**: to create a socket and get a file descriptor for it
- **bind**: to bind a socket address to the file descriptor of a socket
- **connect**: to make a connect request
- **listen**: to specify how many requests should be queued
- **accept**: to accept a connect request
- **write**: send information via connected sockets
- **read**: receive information via connected sockets
- **close**: to destroy the socket when it is no longer needed
Create a Socket

```c
int socket (int domain, int type, int protocol); // prototype

int s;
if(( s = socket(AF_INET, SOCK_DGRAM, 0))<0) {
    perror("socket failed");
    return;
}
```

- **domain**
  - **Must be** `AF_INET`

- **type**
  - `SOCK_DGRAM`: Datagram socket
  - `SOCK_STREAM`: Stream socket

- **protocol**
  - **Specify the protocol**
  - `0`: system to select a suitable protocol
Bind a Socket

**int** `bind (int s, struct sockaddr * socketAddress, int addrlength);`

**struct sockaddr_in {**
  **short sin_family;** // Must be AF_INET
  **u_short sin_port;** // Port number
  **struct in_addr sin_addr;** // IP address; INADDR_ANY: local IP
  **char sin_zero[8];**
}**

- **s**
  - Socket descriptor
- **socketAddress**
  - Specify the socket address
  - Use `struct sockaddr_in`
- **addrlength**
  - The size of the structure in the second argument
- *Server has to bind the socket to an known socket address*
Close a Socket

```c
int close (int s);
```

- `s` 
  - Socket descriptor

A socket has to be closed so that it can be re-used by another program
Send a Message using a Datagram Socket

```
int sendto(int s, char * msg, int len, int flags, struct sockaddr *to, int tolen);
```

- **s**
  - Socket descriptor
- **msg**
  - Pointer to the message string
- **len**
  - Length of the message
- **flags**
  - Normally zero
- **to**
  - Destination Socket
- **tolen**
  - Size of the structure `sockaddr`
**Receive a Message using a Datagram Socket**

```c
int recvfrom(int s, char *buf, int len, int flags, struct sockaddr * from, int *fromlen);
```

- **s**
  - Socket descriptor

- **buf and len**
  - The buffer to save the received message

- **flags**
  - Normally zero

- **from and fromlen**
  - The structure to save the sender’s socket address

- `recvfrom` is a blocking call
Listen to a Socket and Accept a Connection Request

Server functions of a Stream Socket

```c
int listen(int s, int backlog);
```

- Listen on its socket for requests from clients for connections
  - `backlog`: the number of requests for connection that can be queued at that socket

```c
int accept(int s, struct sockaddr * clientAddress, int * clientLength);
```

- Accepts the first connection in the queue at socket `s`
  - The result is a descriptor for a new socket that has been created for use as one end of the stream
Listen to a Socket and Accept a Connection Request

Server functions of a Stream Socket

```c
int listen(int s, int backlog);
```

- Listen on its socket for requests from clients for connections
  - `backlog`: the number of requests for connection that can be queued at that socket

```c
int accept(int s, struct sockaddr * clientAddress, int * clientLength);
```

- Accepts the first connection in the queue at socket `s`
  - The result is a descriptor for a new socket that has been created for use as one end of the stream

How to deal with multiple connections simultaneously?
Listen to a Socket and Accept a Connection Request
Server functions of a Stream Socket

```c
int listen(int s, int backlog);
```

- Listen on its socket for requests from clients for connections
  - `backlog`: the number of requests for connection that can be queued at that socket

```c
int accept(int s, struct sockaddr * clientAddress, int * clientLength);
```

- Accepts the first connection in the queue at socket `s`
  - The result is a descriptor for a new socket that has been created for use as one end of the stream

How to deal with multiple connections simultaneously?
- Spawn a thread for each connection
Request a Connection by a Client Process

```c
int connect(int s, struct sockaddr *server, int addrLen);
```

- Request a connection via the socket address of the listening process
Read and Write via a Connection

```c
int write(int s, char * message, int msglen);
```

- Write a message into the connection

```c
int read(int s, char * buffer, int bufsize);
```

- Read a message from the connection