Network Programming IA 1

Question Bank

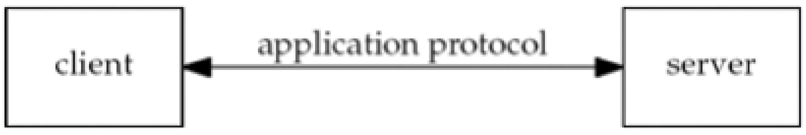
Q1 What is Network Programming & Decisions to be made

Definition

Network programming involves writing programs to communicate with processes either on the same or on other machines on the network using standard protocols.

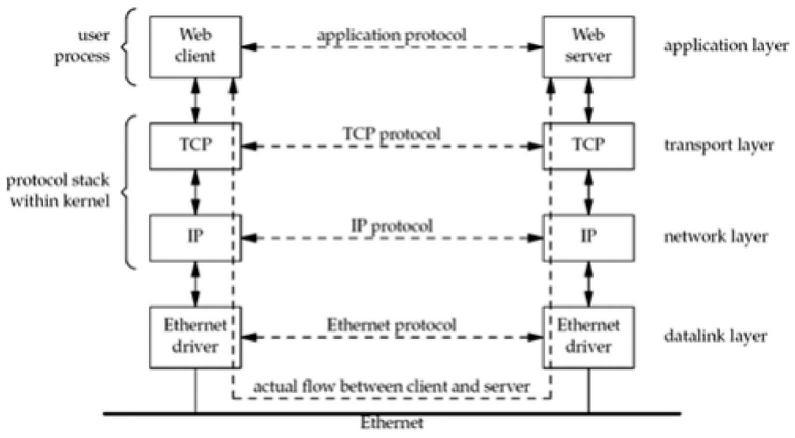
Decisions to be made

Figure 1.1. Network application: client and server.



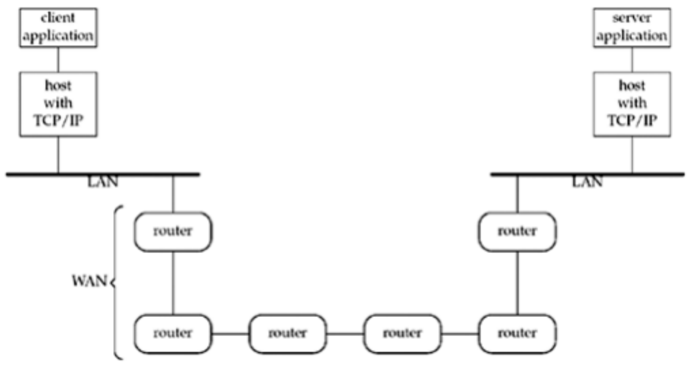
- A high-level decision must be made as to which program would initiate the communication first & when responses are expected.
- For example, A web server is typically thought of as a long-running program that sends network messages only in response to requests coming in from the network.
- The other side of the protocol is a web client, such as a browser which always initiates communication with the server.
- This organization into client and server is used by most network-aware applications. Deciding that the client always initiates requests tends to simplify the protocol as well as the programs themselves.

✓ Q2 Communication over LAN



- Even though the client and server communicate using an application protocol, the transport layers communicate using TCP.
- The actual flow of information between client and server goes down the protocol stack on one side, across the network, and up the protocol stack on the other side.
- Client & Server are typically user processes, while TCP and IP protocols are normally part of the protocol stack within the kernel.
- The four layers labeled in the diagram are the Application layer, Transport layer, Network layer, Data-link layer.
- Some clients and servers use the User Datagram Protocol (UDP) instead of TCP.

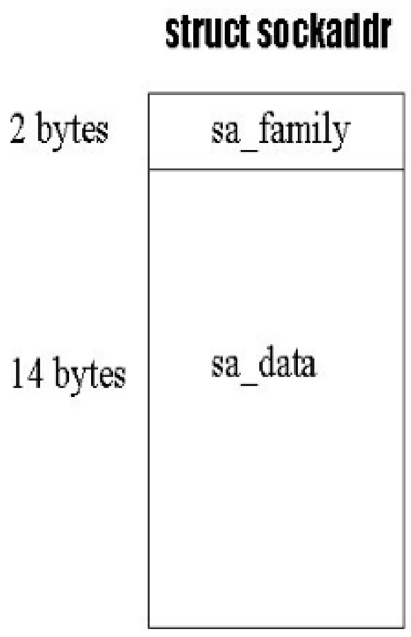
✓ Q3 Communication over WAN



- The client & server on different LANs is connected to a Wide Area Network (WAN) via a router.
- Routers are the building blocks of WANs. The largest WAN today is the Internet.
- Many companies build their own WANs and these private WANs may or may not be connected to the Internet.

Q4 sockaddr & sockaddr_in

sockaddr

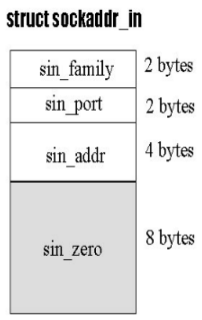


- struct sockaddr is a general structure valid for any protocol. It is the generic socket address type.
- `sa_family` — 16-bit integer value identifying the protocol family being used. Eg: TCP/IP → AF_INET.
- `sa_data` — Address information used in protocol family. Eg: TCP/IP → IP address & port no.
- It is used as the base of a set of address structures that acts like a discrimination union sockaddr holding the socket information.

Structure:

```
struct sockaddr {
    unsigned short sa_family;  // Address family( Eg. AF_INET)
    char sa_data[14]; // Family-specific address info
};
```

sockaddr_in



- struct sockaddr_in is protocol specific, to be specific for IPv4 address family.
- `sin_addr` — 32-bit in_addr structure.
- It specifies a transport address & port for the AF_INET address family.

Structure:

```
struct sockaddr_in {
    short           sin_family; // 2 bytes e.g. AF_INET, AF_INET6
    unsigned short  sin_port; // 2 bytes e.g. htons(3490)
    struct in_addr  sin_addr; // 4 bytes see struct in_addr, below
    char            sin_zero[8];// 8 bytes zero this if you want to
};
```

Q5 Wrapper Function

In any real-world program, it is essential to check every function call for an error return. We check for errors from the socket, inet_pton, connect, read, and fputs, and when one occurs, we call our own functions, err_quit, and err_sys, to print an error message and terminate the program.

Since terminating on an error is the common case, we can shorten our programs by defining a wrapper function that performs the actual function call, tests the return value, and terminates on an error.

`bind` Function

The **bind** function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
/* Returns: 0 if OK,-1 on error */
```

`listen` Function

The **listen** function is called only by a TCP server and it performs two actions:

- Convert an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
- The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

```
#include <sys/socket.h>
int listen (int sockfd, int backlog);
/* Returns: 0 if OK,-1 on error */
```

`accept` Function

The **accept** function is called by a TCP server to return the next completed connection from the front of the completed connection queue. If the completed connection queue is empty, the process is put to sleep.

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
/* Returns: 0 if OK,-1 on error */
```

`connect` Function

The **connect** function is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
/* Returns: 0 if OK,-1 on error */
```

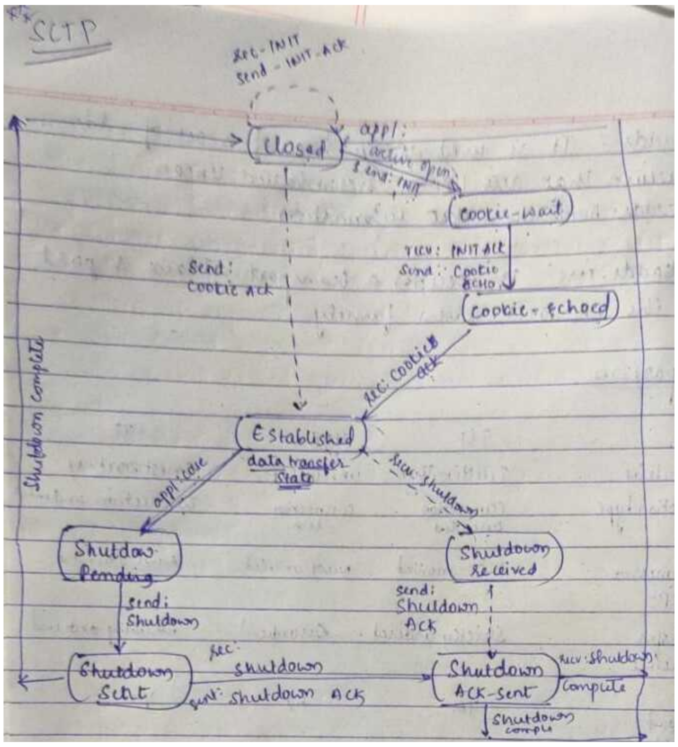
`socket` Function

To perform network I/O, the first thing a process must do is call the **socket** function, specifying the type of communication protocol desired.

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
/* Returns: 0 if OK,-1 on error */
```

Q6 SCTP

Stream Control Transmission Protocol (SCTP) is a transport-layer protocol that ensures reliable, in-sequence transport of data. SCTP exists at an equivalent level with User Datagram Protocol (UDP) and Transmission Control Protocol (TCP), which provides transport layer functions to many Internet applications.



- The transitions from one state to another in the state machine are dictated by the rules of SCTP, based on the current state and the chunk received in that state.
- For example, if an application performs an active open in the CLOSED state, SCTP sends an INIT and the new state is COOKIE-WAIT.
- If SCTP next receives an INIT ACK, it sends a COOKIE ECHO and the new state is COOKIE-ECHOED.
- If SCTP then receives a COOKIE ACK, it moves to the ESTABLISHED state. This final state is where most data transfer occurs.
- The two arrows leading from the ESTABLISHED state deal with the termination of an association.
- If an application calls close before receiving a SHUTDOWN, the transition shifts to the SHUTDOWN-PENDING state.

- However, if an application receives a SHUTDOWN while in the ESTABLISHED state, the transition is to the SHUTDOWN-RECEIVED state.

Q7 TCP and UDP

TCP is a connection-oriented protocol. Connection-orientation means that the communicating devices should establish a connection before transmitting data and should close the connection after transmitting the data.

Uses of TCP

- World Wide Web(HTTP)
- E-mail (SMTP TCP)
- File Transfer Protocol (FTP)
- Secure Shell (SSH)

UDP is a datagram-oriented protocol. This is because there is no overhead for opening a connection, maintaining a connection, and terminating a connection. UDP is efficient for broadcast and multicast types of network transmission.

Uses of UDP

- Domain Name Server (DNS)
- Media streaming
- Online multiplayer games
- Voice over IP (VoIP)
- Tunneling/VPN

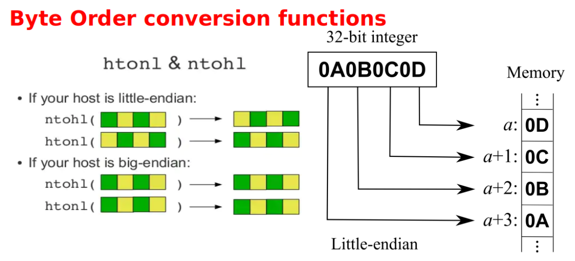
Q8 Difference between IPv4 and IPv6

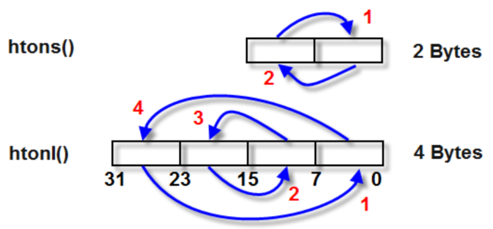
IPv4	IPv6
IPv4 has a 32-bit address length	IPv6 has a 128-bit address length
Address representation of IPv4 is in decimal	Address Representation of IPv6 is in hexadecimal
The security feature is dependent on the application	IPSEC is an inbuilt security feature in the IPv6 protocol
End to end, connection integrity is unachievable	End to end, connection integrity is achievable
In IPv4 checksum field is available	In IPv6 checksum field is not available

Q9 Comparison between TCP, UDP & SCTP

	TCP	UDP	SCTP
Reliability	Trustworthy	Unreliable	Trustworthy
Connection Type	Connection-oriented	Connection-less	Connection-oriented
Data Type	Byte-stream grouped into chunks	Datagram	Byte-stream grouped into chunks
Transfer Sequence	Strictly ordered	Disordered	Partially ordered
Overload Control	Yes	No	Yes
Error Tolerance	No	No	Yes
Transmission	Byte-oriented	Message-oriented	Message-oriented
Security	Yes	Yes	Improved

Q10 Byte Order Conversion Function

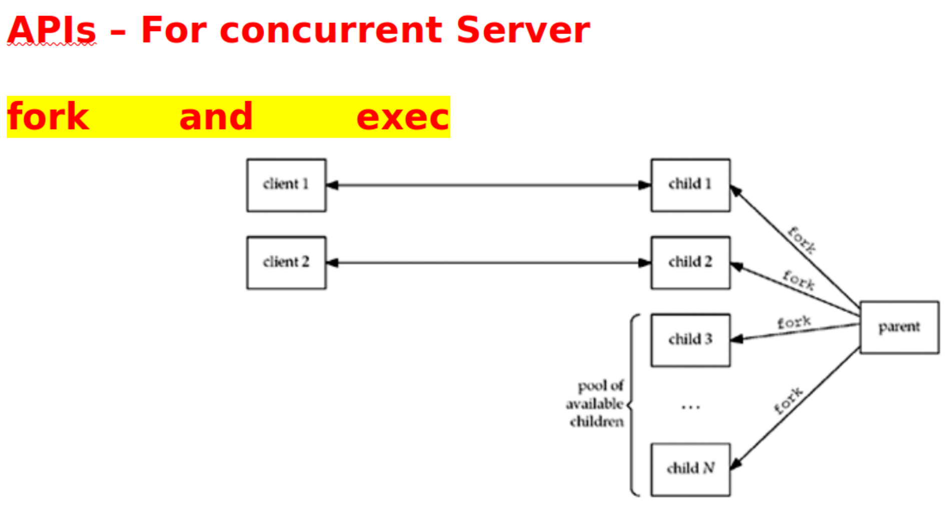




Functions are

- `htons():` **host to network short —** This function converts 16-bit quantities from host byte order to network byte order.
- `ntonl():` **host to network long —** This function converts 32-bit quantities from host byte order to network byte order.
- `ntohs():` **network to host short —** This function converts 16-bit quantities from network byte order to host byte order.
- `ntohl():` **network to host long —** This function converts 32-bit quantities from network byte order to host byte order.

Q11 APIs — For concurrent server (`fork` & `exec`)


APIs – For concurrent Server

fork and exec

```
#include <unistd.h>
pid_t fork(void);
/* Returns: 0 in child, process ID of child in parent, -1 on error */
```

- A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is typical for network servers.
- A process wants to execute another program parallelly. Since the only way to create a new process is by calling **fork**, the process first calls **fork** to make a copy of itself, and then one of the copies calls **exec** to replace itself with the new program. This is typical for programs such as shells.
- **exec** replaces the current process image with the new program file, and this new program normally starts at the main function. The process ID does not change.

```
// fork()
int main()
{
  // make two process which run same
  // program after this instruction
  fork();
  printf("Hello world!\n");
  return 0;
}

/* For exec() */
// EXEC.c
int main()
{
  int i;
  printf("I am EXEC.c called by execvp() ");
  return 0;
}

// execDEMO.c
int main()
{
    //A null terminated array of character
    //pointers
    char *args[]={"./EXEC",NULL};
    execv(args[0],args);

    /*All statements are ignored after execvp() call as this whole
    process(execDemo.c) is replaced by another process (EXEC.c)
    */
    printf("Ending-----");

  return 0;
}
```