Network Socket Programming - II

BUPT/QMUL

2021-03-25

Review

- Basic Concepts in network programming
  - Process
  - File descriptor
  - System call
  - ~~Signal~~
- Some Helpful Points
  - Client-Server Model
  - Connections

Agenda

- Week 2
  - Basic Concepts in Network Programming
  - Review of Some Helpful Points
- Week 4
  - Structures About IP Address and DNS
  - Sockets Interface
  - Major System Calls
  - Sample Programs

Structures about
IP address and DNS

A programmer's view of the Internet

- 1. Hosts are mapped to a set of 32-bit *IP addresses*.
  - 202.112.96.163

- 2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*.
  - 202.112.96.163 is mapped to www.bupt.edu.cn

- 3. A process on one Internet host can communicate with a process on another Internet host over a *connection.*

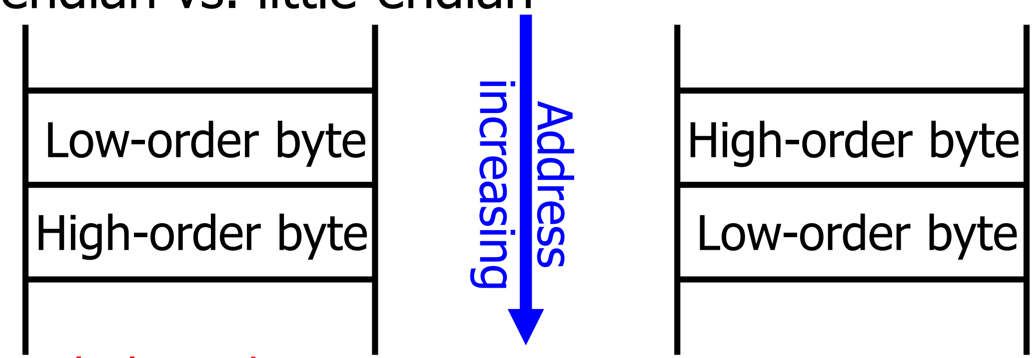
IP Addresses (1)

- 32-bit IP addresses are stored in an *IP Address structure*
  - <netinet/in.h>

```
/* Internet address.  */
typedef uint32_t in_addr_t;
struct in_addr {
    in_addr_t s_addr;
  };
```

```
/*Defined in <stdint.h>*/
typedef unsigned int uint32_t;
```

- Two ways to store multi-byte integers
  - Big-endian vs. little-endian



little-endian

big-endian

IP Addresses (2):
Host byte order vs. network byte order

- Host byte order is machine-dependent
  - You can see it in <bits/endian.h>
  - A program used to output the host byte order
- Network byte order is machine-independent (*big-endian*)
- Byte order conversion functions
  - **htonl**: host byte order → network byte order for ***long int***
  - **htons**: host byte order → network byte order for ***short int***
  - **ntohl**: network byte order → host byte order for ***long int***
  - **ntohs**: network byte order → host byte order for ***short int***

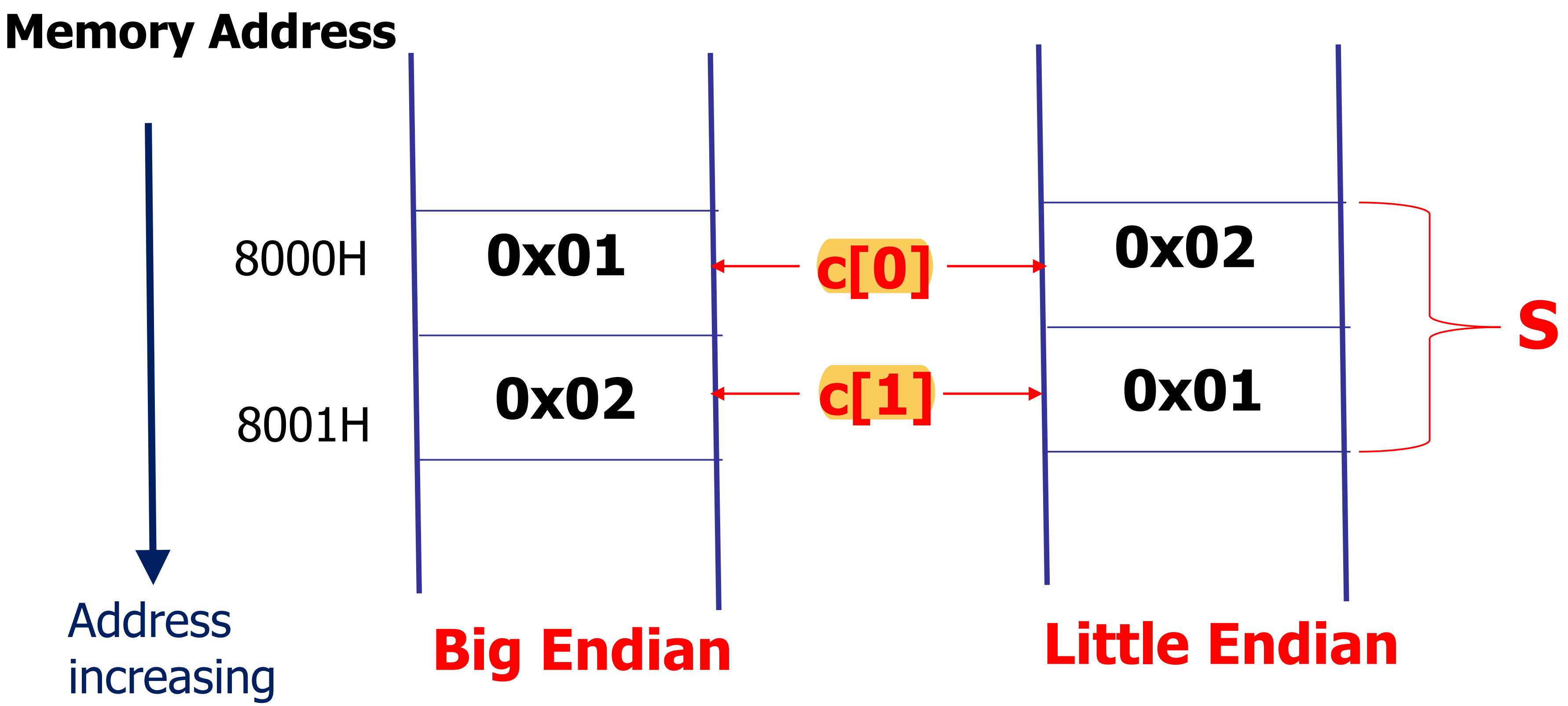
IP Addresses (3)
-- A program used to output the host byte order

- byteorder.c

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        union {
                short s;
                char c[sizeof(short)];
        }un;

        un.s=0x0102;

        if (sizeof(short)==2){
                if (un.c[0]==1 &&un.c[1]==2)
                        printf("big-endian\n");
                else if  (un.c[0]==2 &&un.c[1]==1)
                        printf("littlt-endian\n");
                else
                        printf("Unknow\n");
        }else
                printf("sizeof(short)=%d\n",sizeof(short));

        exit(0);
}
```

a **union** is like a **struct**, only all the data members sit at the same memory location.
This means only one of them can be used at a time.

Storage example of Variable *s* and *c*



Domain Name System (1)

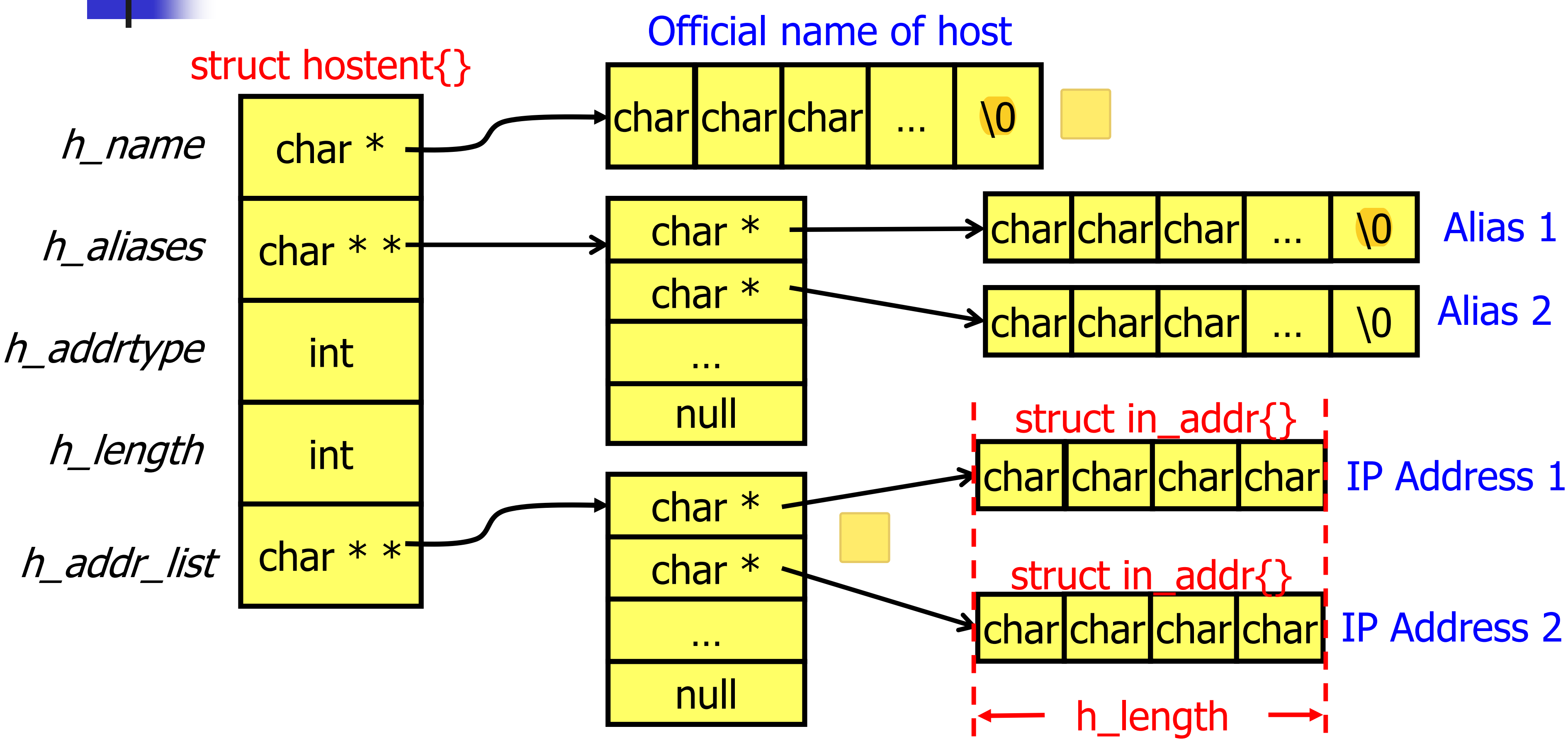
- The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS*.
  - Conceptually, programmers can view the DNS database as a collection of millions of *host entry structures*
  - <netdb.h>

Domain Name System (2): Host Entry Structure

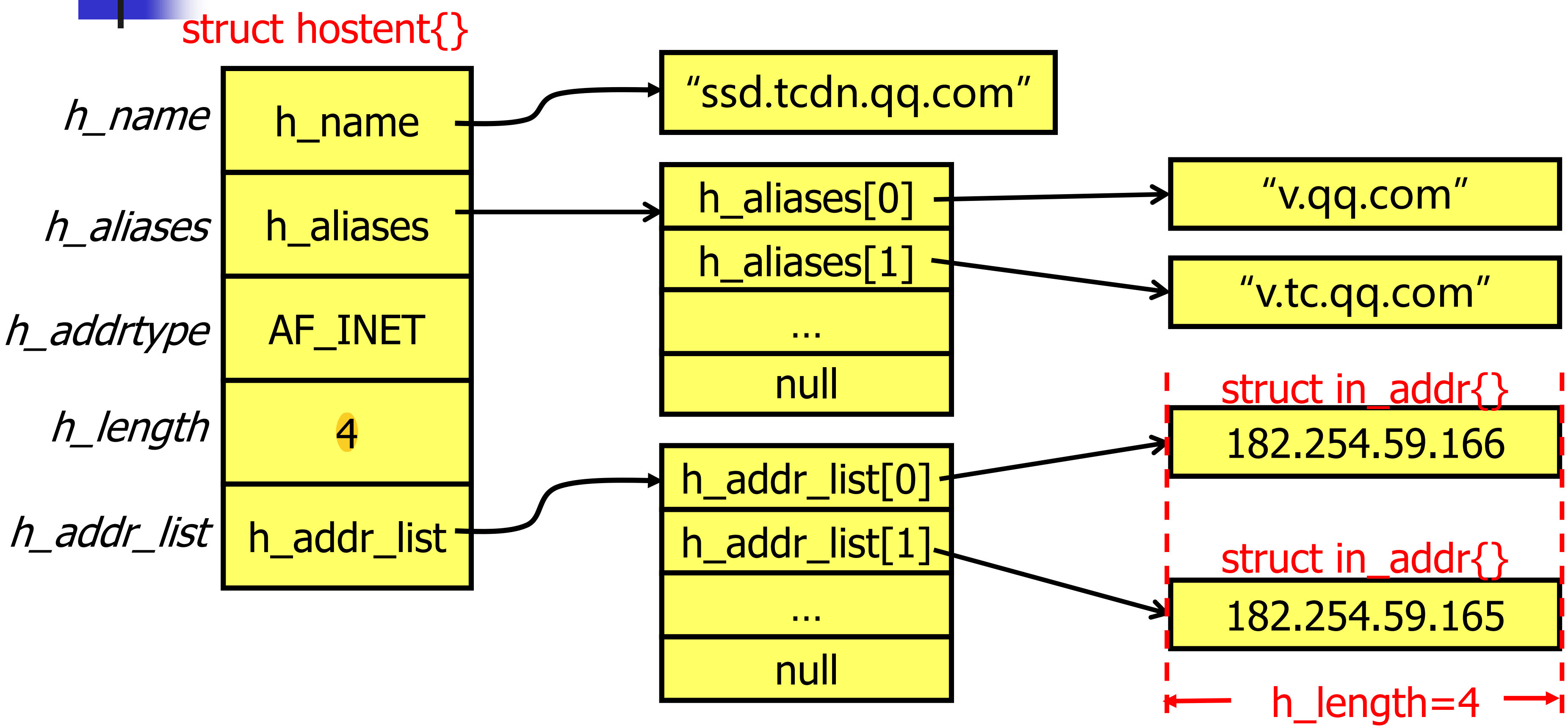
h_addr_list:
An array of pointers to IP addresses for the host (in network byte order), terminated by a NULL pointer.

```
/* Description of data base entry for a single host. */
struct hostent
{
  char *h_name;              /* Official name of host.  */
  char **h_aliases;          /* Alias list.  */
  int h_addrtype;            /* Host address type.  */
  int h_length;              /* Length of address.  */
  char **h_addr_list;       /* List of addresses from name
                               server.*/
#define h_addr  h_addr_list[0]  /* The first address in
                                   the address list.  */
};
```

Domain Name System (3): Host Entry Structure



Domain Name System (4): Example of Host Entry Structure



You can try the command: *nslookup v.qq.com*

Domain Name System (5)

- Functions for retrieving host entries from DNS
  - gethostbyname: query key is a DNS domain name.

```
#include <netdb.h>
struct hostent * gethostbyname (const char *hostname);
```

  - gethostbyaddr: query key is an IP address.

```
#include <netdb.h>
struct hostent * gethostbyaddr (const char *addr, int len, int family );
```

AF_INET for IPv4

Socket Interface

Sockets Interface

- Functions
- Definitions
- Types

Sockets Interface – functions

- Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols
- Provides a user-level interface to the network
- Underlying basis for all Internet applications
- Based on client/server programming model

Sockets Interface – definitions(1)

- What is a socket?
  - To the kernel, a socket is an endpoint of communication.
  - To an application, a socket is a **file descriptor** that lets the application read/write from/to the network.
    - Remember: All Unix I/O devices, including networks, are modeled as files.
- Clients and servers communicate with each other by reading from and writing to socket descriptors.
- The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors.

File Model in Unix/Linux

- In Unix/Linux, all I/O devices are treated as files
  - Identified with File Descriptors
  - File operations
    open
    close
    lseek
    read
    write
    …

Sample File Descriptor Table (One per Process)

0	stdin
1	stdout
2	stderr
3	file
4	device
5	socket
..	…

Sockets Interface – definitions(2)

- **Internet-specific socket address** (bits/socket.h)

```
struct sockaddr_in {
   unsigned short sin_family; /* address family (always AF_INET) */
   unsigned short sin_port; /* port num in network byte order */
   struct in_addr sin_addr; /* IP addr in network byte order */
   unsigned char sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

一定转换成network by order

- **Address family**: Domains refer to the area where the communicating processes exist. Commonly used domains include:
  - AF_UNIX: for communication between processes on one system;
  - AF_INET (IPv4): for communication between processes on the same or different systems using the DARPA standard protocols (IP/UDP/TCP)
  - AF_INET6 (IPv6)
  - AF_LOCAL (Unix domain)
  - AF_UNSPEC (the importance will be explained later)
  - …

Sockets Interface – definitions(3)

- **Generic socket address** (<sys/socket.h>)

```
struct sockaddr {
   unsigned short sa_family; /* protocol family */
   char sa_data[14]; /* protocol-specific address,
                        up to 14 bytes. */
};
```

- **Protocol family**
  - PF_LOCAL: Local to host, pipes and file-domain
  - PF_UNIX: Old BSD name for PF_LOCAL
  - **PF_INET: IP protocol family**
  - PF_AX25: Amateur radio AX.25
  - PF_IPX: Novell internet protocol
  - PF_INET6: IP version 6
  - PF_ATMSVC: ATM SVCs
  - PF_APPLETALK: Appletalk DDP
  - …

Sockets Interface – definitions(4)

- Generic socket address and Internet-specific socket address
  - Pointer to generic socket address is used for address arguments to **connect(), bind() and accept()**
  - Must cast Internet-specific socket address (`sockaddr_in *`) to generic socket address (`sockaddr *`) for connect, bind, and accept

```
struct sockaddr_in serv;
/* fill in serv{}*/
bind (sockfd, (struct sockaddr *)&serv , sizeof(serv));
```

Socket Address

Generic socket address

```
struct sockaddr {
    unsigned short sa_family;       /* PF_INET for IPv4 */
    char sa_data[14];            /* protocol-specific address,
                                    up to 14 bytes. */
};
```

Internet-specific socket address

```
struct sockaddr_in{
     unsigned short  sin_family;  /* AF_INET */
     unsigned short  sin_port;    /* 16-bit port number */
                                  /* Network Byte Order*/
     struct in_addr sin_addr;     /* 32-bit IP Address */
                                  /* Network Byte Order */
     char            sin_zero[8]; /* unused */
}
```

Sockets Interface – types(1)

- Stream Socket
  - Service: reliable (i.e. sequenced, non-duplicated, non-corrupted) bidirectional delivery of byte-stream data
  - Metaphor: a phone call
  - int s = socket (PF_INET, SOCK_STREAM, 0);
- Datagram Socket
  - Service: unreliable, unsequenced datagram
  - Metaphor: sending a letter
  - int s = socket (PF_INET, SOCK_DGRAM, 0);
- Raw Sockets Service
  - allows user-defined protocols that interface with IP
  - Requires `root` access
  - Metaphor: playing with an erector set
  - int s = socket (PF_INET, SOCK_RAW, protocol);
- **SOCK_STREAM and SOCK_DGRAM are the most common types of sockets used within UNIX/Linux**

Sockets Interface – types(2)

- Reliably-delivered Message Socket
  - Service: reliable datagram
  - Metaphor: sending a registered letter
  - Similar to datagram socket but ensure the arrival of the datagrams
  - int s = socket (PF_NS, SOCK_RDM, 0);
- Sequenced Packet Stream Socket
  - Service: reliable, bi-directional delivery of recordoriented data
  - Metaphor: record-oriented TCP
  - Similar to stream socket but using fixed-size datagrams
  - int s = socket (PF_NS, SOCK_SEQPACKET, 0);

Major System Calls

Socket Programming: Telephone Analogy

- A telephone call over a “telephony network” works as follows:
  - Both parties have a telephone installed.
  - A phone number is assigned to each telephone.
  - Turn on ringer to listen for a caller.
  - Caller lifts telephone and dials a number.
  - Telephone rings and the receiver of the call picks it up.
  - Both Parties talk and exchange data.
  - After conversation is over they hang up the phone.

Dissecting the Analogy

- A network application works as follows:
  - An endpoint (telephone) for communication is created on both ends.
  - An address (phone no) is assigned to both ends to distinguish them from the rest of the network.
  - One of the endpoint (receiver) waits for the communication to start.
  - The other endpoint (caller) initiates a connection.
  - Once the call has been accepted, a connection is made and data is exchanged (talk).
  - Once data has been exchanged the endpoints are closed (hang up).

In the world of sockets……

- socket() – Create endpoint for communication
- bind() - Assign a unique telephone number
- listen() – Wait for a caller
- connect() - Dial a number
- accept() – Receive a call
- send(), recv() – Talk
- close() – Hang up

System Calls

- Socket operation
- Byte order operation
- Address formats conversion
- Socket option
- Name and address operation

System Calls – Socket Operation

- socket()
  - returns a socket descriptor
- bind()
  - What address I am on / what port to attach to
- connect()
  - Connect to a remote host
- listen()
  - Waiting for someone to connect to my port
- accept()
  - Get a socket descriptor for an incoming connection
- send() and recv()
  - Send and receive data over a connection
- read(), write()
  - Read from / Write to a particular socket, similar to recv()/ send()
- sendto() and recvfrom()
  - Send and receive data without connection
- close() and shutdown()
  - Close a connection Two way / One way

System Calls – Byte Order Conversion

- htonl()
  - host byte order → network byte order for ***long int***
- htons()
  - host byte order → network byte order for ***short int***
- ntohl()
  - network byte order → host byte order for ***long int***
- ntohs()
  - network byte order → host byte order for ***short int***

System Calls – Address Formats Conversion

- **inet_aton()**
  - IP address in numbers-and-dots notation (ASCII string) → IP address structure in network byte order
- **inet_addr()**
  - same function with inet_aton()
- **inet_ntoa()**
  - IP address structure in network byte order → IP address in numbers-and-dots notation (ASCII string)
- **inet_pton()**
  - Similar to inet_aton() but working with IPv4 and IPv6
- **inet_ntop()**
  - Similar to inet_ntoa() but working with IPv4 and IPv6

System Calls – Socket Option

- getsockopt()
  - Allow an application to require information about the socket
- setsockopt()
  - Allow an application to set a socket option

- eg. get/set sending/receiving buffer size of a socket

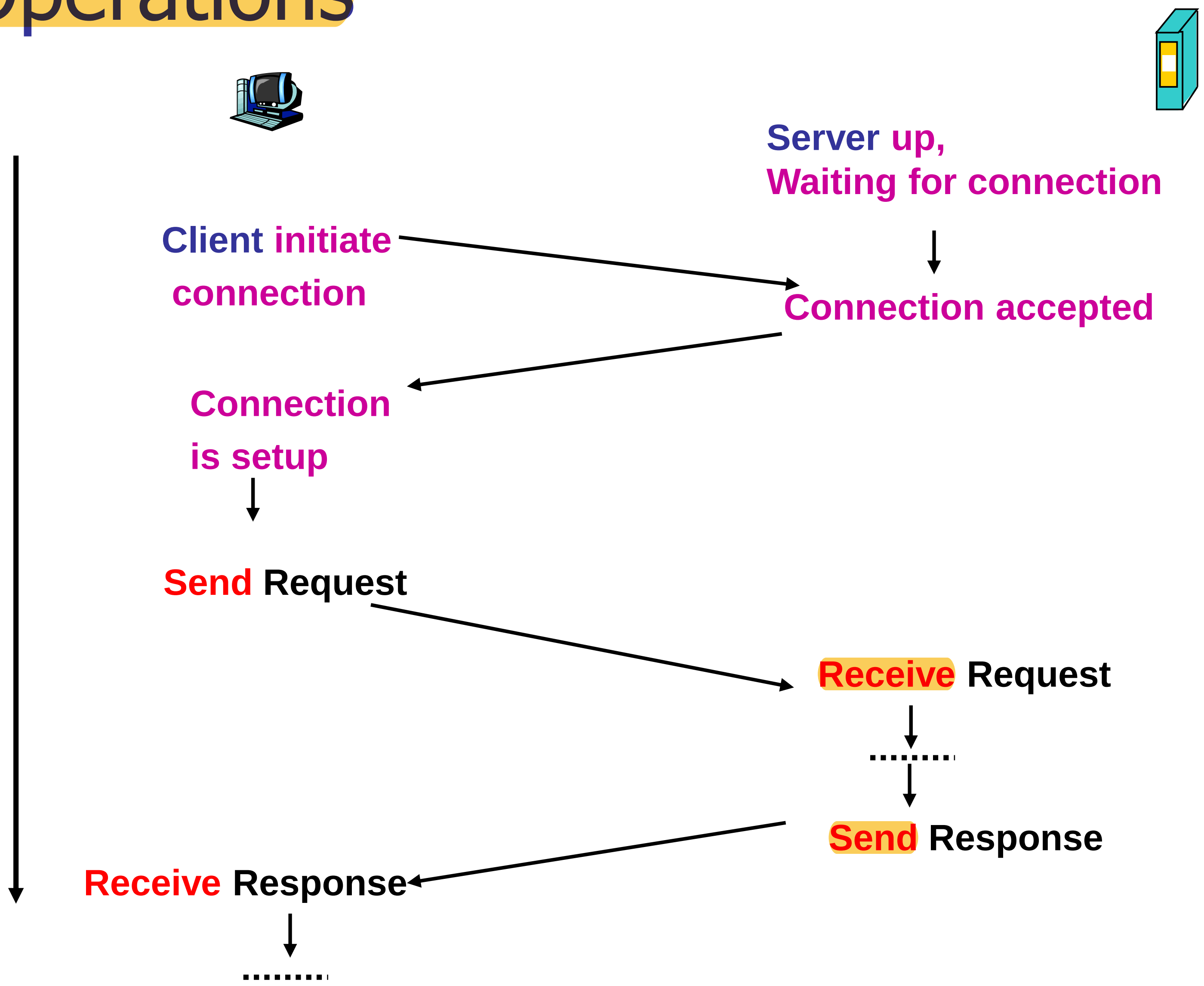
System Calls – Name and Address Operation

- **gethostbyname()**
  - retrieving host entries from DNS and the query key is a DNS domain name
- **gethostbyaddr()**
  - retrieving host entries from DNS and the query key is an IP address
- gethostname()
  - Obtaining the name of a host
- getservbyname()
  - Mapping a named service onto a port number
- getservbyport()
  - Obtaining an entry from the services database given the port number assigned to it

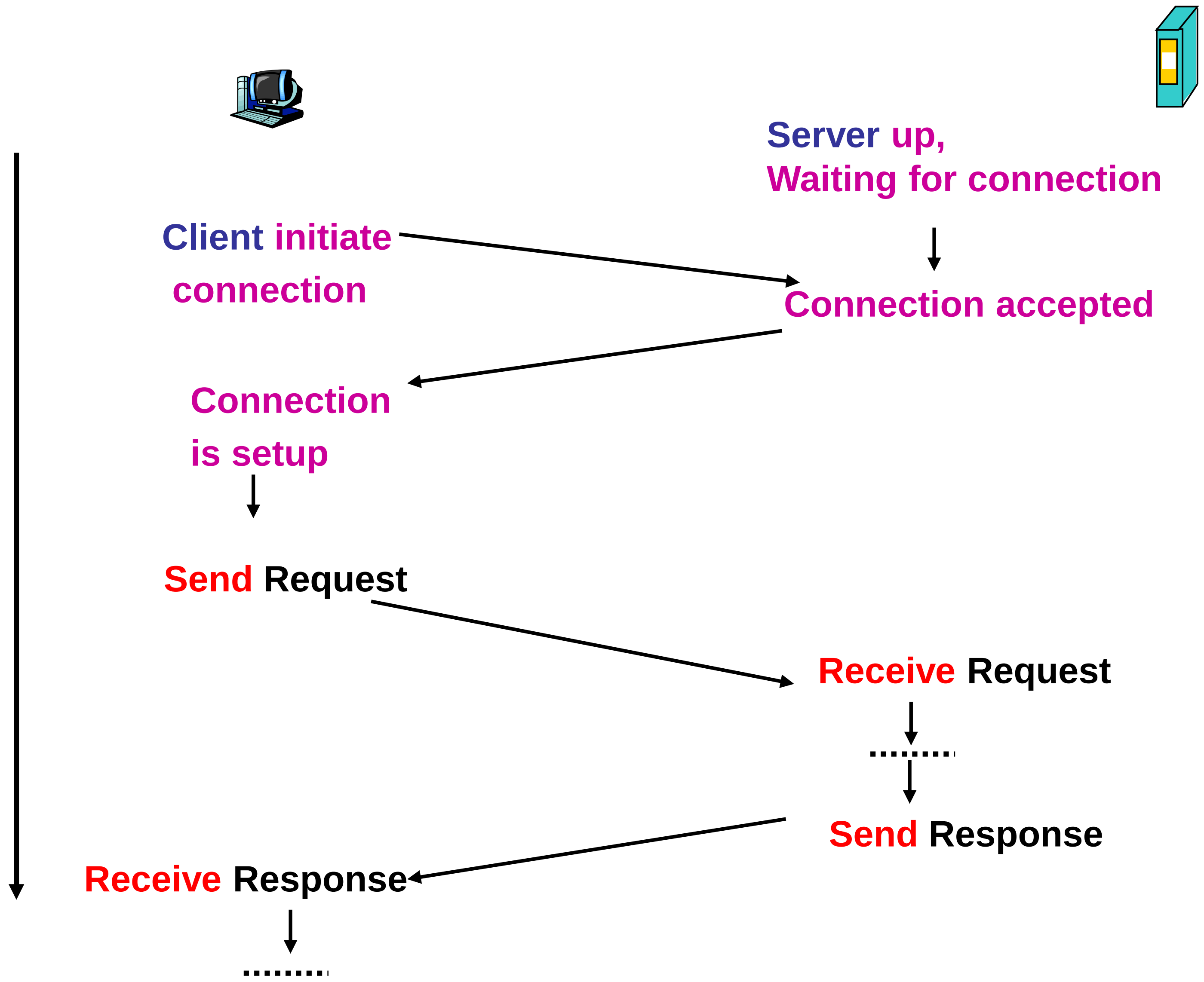
Using UDP to query Local DNS Server

Returns a pointer to struct hostent (host entry structure) on success

Process of Socket Operation:
TCP Operations



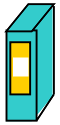
Map to TCP socket programming



Map to TCP socket programming

1

```
Sock-fd = socket( PF_INET,
      SOCK_STREAM, 0);
bind(Sock-fd, "1.2.3.4", 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,..)
```



Blocking call, waiting ….

Client initiate connection

Connection accepted

Connection is setup

Send Request

Receive Request

……….

Send Response

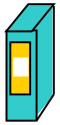
Receive Response

……….

Map to TCP socket programming

1

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
bind(Sock-fd, "1.2.3.4", 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)
```



Blocking call, waiting ….

2

```
Sock-fd = socket( PF_INET,
     SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
     AddrLen)
```

Blocking call, waiting for server to accept ….

Connection accepted

Connection is setup

Send Request

Receive Request

………

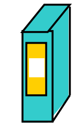
Send Response

Receive Response

………

Map to TCP socket programming

1



```
Sock-fd = socket( PF_INET,
      SOCK_STREAM, 0);
bind(Sock-fd, "1.2.3.4", 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)
```

2

```
Sock-fd = socket( PF_INET,
      SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
      AddrLen)
```

Blocking call, waiting for server to accept ….

3

Unblock, accept() returns, getting client IP address and port number

Connection is setup

Send Request

Receive Request

………

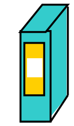
Send Response

Receive Response

………

Map to TCP socket programming

1
Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)



2
Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
AddrLen)

3
Unblock, accept() returns,
getting client IP address
and port number

4
Unblock, connect( )
will return

Send Request

Receive Request

………

Send Response

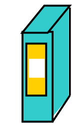
Receive Response

………

Map to TCP socket programming

1

Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
bind(Sock-fd, "1.2.3.4", 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,...)



2

Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
AddrLen)

3

Unblock, accept() returns, getting client IP address and port number

4

Unblock, connect( ) will return

5

send(Sock-fd, ReqMessage,
MsgLen,...);

Receive Request

....

Send Response

Receive Response

....

Map to TCP socket programming

1 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)

2 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
AddrLen)

3 Unblock, accept() returns, getting client IP address and port number

4 Unblock, connect( ) will return

5 send(Sock-fd, ReqMessage,
MsgLen,…);

6 recv(Sock-fd, ReqMessage,
MsgLen,…);

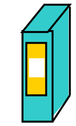
………

Send Response

Receive Response

………

Map to TCP socket programming



1 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)

2 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
connect(Sock-fd, Server_Addr,
AddrLen)

3 Unblock, accept() returns, getting client IP address and port number

4 Unblock, connect( ) will return

5 send(Sock-fd, ReqMessage,
MsgLen,…);

6 recv(Sock-fd, ReqMessage,
MsgLen,…);

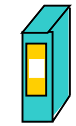
………

7 send(Sock-fd, ResMessage,
MsgLen,…);

Receive Response

………

Map to TCP socket programming



1 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
bind(Sock-fd, “1.2.3.4”, 9000);
listen(Sock-fd, QueueLen);
accept(Sock-fd, ClientAddr,…)

2 Sock-fd = socket(PF_INET,
SOCK_STREAM, 0);
Connect(Sock-fd, Server_Addr,
AddrLen)

3

Unblock, accept() returns, getting client IP address and port number

4 Unblock, connect( ) will return

5 send(Sock-fd, ReqMessage,
MsgLen,…);

6 recv(Sock-fd, ReqMessage,
MsgLen,…);

7 send(Sock-fd, ResMessage,
MsgLen,…);

8 recv(Sock-fd, ResMessage,
MsgLen,…);

System Calls – socket()

- An application calls *socket()* to create a new socket that can be used for network communication
- The call returns a descriptor for the newly created socket

```
#include <sys/socket.h>
int socket ( int family, int type, int protocol);
```

int (return value):
- Socket descriptor – success
- -1 – failed

int family — Protocol family
- **PF_INET**
- PF_INET6
- PF_UNIX
- ……

int type — Socket type
- **SOCK_STREAM**
- **SOCK_DGRAM**
- SOCK_RAW
- ……

int protocol — Specific protocol
- Often be set as **0**
- Be not 0 in raw socket

System Calls – bind()

- An application calls bind() to specify the local endpoint address (a **local IP address and protocol port number**) for a socket
- For TCP/IP, the endpoint address uses the ***sockaddr_in*** structure.
- Must cast Internet-specific socket address (`struct sockaddr_in *`) to generic socket address (`struct sockaddr *`) for bind
- **Servers** use *bind()* to specify the well-known port at which they will await connections

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

- int: 0 – success, -1 – failed
- int sockfd: Socket descriptor to be bound
- const struct sockaddr *myaddr: The local address to which the socket should be bound
- socklen_t addrlen: The length of the structure (bytes)

System Calls – listen()

- Connection-oriented **servers** call *listen()* to place a socket in *passive mode* and make it ready to accept incoming connections
- *listen()* also sets the number of incoming connection requests that the protocol software should enqueue for a given socket while the server handles another request
- It only applies to socket used with **TCP**

```
#include <sys/socket.h>
int listen (int sockfd,        int qlength);
```

0 – success
-1 – failed

Socket that should be prepared for use by a server

The length of the request queue for that socket

System Calls – accept()

- The **server** calls accept() to extract the next incoming request

new

- *accept()* creates a new socket for each new connection request, and returns the descriptor of the new socket to its caller
- *accept()* fills in the structure (*sockaddr*) with the IP address and protocol port number of the remote machine

comm

- Must cast Internet-specific socket address (`struct sockaddr_in *`) to generic socket address (`struct sockaddr *`) for accept()

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

- int: Socket Descriptor (non-zero) – success; -1 – failed
- int sockfd: Socket on which to wait
- struct sockaddr *cliaddr: The address of the client that placed the request
- socklen_t *addrlen: The length of the client address

System Calls – connect()

- After creating a socket, a **client** calls *connect()* to establish an active connection to a remote server
- Must cast Internet-specific socket address (`struct sockaddr_in *`) to generic socket address (`struct sockaddr *`) for connect

```
#include <sys/socket.h>
int connect (int sockfd, const struct sockaddr *servaddr , socklen_t addrlen );
```

- int: 0 – success, -1 – failed
- int sockfd: Socket to connect
- const struct sockaddr *servaddr: The server's address to which the socket should connect
- socklen_t addrlen: The length of the server's address (bytes)

System Calls – send()

- Both clients (to transmit request) and servers (to transmit replies) used *send()* to transfer data across a **TCP connection**
- The application passes the descriptor of a socket to which the data should be sent, the address of the data to be sent, and the length of the data
- Usually, send copies outgoing data into buffers in the OS kernel

```
#include <sys/socket.h>
ssize_t send (int sockfd, const void *buff, size_t nbytes, int flags);
```

- ssize_t: Number of bytes that are sent successfully; -1 – failed
- int sockfd: socket descriptor you want to send data to
- const void *buff: The address of the data to be sent
- size_t nbytes: The number of bytes to be sent
- int flags: The flag controlling the connection, usually 0

System Calls – recv()

- Both clients (to receive a reply) and servers (to receive a request) use *recv* to receive data from a **TCP connection**
- If the buffer cannot hold an incoming user datagram, *recv* fills the buffer and discar the remainder

```
#include <sys/socket.h>
ssize_t recv (int sockfd, void *buff, size_t nbytes, int flags);
```

- ssize_t: Number of bytes that are received successfully, -1 – failed
- int sockfd: socket descriptor to read from
- void *buff: The address in memory into which the data to be placed
- size_t nbytes: The length of the buffer area
- int flags: The flag controlling the reception, usually 0

System Calls – sendto() & recvfrom()

- Allow the caller to send or receive a message over **UDP**
- sendto() requires the caller to specify a destination
- recvfrom() uses an argument to specify where to record the sender's address

Destination address

The length of the destination address

```
#include <sys/socket.h>
ssize_t sendto (int sockfd, const void *buff, size_t nbytes, int flags,
                const struct sockaddr *to, socklen_t addrlen);
```

```
#include <sys/socket.h>
ssize_t recvfrom (int sockfd, void *buff, size_t nbytes, int flags,
                  struct sockaddr *from, socklen_t *addrlen );
```

Number of bytes that are sent or received successfully
-1 – failed

Where to record the sender's address

The length of the sender's address

Using Read and Write with sockets

- In Linux, as in most other UNIX systems, programmers can use *read* instead of *recv*, and *write* instead of *send*
  - *int read (sockfd, bptr, buflen)*
  - *int write (sockfd, bptr, buflen)*
- The chief advantage of *send* and *recv* is that they are easier to spot in the code

System Calls – close()

- Once a client or server finishes using a socket, it calls *close* to deallocate it
- Any unread data waiting at the socket will be discarded

```
#include <unistd.h>
int close (int sockfd);
```

0 – success
-1 – failed

Socket to be closed

System Calls – inet_aton() & inet_addr()

- Converts an IP address in numbers-and-dots notation into unsigned long in network byte order

```
#include <arpa/inet.h>
int inet_aton (const char *string, struct in_addr *address);
```

1 – success
0 – error

Pointer to the string that contains the numbers-and-dots notation

Pointer to IP address structure

```
#include <arpa/inet.h>
in_addr_t inet_addr (const char *string,);
```

When success: return the 32-bit address in network byte order
When failed: return INADDR_NONE

Pointer to the string that contains the numbers-and-dots notation

System Calls – inet_ntoa()

- Mapping a 32-bit integer (an IP address in network byte order) to an ASCII string in dotted decimal format

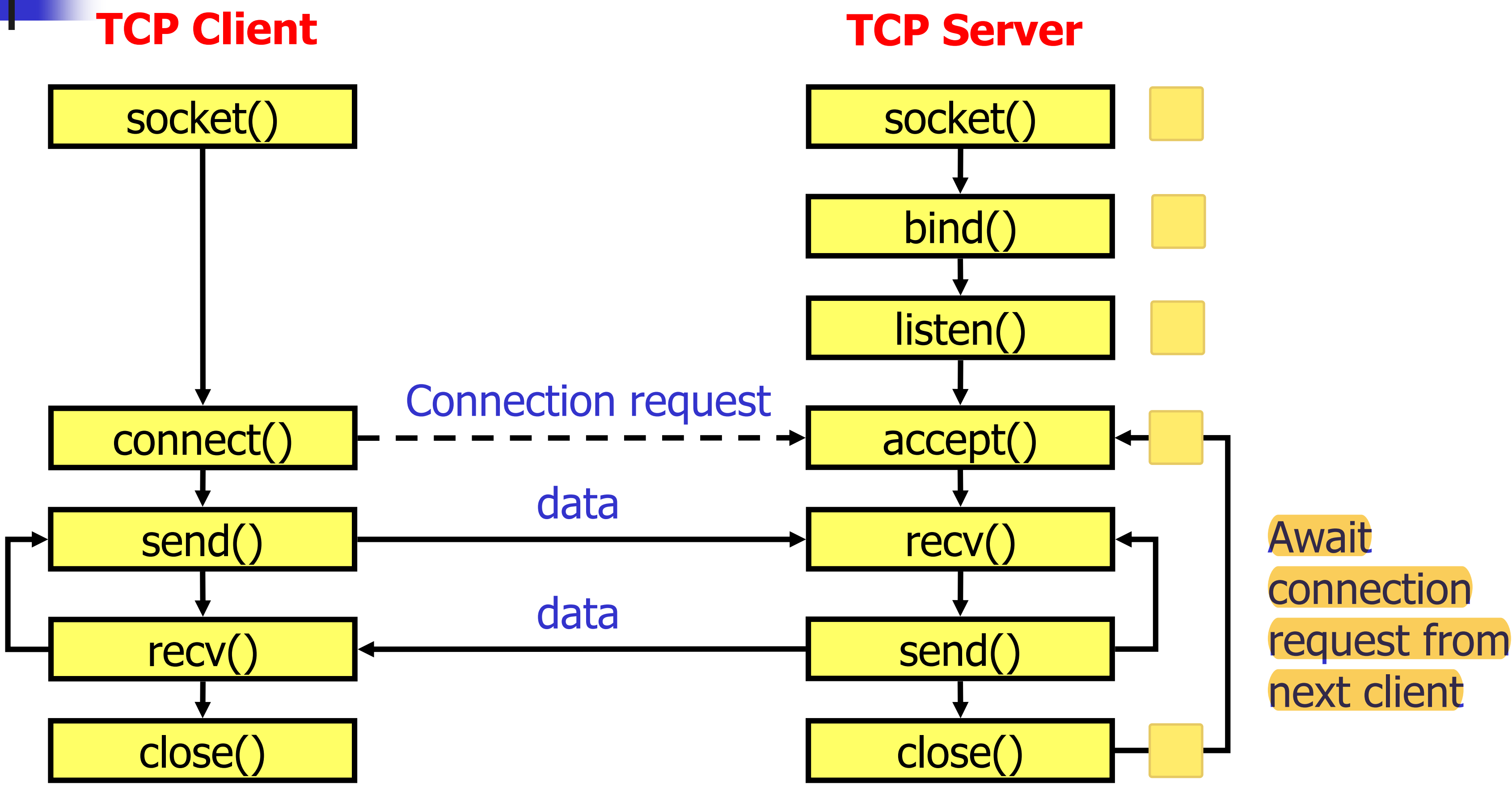
```
#include <arpa/inet.h>
char * inet_ntoa ( struct in_addr inaddr );
```

Pointer to the string contains the numbers-and-dots notation

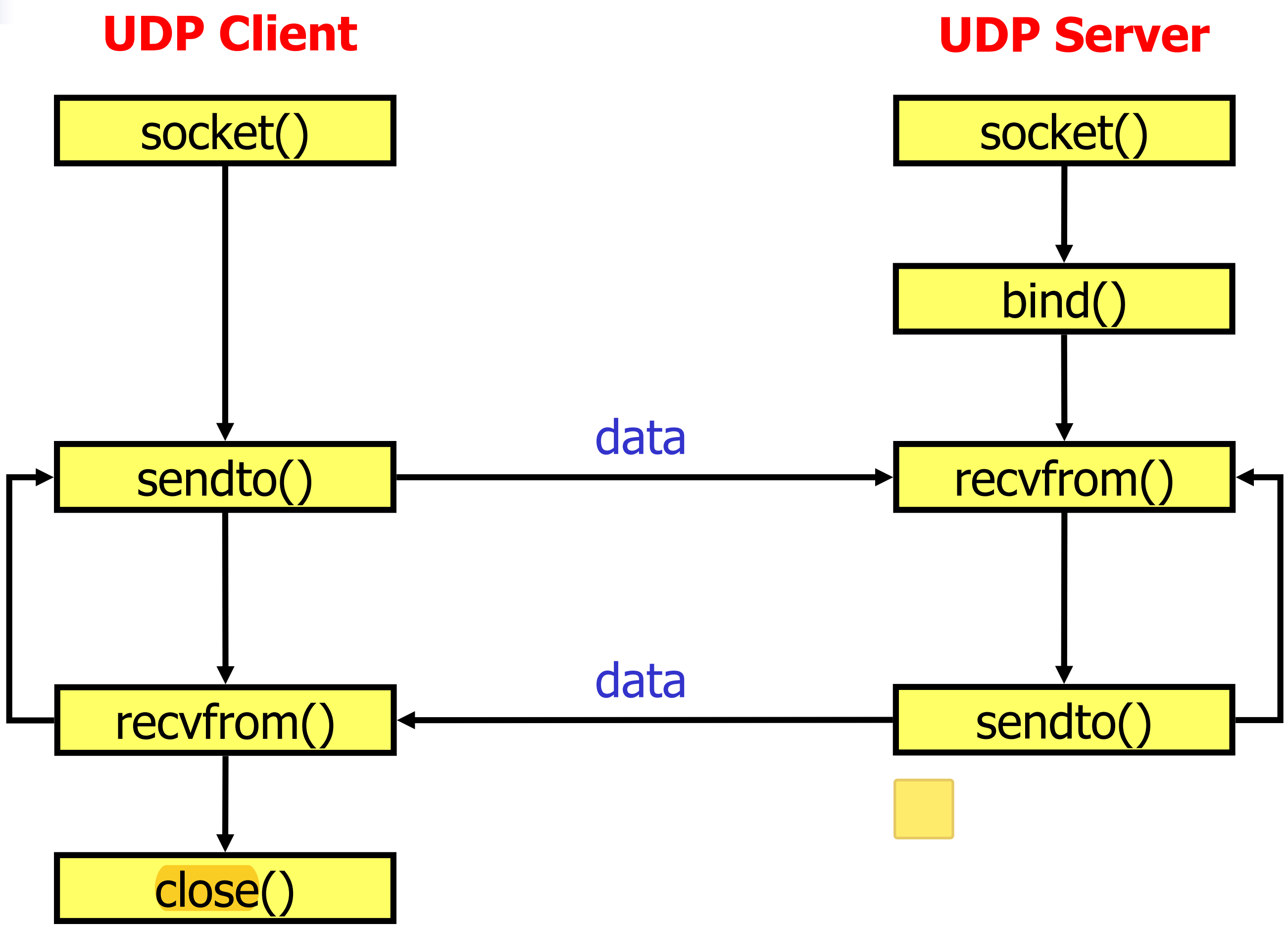
IP address structure in network byte order

Sample Programs

Overview of TCP-based sockets API



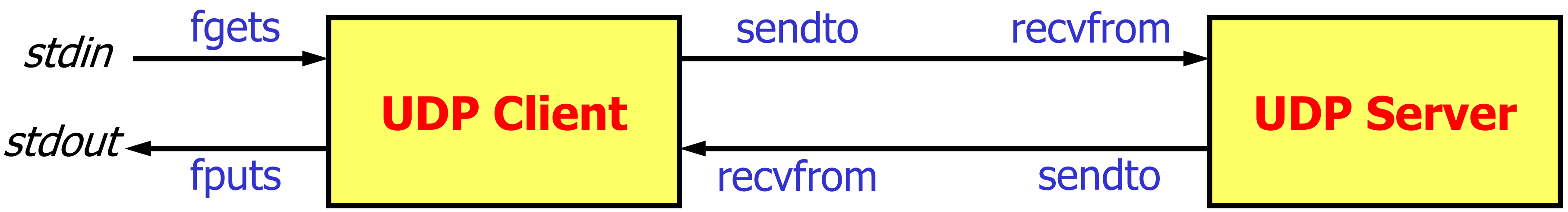
Overview of UDP-based sockets API



Sample programs

- UDP-based echo service
  - An echo service simply sends back to the originating source any data it receives
  - A very useful debugging and measurement tool
  - UDP Based Echo Service：be defined as a datagram based application on UDP. A server listens for UDP datagrams on UDP port 7. When a datagram is received, the data from it is sent back in an answering datagram.
- Sample programs
  - udpechoclt.c
  - udpechosvr.c

Basic flow of UDP-based echo service



Head part of UDP EchoClient

```
#include <stdio.h> /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), sendto() and
                           recvfrom() */
#include <arpa/inet.h> /* for sockaddr_in and inet_addr() */
#include <stdlib.h> /* for atoi() and exit() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for close() */
```

Initial part of UDP EchoClient

```
#define ECHOMAX 255 /* Longest string to echo */

int main(int argc, char *argv[])
{
   int sock; /* Socket descriptor */
   struct sockaddr_in echoServAddr; /* Echo server address */
   struct sockaddr_in fromAddr; /* Source address of echo */
   unsigned short echoServPort; /* Echo server port */
   unsigned int fromSize; /* In-out of address size
                             for recvfrom() */
   char *servIP; /* IP address of server */
   char *echoString; /* String to send to echo server */
   char echoBuffer[ECHOMAX+1]; /* Buffer for receiving
                                  echoed string */
   int echoStringLen; /* Length of string to echo */
   int respStringLen; /* Length of received response */
```

Argument check part of UDP EchoClient

```
if ((argc < 3) || (argc > 4)) /* Test for correct number of
 arguments */
{
    printf("Usage: %s <Server IP> <Echo Word> [<Echo Port>]\n",
                   argv[0]);
    exit(1);
 }

servIP = argv[1]; /* First arg: server IP address (dotted quad) */
echoString = argv[2]; /* Second arg: string to echo */
if ((echoStringLen = strlen(echoString)) > ECHOMAX) /* Check input
                                                       length */
    printf("Echo word too long.\n");
if (argc == 4)
    echoServPort = atoi(argv[3]); /* Use given port, if any */
else
    echoServPort = 7; /* 7 is the well-known port for echo service */
```

ASCII to integer

I/O part of UDP EchoClient

```
/* Create a datagram/UDP socket */
if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    printf("socket() failed.\n");
/* Construct the server address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr));/*Zero out structure*/
echoServAddr.sin_family = AF_INET; /* Internet addr family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);/*Server IP address*/
echoServAddr.sin_port = htons(echoServPort); /* Server port */
/* Send the string to the server */
if ((sendto(sock, echoString, echoStringLen, 0,
        (struct sockaddr *) &echoServAddr, sizeof(echoServAddr))
        != echoStringLen)
    printf("sendto() sent a different number of bytes than expected.\n");
/* Recv a response */
fromSize = sizeof(fromAddr);
if ((respStringLen = recvfrom(sock, echoBuffer, ECHOMAX, 0,
        (struct sockaddr *) &fromAddr, &fromSize)) != echoStringLen)
    printf("recvfrom() failed\n");
```

Generic socket address

Last part of UDP EchoClient

```
if (echoServAddr.sin_addr.s_addr != fromAddr.sin_addr.s_addr)
{
     printf("Error: received a packet from unknown source.\n");
     exit(1);
 }

/* null-terminate the received data */
echoBuffer[respStringLen] = '\0';
printf("Received: %s\n", echoBuffer);/*Print the echoed message*/
close(sock);
exit(0);
}
```

Head part of UDP EchoServer

```
#include <stdio.h> /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), sendto()
                                        and recvfrom() */
#include <arpa/inet.h> /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h> /* for atoi() and exit() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for close() */
```

Initial part of UDP EchoServer

```
#define ECHOMAX 255 /* Longest string to echo */

int main(int argc, char *argv[])
{
    int sock; /* Socket */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned int cliAddrLen; /* Length of client address */
    char echoBuffer[ECHOMAX]; /* Buffer for echo string */
    unsigned short echoServPort; /* Server port */
    int recvMsgSize; /* Size of received message */
```

Argument check part of UDP EchoServer

```
if (argc != 2)
{
    printf("Usage: %s <UDP SERVER PORT>\n", argv[0]);
    exit(1);
 }
```

Socket part of UDP EchoServer

```
echoServPort = atoi(argv[1]); /* First arg: local port */

/* Create socket for sending/receiving datagrams */
if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
    printf("socket() failed.\n");
/* Construct local address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr));
echoServAddr.sin_family = AF_INET;
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);
echoServAddr.sin_port =htons(echoServPort);
/* Bind to the local address */
if ((bind(sock, (struct sockaddr *) &echoServAddr,
       sizeof(echoServAddr))) < 0)
    printf("bind() failed.\n");
```

Main loop of UDP EchoServer

```
for (;;) /* Run forever */
{
    /* Set the size of the in-out parameter */
    cliAddrLen = sizeof(echoClntAddr);
    /* Block until receive message from a client */
    if ((recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX,
         0,(struct sockaddr *) &echoClntAddr, &cliAddrLen)) < 0)
        printf("recvfrom() failed.\n");
    printf("Handling client %s\n",inet_ntoa(echoClntAddr.sin_addr));
    /* Send received datagram back to the client */
    if ((sendto(sock, echoBuffer, recvMsgSize, 0,
         (struct sockaddr *) &echoClntAddr,
          sizeof(echoClntAddr))) != recvMsgSize)
        printf("sendto() sent a different number of bytes
                than expected.\n");
 }

}
```

Run the Sample Programs (1)

- Give correct arguments

Server process window

```
[shiyan@localhost 20071022]$ ./udpechosvr
Usage: ./udpechosvr <UDP SERVER PORT>
```

Client process window

```
[shiyan@localhost 20071022]$ ./udpechoclt
Usage: ./udpechoclt <Server IP> <Echo Word> [<Echo Port>]
```

Run the Sample Programs (2)

- Use correct username

Server process window

```
[shiyan@localhost 20071022]$ ./udpechosvr 7
bind() failed.
```

Note: binding the port number less than 1024 requires root authority

Client process window

```
[shiyan@localhost 20071022]$ ./udpechoclt 192.168.1.253 hello
```

Run the Sample Programs (3)

- Successful running using root

Server process window

```
[root@localhost 20071022]# ./udpechosvr 7
Handling client 192.168.1.253
```

Client process window

```
[root@localhost 20071022]# ./udpechoclt 192.168.1.253 hello
Received: hello
[root@localhost 20071022]#
```

Run the Sample Programs (4)

- Successful running using other username

Server process window

```
[shiyan@localhost 20071022]$ ./udpechosvr 1500
Handling client 192.168.1.253
```

Client process window

```
[shiyan@localhost 20071022]$ ./udpechoclt 192.168.1.253 hello 1500
Received: hello
[shiyan@localhost 20071022]$
```