Socket Programming in C

CSIS0234/COMP3234 Lecture Four

Contents

- What is Berkeley Unix Socket?
- Connectionless and Connection (Connection-oriented) communication modes
- Client-Server Communication
- Socket functions
  - socket(), bind(), listen(), accept(), connect(), send(), recv(), sendto(), recvfrom(), close()
  - Network Byte Order routines
  - Address Conversion Routines
    - gethostbyname(), getaddrinfo()
  - Other utilities

Learning Outcome

- [**ILO4 - Implementation**] be able to demonstrate knowledge in using Socket Interface to design and implement network protocols

References

- Beej's Guide to Network Programming
  - http://beej.us/guide/bgnet/

- Computer Networking – A Top-Down Approach Featuring the Internet, 6th edition by J. Kurose et. al
  - Section 2.1

- W. Stevens, UNIX Network Programming, Volume 1, Prentice Hall
  - Chapter 3 & 4

Communication

- Communication always happens between two parties
- In daily life, we have many modes of communication
  - e.g., by phone
  - e.g., by post
- Always one party be the "active" one who initiates the communication

Using the telephone

- Caller enters callee's phone number
- Caller initiates the connection
- Callee accepts the connection
- They start the communication
- Either party could end the connection

Using postal service

- Sender writes receiver's address on envelop
- Sender drops the letter into posting box
- Postal service picks up the letter
- Postman delivers the letter to receiver's address
- Sender does not know whether receiver will receive the letter

Berkeley UNIX Socket API

- Berkeley sockets allows you to write network applications on top of TCP or UDP
  - provides generic access to process-to-process (end-to-end) communications services, which is good for students to learn the principles of protocols and distributed applications by hands-on program development
- Was originally designed
  - For BSD Unix
- Now
  - Industry standard
  - Available on almost all operating systems

What is a socket ?

- It is an OS abstraction – a communication endpoint
  - Being created dynamically by the program during runtime
    - Persists only while application runs
  - Under UNIX, socket is integrated with I/O system
    - Uses the "open-read-write-close" paradigm as the familiar way that we handle a file
  - An open socket is referenced by a file descriptor
    - Small integer that identifies the socket
    - Generated by OS when socket created
    - One per active socket
    - Used by the application on all communication operations through the socket

Available Service Mode

- **Connection-oriented**
  - Create the local endpoint
    - Get a mobile phone with your SIM card
  - Identify the remote endpoint
    - Find your remote partner's telephone no.
  - Initiate a connection
    - Call your partner
    - Accept by your partner
  - Send and Receive data
    - Start the chat
  - Terminate a connection

- Uses by **TCP**
- First, setup logical connection between two peer application processes
- Reliable bidirectional in-sequence transfer of byte stream
- Multiple write/read between peer processes
- Finally, connection release

Available Service Mode (2)

- **Connectionless**
  - Create the local endpoint
    - You have a mailbox be available for receiving mails
  - Identify the remote endpoint
    - Write down the destination address of the recipient on the envelop
  - Send out your mail
    - Using the postal service
  - Deliver to the destined mailbox

- Uses by **UDP**
  - Destination address has to be written in each message (letter)
  - Immediate transfer of one block of information
    - No setup overhead & delay
  - Send/receive to/from multiple peer processes
  - Best-effort service only
    - Possibly out-of-order
    - Possibly loss

Client-Server Communication

- One application
  - Begins execution (**be online**) first
  - **Waits passively** at a **known** location
    - **IP address and port number** must be publicly known by caller/sender
      - just like you must know the phone # or postal address of your peer
    - IP address tells where the end-system is at; port number tells which process in that end-system involves in the communication
  - ***Passive*** program called a **server**
- Another application
  - Begins execution later
  - **Actively contacts** (initiate connection or send letter to) first program
  - ***Active*** program called a **client**
- This interaction mode is used by all network applications (including the P2P applications)

Characteristics of Server & Client

- Servers
  - Special-purpose program which provides some services
  - Waits passively for clients to contact
  - Accepts requests from arbitrary clients
  - Can handle multiple remote clients simultaneously
- E.g., Web server (Apache)

- Clients
  - Arbitrary application program which actively initiates contact with a server
    - Some clients can be the servers for other clients
  - Request data or service provided by the server
  - usually direct interact or control by user
- E.g., Web browser (Firefox)

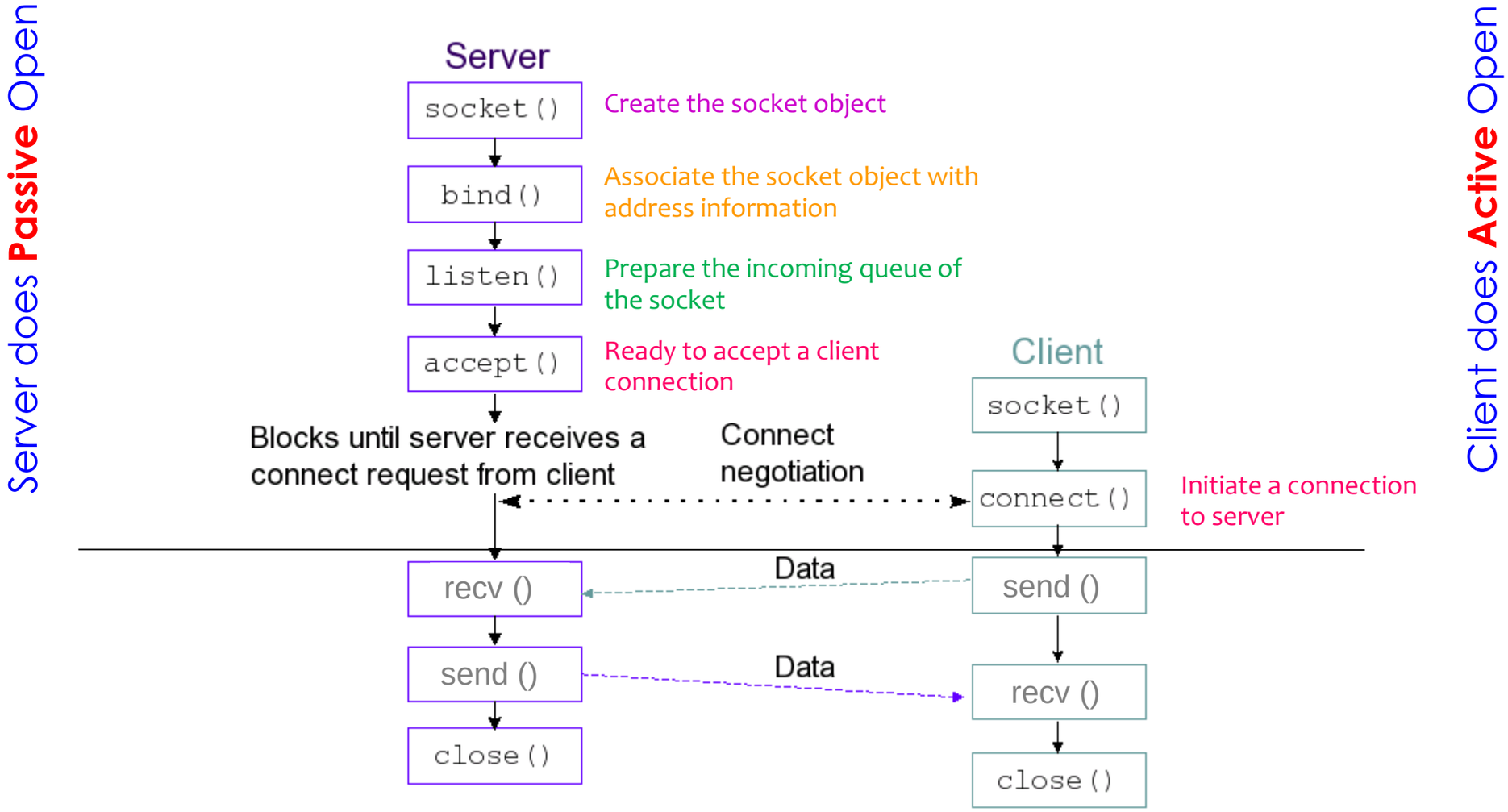
A Service

- Each network service must get a unique port number P
  - e.g., HTTP - 80, Telnet - 23, SSH - 22, FTP - 21 (20), . . .
  - **This port # (service) must be known by all clients**
- Server
  - When a server is up running, it informs OS it is using port P
    - **No other application within the same machine** can use the same port P
  - Then, it waits for client requests to arrive
- Client
  - Usually we don't care which port # it uses on client's end
    - randomly assigned one port (*ephemeral* port) but must be unique within the client's machine
  - Constructs request to the server's port P
    - Send request to port P on server computer

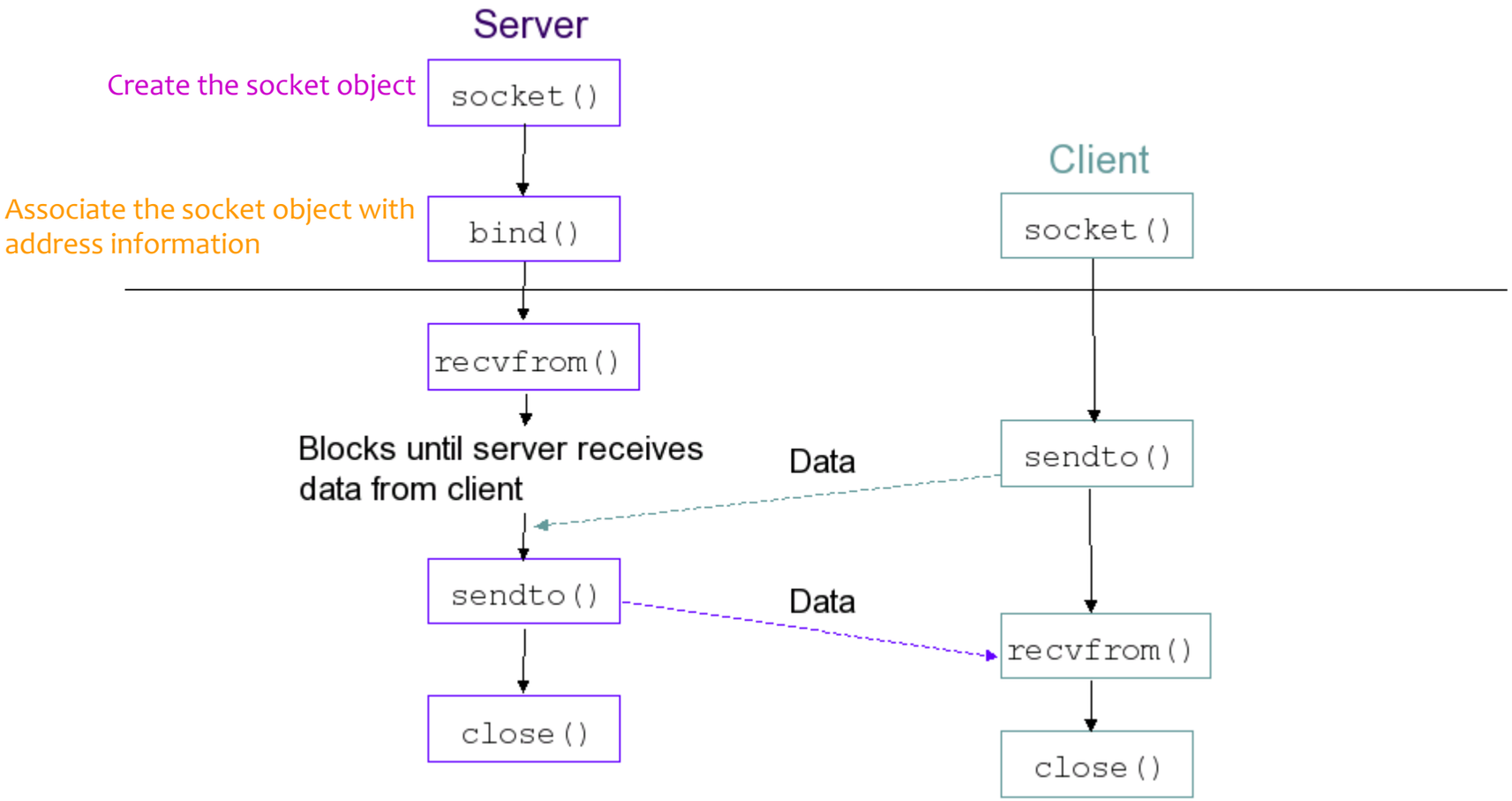
Port Number Assignment

- Valid Range: 0 to 65535 (of size 16 bits)
- Divide into three ranges:
  - Well-known ports: **0 to 1023**
    - Reserved ports
    - Can only use by
      - system processes
      - privileged users
  - Registered ports: **1024 to 49151**
    - Available for ordinary user processes
    - Also allow to register with IANA (Internet Assigned Number Authority)
  - Dynamic and/or Private ports: **49152 to 65535**

Socket Calls for Connection-Oriented Mode



Socket Calls for Connectionless Mode



Create a Socket – socket()

- Reference:
  - http://beej.us/guide/bgnet/output/html/multipage/socketman.html
- int sockfd = socket (int family, int type, int protocol);
  - OS returns descriptor for socket to send data, receive data, or both
  - Descriptor valid until application closes socket or exits
- For Internet networking
  - family = AF_INET
  - type = SOCK_STREAM or SOCK_DGRAM
  - protocol – normally is safe to set to 0 (as default) for most cases

UDP

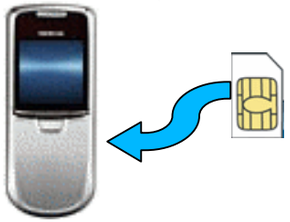
```
socket(AF_INET, SOCK_DGRAM, 0);
```

TCP

```
socket(AF_INET, SOCK_STREAM, 0);
```

Associate the socket object with address information

- Reference:
  - http://beej.us/guide/bgnet/output/html/multipage/bindman.html
- int bind (int sockfd, struct sockaddr * localaddr, int addrlen);
  - Assign a socket address to the newly created socket
    - includes a protocol port number AND the IP address used by the host
    - stores the socket address in a variable of type struct sockaddr
- Two uses of bind
  - **Server registers its socket address** with the system; which must be known to the clients
    - So the IP address + Port # becomes its phone no., somebody can call it now
  - Client can register a specific address for itself [Optional]
    - Normally, client doesn't need to explicitly call bind as the system automatically assigns that to the client



For our programming project, the client process has to explicitly call bind() to register its socket address

Accepting Incoming Calls – TCP server

- Reference:
  - http://beej.us/guide/bgnet/output/html/multipage/listenman.html
  - http://beej.us/guide/bgnet/output/html/multipage/acceptman.html

- Listen
  - Used by server when using Stream socket
  - Prepares socket to accept incoming connections
  - int listen(int sockfd, int req_queuesize);
    - req_queuesize specifies the max. no. of incoming connection requests that can be queued while waiting for server to accept them

- Accept
  - Used by server when using Stream socket
  - After calling accept(), the (server) process will be blocked waiting for new incoming connection; once the connection is established, it **returns a new socket** (descriptor) to server
    - **The server uses the new socket** to communicate with the client instead of using the original socket (sockfd)
  - int accept(int sockfd, struct sockaddr * client_addr, int * addrlen);

How to Connect to a Remote Host?

- Reference:
  - http://beej.us/guide/bgnet/output/html/multipage/connectman.html
- Connect
  - Used by client
    - Like pick up the phone and dial the number
  - Either
    - Stream Socket - Performs a request for TCP connection setup
      - Establish a connection to a server that has called accept() and wait
    - Datagram Socket - Fully specifies addresses for UDP
      - **No connection is setup !!**
        - Just like you write the same address on all envelops
      - Allow the client to send many messages to the same registered address

Seldom used

But we do use this in the programming project

  - int connect(int sockfd, struct sockaddr * peer_addr, int addrlen);

Let's Exchange Messages !

- Send & Sendto
  - Transmit outgoing data
- Recv & Recvfrom
  - Receive incoming data

```
int send(int sockfd,
         const void * out_msg,
         int msg_len,
         int flags);
```

For most cases, set it to zero

TCP

```
int recv(int sockfd,
         void * in_buf,
         int buf_len,
         int flags);
```

UDP can use these functions if connect() is used to register the peer's addr info

```
int sendto(int sockfd,
           const void * out_msg,
           int msg_len,
           int flags,
           const struct sockaddr * to_addr,
           int to_len);
```

Give the addressing info of destination

UDP

```
int recvfrom(int sockfd,
             void * in_buf,
             int buf_len,
             int flags,
             struct sockaddr * from_addr,
             int * from_len);
```

Give the addressing info of sender

Terminate

- Close
  - Close the socket permanently
  - int **close**(int sockfd);

struct sockaddr – Socket Addresses

- As sockets can be used with arbitrary protocols
  - Each protocol has its own addressing scheme

- Generic Address Format

16 bytes

```
struct sockaddr {
    sa_family_t     sa_family;      // address family, AF_xxx
    char            sa_data[14];    // 14 bytes of protocol address
};
```

- For TCP/IP protocol, the **IPv4 address** format

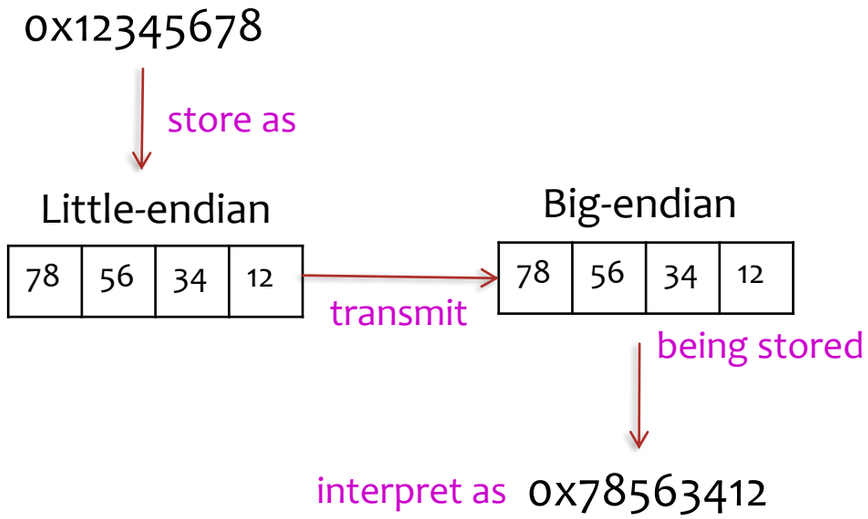
16 bytes

```
struct sockaddr_in {
    sa_family_t     sin_family;     // Address family, AF_INET
    in_port_t       sin_port;       // 16-bit Port number  (must in Network Byte Order)
    struct in_addr  sin_addr;       // Internet address  (must in Network Byte Order)
    char            sin_zero[8];    // Unused; usually pad with zeros
};

// Internet address (a structure for historical reasons)
struct in_addr {
    in_addr_t    s_addr;            // that's a 32-bit long in network byte order
};
```

What is Byte Ordering ?

- Problem:
  - Different machines use different word orderings
    - Little-endian
    - Big-endian
  - These machines may communicate with one another over the network



- Example: the number
  0x12345678
- Little endian machine
  - Lower bytes first
  - Stores the four bytes as:

byte location	content
184	78
185	56
186	34
187	12

- Big endian machine
  - Higher bytes first
  - Stores the four bytes as:

byte location	content
184	12
185	34
186	56
187	78

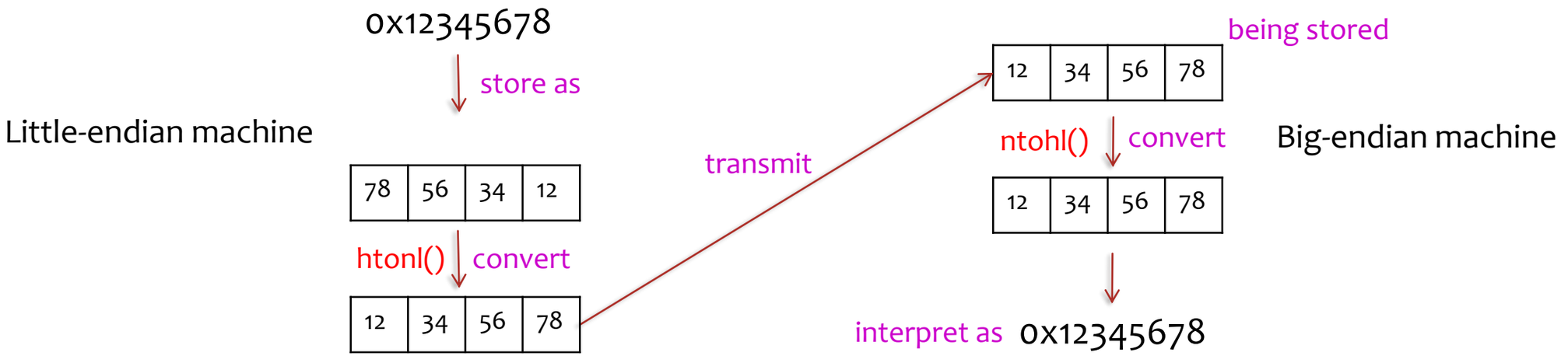
What is Byte Ordering ?

- Solution:
  - For any data objects carry in a message that have **data unit size larger than a byte**, we need to perform byte order conversion
    - e.g., short integer, long integer, floating point, double, . . .
    - How about the text string? e.g., "hello world"
      - The basic data unit of a string is the character, which is of 1-byte
  - When transmitting message, first do the conversion of all data contents with a unit size larger than a byte to Network Byte Order (that is big-endian) when sending via the network
  - Convert it back at the receiving end

Byte Ordering Conversion Routines

- Before sending out
  - uint32_t **htonl** (uint32_t hostlong);
  - uint16_t **htons** (uint16_t hostshort);
- After receiving
  - uint32_t **ntohl** (uint32_t netlong);
  - uint16_t **ntohs** (uint16_t netshort);

host
to
network
long



Example Program

If you don't care, you can tell the system to automate this

```
// choose an unused port # at random
 my_addr.sin_port = htons(0);
// use any IP address of current host
 my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in  my_addr;
```

Create socket

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    // do some error checking!
```

Set up the socket address structure

```
    my_addr.sin_family = AF_INET;                          // host byte order
    my_addr.sin_port = htons(MYPORT);                      // u_int16_t, network byte order
    my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");   // u_int32_t, network byte order
    memset(&(my_addr.sin_zero), '\0', 8);                  // zero the rest of the struct
```

Assign the address to the socket

```
    // don't forget your error checking for bind():
    bind(sockfd,  (struct sockaddr *)&my_addr,  sizeof(struct sockaddr_in));
    .
    .
```

Looking up network address (for IPv4, still valid)

- Giving a hostname or IP address, convert it to 32-bit network byte order format
- Reference:
  - http://beej.us/guide/bgnet/output/html/multipage/gethostbynameman.html
- struct hostent * gethostbyname (const char * name);
  - name – passing the hostname or IP address in standard dot notation
    - e.g., "i.cs.hku.hk" or "147.8.179.14"
  - returns a pointer which points to a struct hostent that contains information about the target host, particularly the list of IP addresses of that target host

```
/* For client wants to find out addressing info of server i.cs.hku.hk */

struct hostent * he = gethostbyname("i.cs.hku.hk");       // lookup i.cs.hku.hk

struct sockaddr_in peer_addr;
    // fill up the sockaddr_in structure for connecting to i.cs.hku.hk port 80
peer_addr.sin_family = AF_INET;
peer_addr.sin_port = htons(80);
peer_addr.sin_addr = *((struct in_addr *)he->h_addr);      // the 1st address of this host
memset(&(peer_addr.sin_zero), 0, 8);  // zero the rest of the struct
```

Server and Concurrency

- Iterative (Sequential) server
  - Only accepts (& handles) one client connection at a time
  - There can have multiple client requests waiting for the service in the listen queue
    - A client in the tail of the queue has to wait for all previous requests to be processed before being served
  - Unacceptable to users if long request blocks short requests

- Concurrent server
  - Can handle multiple client requests at a time
  - Server creates new thread of control (e.g. pthread) to handle each request
  - Client only waits for its request to be processed

A Typical Scenario - A Concurrent Server Using Stream Socket

```
#define MYPORT 34567
#define BACKLOG 10

main() {
        int sockfd, new_fd;             // listen on sockfd, new connection on new_fd
        struct sockaddr_in my_addr;     // my address information
        struct sockaddr_in their_addr;  // client's address information
        int sin_size;

        sockfd = socket(AF_INET, SOCK_STREAM, 0);    // do some error checking!

        my_addr.sin_family = AF_INET;                // host byte order
        my_addr.sin_port = htons(MYPORT);      // short, network byte order
        my_addr.sin_addr.s_addr = htol(INADDR_ANY);      // auto-fill with my IP
        memset(&(my_addr.sin_zero), '\0', 8);        // zero the rest of the struct

        // don't forget your error checking for these calls:
        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
        listen(sockfd, BACKLOG);

        for ( ; ; ) {
                        sin_size = sizeof(struct sockaddr_in);
                        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
                        if (new_fd < 0)
                                fprintf(stderr, "accept error\n");

                        if (fork() == 0) {
                                close(sockfd);                  // within child process
                                . . . .                         // do some useful work
                                . . . .                         // comm with client using new_fd
                                exit(0);
                        }
                        close(new_fd);                  // within original process
                }
}
```

Summary

- The socket interface is a set of declaration, definitions, and procedures for writing client-server programs

- To use TCP, we need to create a stream socket
- To use UDP, we need to create a datagram socket

- Network communication uses big-endian as the network byte order for transmitting of non-character data units

- To use a TCP server socket, these functions are involved
  - socket() bind() listen() accept() [read() write()]* close()

- To communicate with a TCP server, clients would invoke
  - socket() connect() [write() read()]* close()

Summary (2)

- To use a UDP server socket, these functions are involved
  - socket() bind() [recvfrom() sendto()]* close()
- For the UDP client, these functions are involved
  - socket() [sendto() recvfrom()]* close()
- A TCP server usually creates multiple child processes/threads to handle many clients simultaneously
- Once a TCP connection is made, the server switches the dialogue to a different socket to free up the main socket for additional incoming calls

Backup Slides

Address Conversion Routines (for IPv4, still valid)

- Reference:
  - http://beej.us/guide/bgnet/output/html/multipage/inet_ntoaman.html
- in_addr_t inet_addr (const char * ip_addr);
  - Input argument: IP address in a dots-and-number string
  - returns the addr in Network Byte Order and stores in the in_addr_t
    - in_addr_t is an integer value to place in struct in_addr
- int inet_aton (const char * ip_addr, struct in_addr * inp);
  - another way to do the conversion
- char * inet_ntoa (struct in_addr in);
  - just the reverse

Looking up network address (IPv4 and IPv6)

- Giving a hostname or (IPv4 / IPv6) address, compose the sockaddr structure for us automatically
  - a better way for address conversion as it supports both IPv4 and IPv6 addressing formats
- Reference:
  - http://beej.us/guide/bgnet/output/html/multipage/getaddrinfoman.html
- int getaddrinfo (const char * node, const char * port, const struct addrinfo * hints, struct addrinfo **result);
  - node – give the hostname or IPv4 / IPv6 address to be looked-up OR NULL
  - port – give which port number or service name to be used for connecting to the host (i.e., the service port number) OR NULL
  - hints – use by caller to provide specific information (stored in an addrinfo structure) to the system for setting the returned sockaddr structures or set to NULL for default setting
  - result – return a linked-list of addrinfo structures, with each addrinfo structure contains a (pointer to) sockaddr structure for each valid IP address of target host

```
struct addrinfo {
    int ai_flags;               /* AI_PASSIVE, AI_CANONNAME */
    int ai_family;              /* AF_xxx */
    int ai_socktype;            /* SOCK_xxx */
    int ai_protocol;            /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
    socklen_t ai_addrlen;               /* length of ai_addr */
    char *ai_canonname;                 /* ptr to canonical name for host */
    struct sockaddr *ai_addr;           /* ptr to socket address structure */
    struct addrinfo *ai_next;           /* ptr to next structure in linked list */
};
```

/* For client wants to find out addressing info of server i.cs.hku.hk */

```
struct addrinfo hints, *result;

/* Fill in the hints addrinfo structure */
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;          // using IPv4
hints.ai_socktype = SOCK_STREAM;    // using TCP

/* Looking up the information about the server and filling up the */
/* sockaddr_in structures in the result addrinfo linked-list      */
getaddrinfo("i.cs.hku.hk", "80", &hints, &result);

struct sockaddr_in * peer_addr = (struct sockaddr_in *) result->ai_addr; // the 1st address of this host
                                        :
                                        :
freeaddrinfo(result);
```

Other Utility Functions

- Who are you ?
  - getpeername( ) - get remote address and remote port of a connected stream socket
- Who am I ?
  - getsockname( ) - get local address and local port of a socket
- What is my name ?
  - gethostname( ) - get the hostname of current machine
- Name looking up
  - gethostbyaddr( ) - given the IP address and lookup the host name
- Socket has many parameters and options
  - getsockopt( )
  - setsockopt( )
- select( )
  - a powerful tool to blocked-wait for multiple active sockets at the same time
- fcntl( )
  - use for setting up non-blocking waiting mode of the socket