DATE
PAGE

Unit-2

Byte Ordering :- Byte order refers to the order of digits in computer words at least 16 bits long. It tells us how bytes are arranged when sending data over a network. It is an important concept in networking.

A byte (8 bits) has a limited range of 256 values. When a value is beyond this range, it has to be stored in multiple bytes.

For Ex :- A 16 bit Number has two bytes, one is called lower byte and the other is the higher order byte.

Suppose we have two memory address An and An+1.

Which of the high-order and low order bytes goes in An?

There are two ways to store this value :-

1) **Little Endian** :- In this scheme,

low-order byte is stored on the starting address (An) and high order byte is stored on the next address (A_{n+1}) Ex- Intel 'x 86 family

2) Big Endian :- In this scheme, high order byte is stored on the starting address (An) and low order byte is stored on the next address (A_{n+1}). It is known as "Network Byte Order". the TCP/IP Internet protocol also uses big endian. Ex- Motorola 68K

word address				
0	3	2	1	0
4	7	6	5	4
8	11	10	9	8

⟵ 4B ⟶

Little Endian

Words				
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11

⟵ 4B ⟶

Big Endian

Example :- Suppose we have two word address :-

1) 1A C8 B2 46
2) FD 8C 1E DF

i) Big Endian :-

0	1A	C8	B2	46
4	FD	8C	1E	DF

Suppose we have to calculate the address of word 1 then,

1A	C8	B2	46
00011010	11001000	10110010	0100 0110

= 449,360,454

ii) Little Endian :-

0	46	B2	C8	1A
4	DF	1E	8C	FD

46	B2	C8	1A
01000110	10110010	11001000	00011010

= 1,186,121,754

Now you can observe these two num are different. So the Numbers var in the use of different Architectu

⇒ Some processors support both orde and are called Bi-Endian

Ex- PowerPC and Itanium

DATE
PAGE

⟹ Since computers using different byte ordering and exchanging data, have to operate correctly on data, the convention is to always send data on the network in big-endian format. We call this Network Byte Ordering and the ordering used on a computer is called Host-byte Ordering.

Byte Ordering Conversion functions :-

A host system may be little-endian but when sending data into the network, it must convert data into big-endian format. Likewise, a little-endian machine must first convert network data into little-endian before processing it. Four common functions to do these conversions are (for 32-bit long and 16-bit short) as follows:-

Function	Description
htons()	Host to Network Short
htonl()	Host to Network long
ntohl()	Network to Host Long
ntohs()	Network to Host Short

DATE
PAGE

1) htons() :- This function convert 16 bits (2 byte) quanti from host byte order to network byte order.

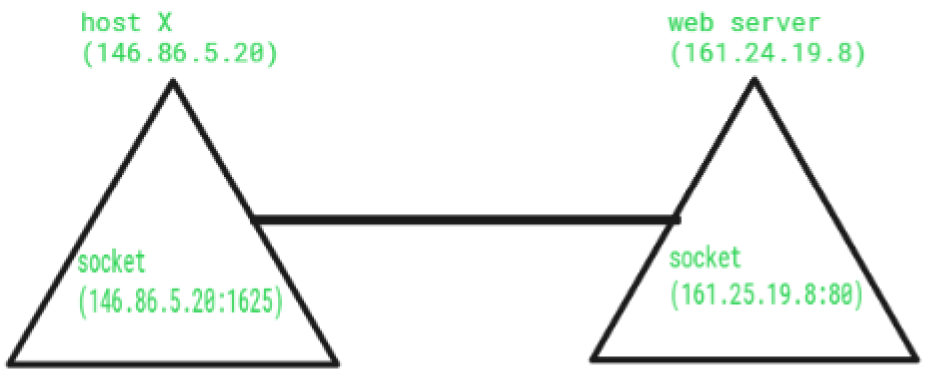
2) htonl() :- This function converts 32 bits (4 byte) quantit from host byte order to networ byte order.

3) ntohs() :- This function conver 16-bit (2-byte) quant from network byte order to host byt order.

4) ntohl() :- This function converts 32- quantities from network byt order to host byte order.

UNIT-2

Socket in Computer Network: A socket is one endpoint of a two way communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication takes place**.**
Sockets are created using **'socket'** system call. Each socket has a specific address. This address is composed of an IP address and a port number.
Sockets are generally employed in client server applications. The server creates a socket, attaches it to a network port addresses then waits for the client to contact it. The client creates a socket and then attempts to connect to the server socket. When the connection is established, transfer of data takes place.



Types of Sockets:

There are two types of Sockets: the **datagram** socket and the **stream** socket.

1. **Datagram Socket :** This is a type of network which has connection less point for sending and receiving packets
2. **Stream Socket:** which provides a connection-oriented, sequenced, and unique flow of data without record boundaries with well-defined mechanisms for creating and destroying connections and for detecting errors? It is similar to phone. A connection is established between the phones (two ends) and a conversation (transfer of data) takes place.

TCP Socket API: The sequence of function calls for the client and a server participating in a TCP connection is as follows:

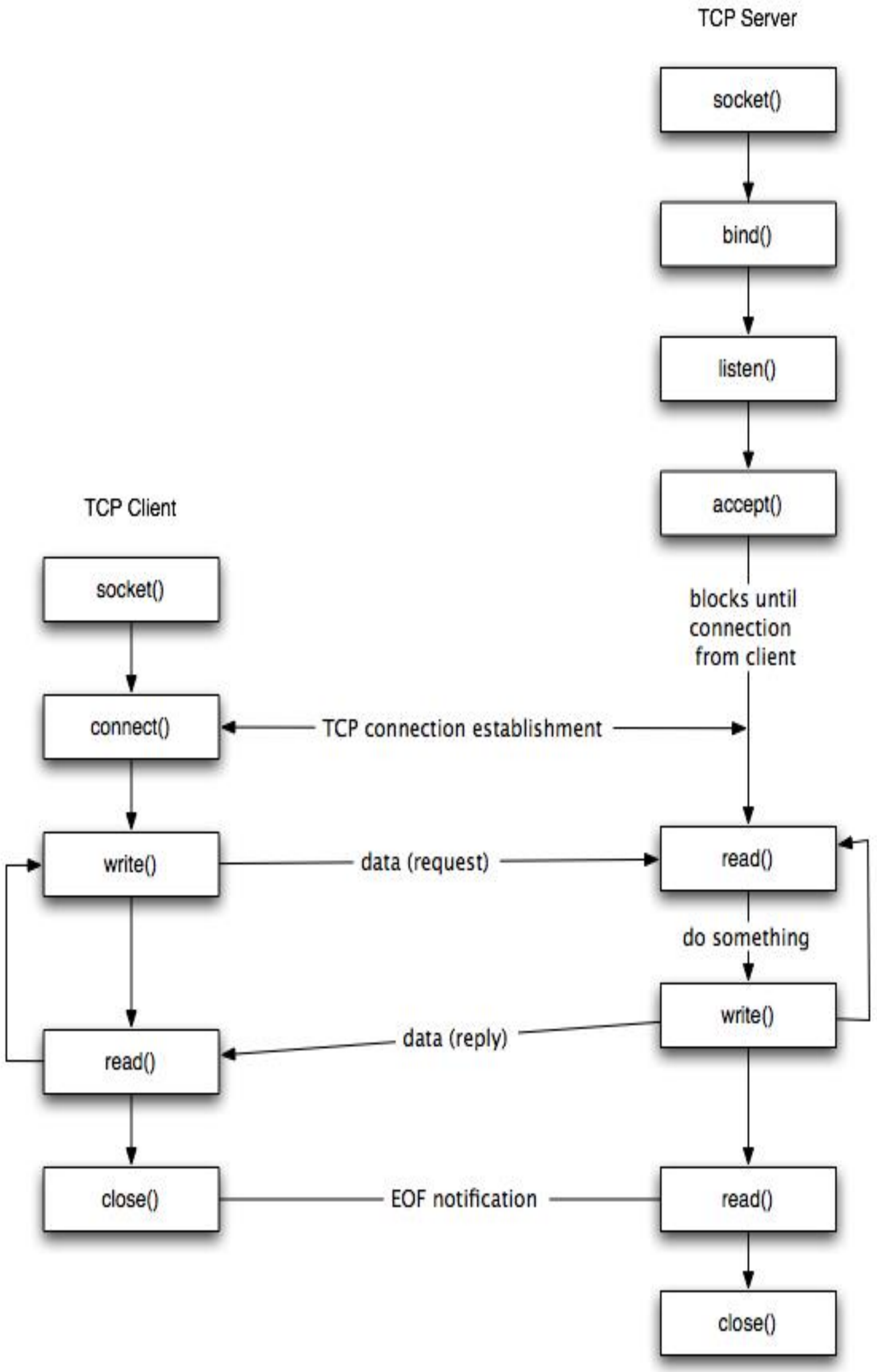


Figure 3: TCP client-server.

The steps for establishing a TCP socket on the client side are the following:

- Create a socket using the socket() function;
- Connect the socket to the address of the server using the connect() function;
- Send and receive data by means of the read () and write () functions.

The steps involved in establishing a TCP socket on the server side are as follows:

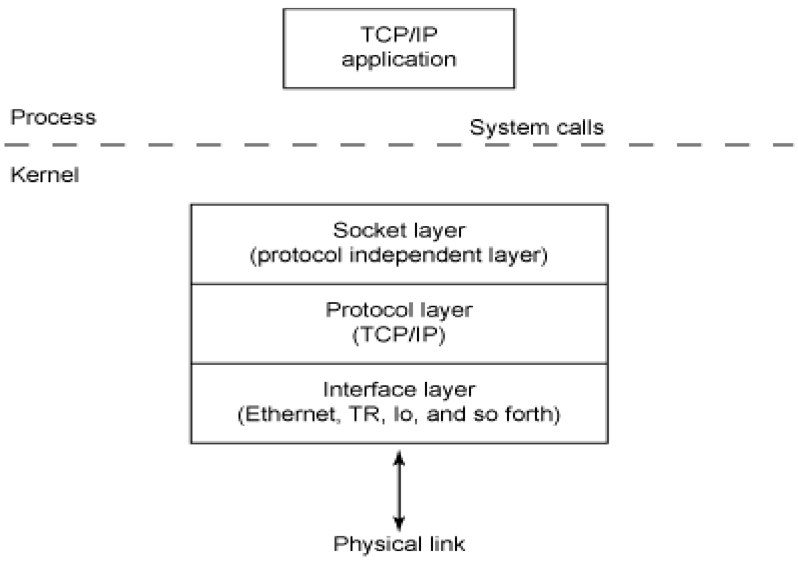
- Create a socket with the socket() function;
- Bind the socket to an address using the bind() function;
- Listen for connections with the listen() function;
- Accept a connection with the accept () function system call. This call typically blocks until a client connects with the server.
- Send and receive data by means of send () and receive ().

System Calls: A typical TCP client and server application issues a sequence of TCP system calls to attain certain functions. Some of these system calls include socket (), bind (), listen (), accept (), send (), and receive ().

Normal sequence of calls made by the TCP application

TCP Client Application	**TCP Server Application**
Socket	Socket
Bind	Bind
..	Listen
Connect	Accept
Send	Receive
Receive	Send

Layers of the TCP system call


TCP/IP application
Process
System calls
Kernel
Socket layer (protocol independent layer)
Protocol layer (TCP/IP)
Interface layer (Ethernet, TR, lo, and so forth)
Physical link

1. **Socket Layer**: Any TCP system call that is made is received by the socket layer. The socket layer validates the correctness of the parameters passed by the TCP application. This is a *protocol-independent* layer because the protocol is not yet hooked onto the call.
2. **Protocol Layer:** Below the socket layer is the protocol layer, which contains the actual implementation of the protocol (in this case, TCP). When the socket layer makes calls into the protocol layer, it ensures that it has exclusive access for the data structures that are shared between the two layers. This is done to avoid any data structure corruption.
3. **Interface Layer:** The various network device drivers run at the interface layer, which receives and transmits data from and to the physical link.

Each socket has a socket queue, and each interface has an interface queue used for data communication. However, there is only one protocol queue, which is called the IP input queue, for the entire protocol layer. The interface layer inputs data to the protocol layer through this IP input queue. The protocol layer outputs data to the interface using the respective interface queues.

Function Call	Description
Socket()	To create a socket
Bind()	It’s a socket identification like a telephone number to contact
Listen()	Ready to receive a connection
Connect()	Ready to act as a sender
Accept()	Confirmation, it is like accepting to receive a call from a sender
Write()	To send data
Read()	To receive data
Close()	To close a connection

The socket () Function

The first step is to call the socket function, specifying the type of communication protocol (TCP based on IPv4, TCP based on IPv6, UDP).

The function is defined as follows:

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);
```

where family specifies the protocol family, type is a constant described the type of socket (SOCK_STREAM for stream sockets and SOCK_DGRAM for datagram sockets.

The function returns a non-negative integer number, similar to a file descriptor, that we define *socket descriptor* or -1 on error.

The connect() Function

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

The function is defined as follows:

```
#include <sys/socket.h>

int connect (int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

where sockfd is the socket descriptor returned by the socket function.

The function returns 0 if it succeeds in establishing a connection (i.e., successful TCP three-way handshake, -1 otherwise.

The client does not have to call bind() in Section before calling this function: the kernel will choose both an ephemeral port and the source IP if necessary.

The bind() Function

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

The function is defined as follows:

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

where sockfd is the socket descriptor, myaddr is a pointer to a protocol-specific address and addrlen is the size of the address structure.

bind() returns 0 if it succeeds, -1 on error.

The listen() Function

The listen() function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. It is defined as follows:

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

where sockfd is the socket descriptor and backlog is the maximum number of connections the kernel should queue for this socket. The backlog argument provides an hint to the system of the number of outstanding connect requests that is should enqueue in behalf of the process. Once the queue is full, the system will reject additional connection requests. The backlog value must be chosen based on the expected load of the server.

The function listen() return 0 if it succeeds, -1 on error.

The accept() Function

The accept() is used to retrieve a connect request and convert that into a request. It is defined as follows:

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr,
socklen_t *addrlen);
```

where sockfd is a new file descriptor that is connected to the client that called the connect(). The cliaddr and addrlen arguments are used to return the protocol address of the client. The new socket descriptor has the same socket type and address family of the original socket. The original socket passed to accept() is not associated with the connection, but instead remains available to receive additional connect requests.

If we don’t care about the client’s identity, we can set the cliaddr and addrlen to NULL. Otherwise, before calling the accept function, the cliaddr parameter has to be set to a buffer large enough to hold the address and set the interger pointed by addrlen to the size of the buffer.

The send() Function

Since a socket endpoint is represented as a file descriptor, we can use read and write to communicate with a socket as long as it is connected. However, if we want to specify options we need another set of functions.

For example, send() is similar to write() but allows to specify some options. send() is defined as follows:

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

where buf and nbytes have the same meaning as they have with write. The additional argument flags is used to specify how we want the data to be transmitted. We will not consider the possible options. We will assume it equal to 0.

The function returns the number of bytes if it succeeds, -1 on error.

The receive() Function

The recv() function is similar to read(), but allows to specify some options to control how the data are received. We will not consider the possible options. We will assume it is equal to 0.

receive is defined as follows:

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

The function returns the length of the message in bytes, 0 if no messages are available and peer had done an orderly shutdown, or -1 on error.

The close() Function

The normal close() function is used to close a socket and terminate a TCP socket. It returns 0 if it succeeds, -1 on error. It is defined as follows:

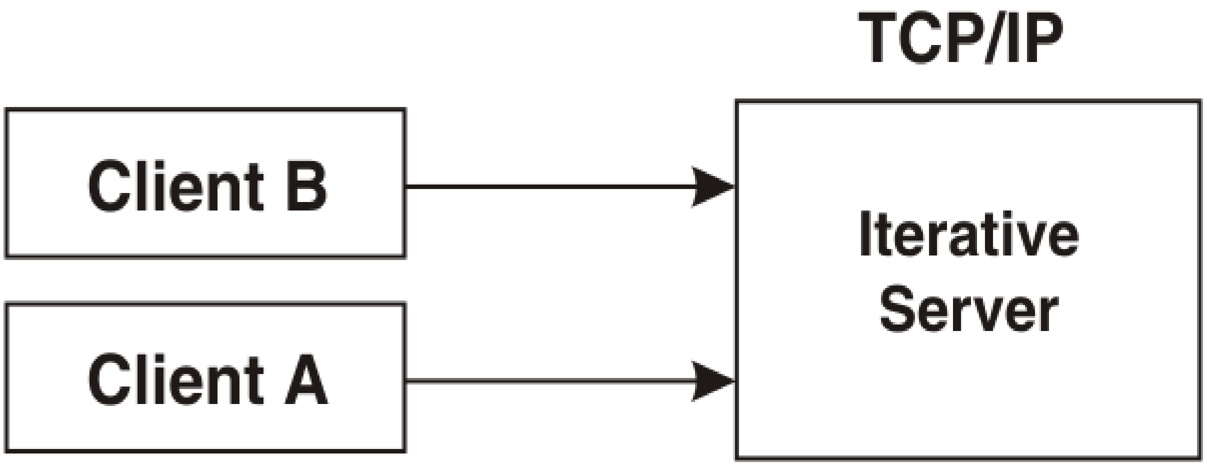
```
#include <unistd.h>

int close(int sockfd);
```

Concurrent and iterative servers: There are two main classes of servers, iterative and concurrent.

An iterative server iterates through each client, handling it one at a time. It handles both the connection request and the transaction involved in the call itself. Iterative servers are fairly simple and are suitable for transactions that do not last long.

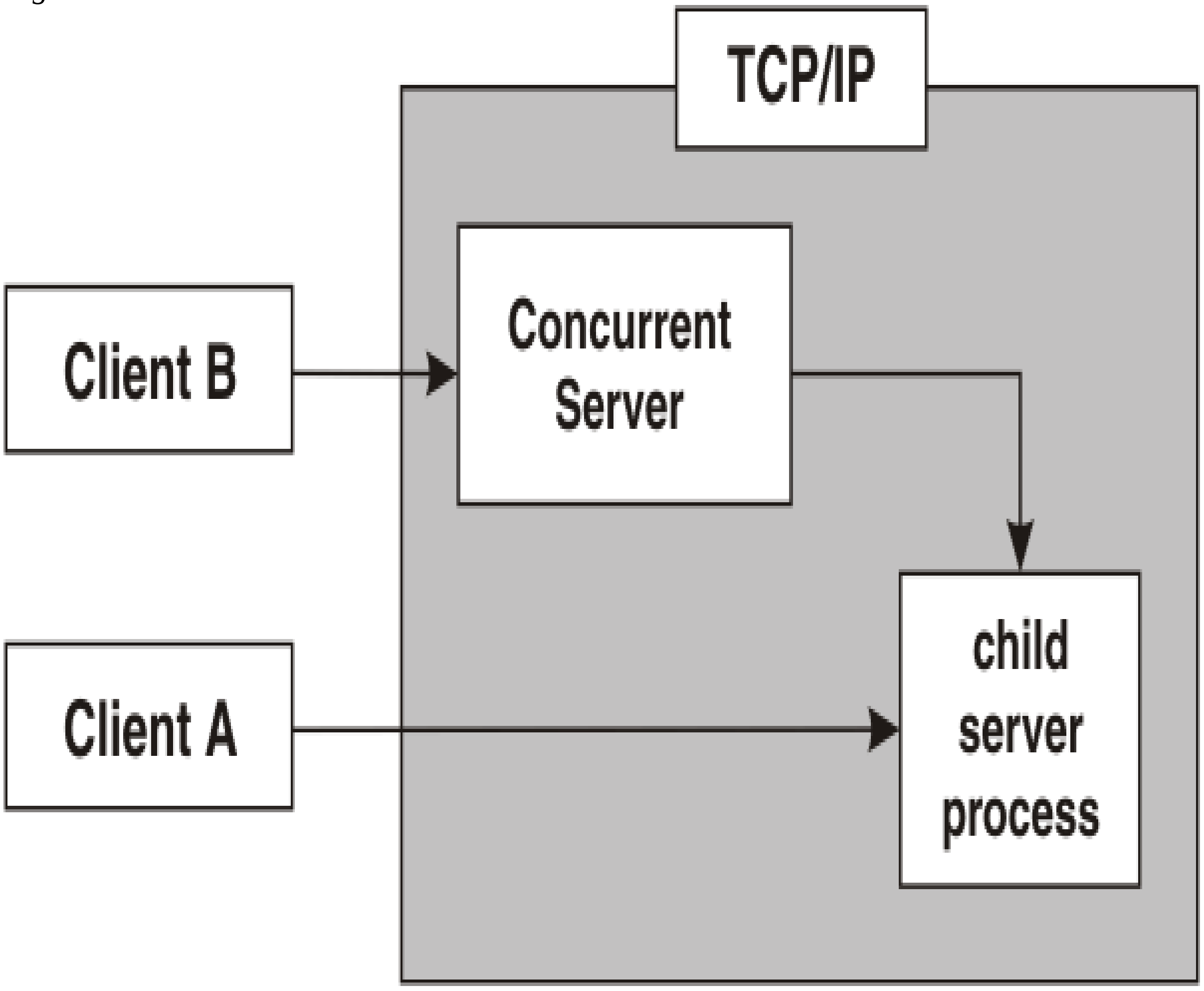
However, if the transaction takes more time, queues can build up quickly. In Figure once Client A starts a transaction with the server, Client B cannot make a call until A has finished.

Figure 1. An iterative server



A *concurrent* server handles multiple clients at the same time. As shown in the figure, Client A has already established a connection with the server, which has then created a *child server process* to handle the transaction. This allows the server to process Client B's request without waiting for A's transaction to complete. More than one child server can be started in this way.

Figure 2. A concurrent server



The simplest technique for a concurrent server is to call the fork function, creating one child process for each client. An alternative technique is to use *threads* instead (i.e., light-weight processes).

```
#include <unist.h>

pid_t fork(void);
```

The function returns 0 if in child and the process ID of the child in parent; otherwise, -1 on error.

In fact, the function fork() is called once but returns *twice*. It returns once in the calling process (called the parent) with the process ID of the newly created process (its child). It also returns in the child, with a return value of 0. The return value tells whether the current process is the parent or the child.

SOCKET Programming Interface

There are three types of socket programming interface

Stream sockets: Stream socket is the most common type of socket programming interface. The communicating parties first establish a socket connection between them, so that any data passed through the connection will arrive in the order in which it was sent by the sender because of the connection-oriented service.

Datagram sockets: Provides connection-less services. No connection is established before data transmission. Communicating party transmits the datagrams as required or waits for the response. Data can be lost during the transmission or may arrive out of order. Implementing a datagram provides more flexibility in comparison to using stream sockets.

Raw sockets: This socket interface Bypasses the library’s built-in support for standard protocols such as UDP(User Datagram Protocol) and TCP(Transmission Control Protocol). Raw sockets are socket programming interfaces which are used for custom low-level protocol development.

Remote Procedure Call

A common pattern of communication used by application programs structured as a *client/server* pair is the request/reply message transaction: A client sends a request message to a server, and the server responds with a reply message, with the client blocking (suspending execution) to wait for the reply. Figure 137 illustrates the basic interaction between the client and server in such an exchange.

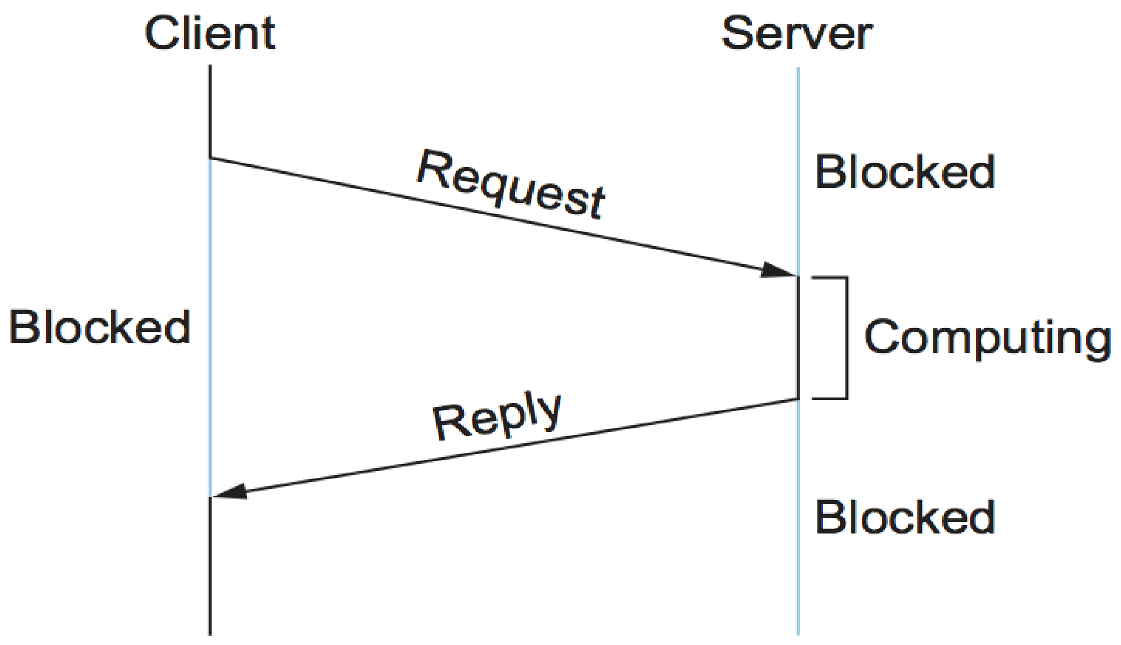


Figure. *Timeline for RPC*

A transport protocol that supports the request/reply paradigm is much more than a UDP message going in one direction followed by a UDP message going in the other direction. It needs to deal with correctly identifying processes on remote hosts and correlating requests with responses. It may also need to overcome some or all of the limitations of the underlying network outlined in the problem statement at the beginning of this chapter. While TCP overcomes these limitations by providing a reliable byte-stream service, it doesn't perfectly match the request/reply paradigm either.

Now a third category of transport protocol, called *Remote Procedure Call* (RPC) describes that more closely matches the needs of an application involved in a request/reply message exchange.

Remote procedure call is an interprocess communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

How does RPC work?

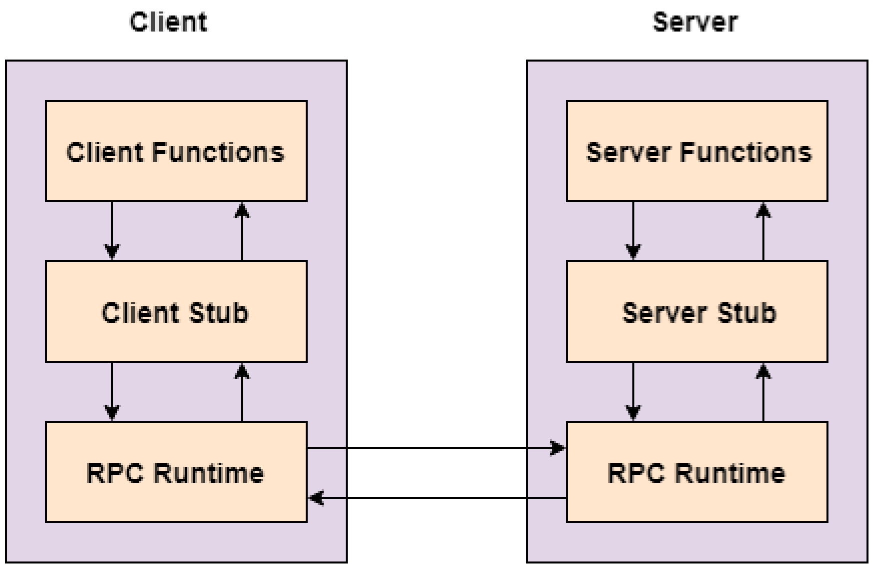
When a remote procedure call is invoked, the calling environment is suspended, the procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is then executed in that environment.

When the procedure finishes, the results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

The sequence of events in a remote procedure call is given as follows −

- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message.
- The message is sent from the client to the server by the client’s operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.

A diagram that demonstrates this is as follows −



Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows −

- Remote procedure calls support process oriented and thread oriented models.
- The internal message passing mechanism of RPC is hidden from the user.
- The effort to re-write and re-develop the code is minimum in remote procedure calls.
- Remote procedure calls can be used in distributed environment as well as the local environment.
- Many of the protocol layers are omitted by RPC to improve performance.

Disadvantages of Remote Procedure Call

Some of the disadvantages of RPC are as follows −

- The remote procedure call is a concept that can be implemented in different ways. It is not a standard.
- There is no flexibility in RPC for hardware architecture. It is only interaction based.

SCTP (Stream Control Transmission Protocol):

- Stream Control Transmission Protocol (SCTP) is a reliable, message-oriented transport layer protocol in computer networks which provides a full-duplex association i.e., transmitting multiple streams of data between two end points at the same time that have established a connection in network.
- SCTP has mixed features of TCP and UDP.
- SCTP is a standard protocol (RFC 2960) and is developed by Internet Engineering Task Force (IETF).
- SCTP maintains the message boundaries and detects the lost data, duplicate data as well as out-of-order data.
- SCTP provides the Congestion control as well as Flow control.
- SCTP is especially designed for internet applications.
- SCTP makes it easier to support telephonic conversation on Internet. A telephonic conversation requires transmitting of voice along with other data at the same time on both ends.
- SCTP is also intended to make it easier to establish connection over wireless network and managing transmission of multimedia data.

SCTP Services: Some important services provided by SCTP are as stated below:

1. Process-to- Process communication: SCTP uses all important ports of TCP.

2. Multi- Stream Facility: SCTP provides multi-stream service to each connection, called as association. If one stream gets blocked, then the other stream can deliver the data.

3. Full- Duplex Communication: SCTP provides full-duplex service (the data can flow in both directions at the same time).

4. Connection- Oriented Service: The SCTP is a connection oriented protocol, just like TCP with the only difference that, it is called association in SCTP. If User1 wants to send and receive message from user2, the steps are:

Step1: The two SCTPs establish the connection with each other.
Step2: Once the connection is established, the data gets exchanged in both the directions.
Step3: Finally, the association is terminated.

5. Reliability: SCTP uses an acknowledgement mechanism to check the arrival of data.

Features of SCTP: Some important features of SCTP are as stated below:

1. Transmission Sequence Number (TSN): The unit of data in SCTP is a data chunk. Data transfer in SCTP is controlled by numbering the data chunks. In SCTP, TSN is used to assign the numbers to different data chunks.

2. Stream Identifier (SI): The SI is a 16 bit number and starts with 0. In SI, there are several streams in each association and it is needed to identify them. Each data chunk needs to carry the SI in the header, so that it is properly placed in its stream on arrival.

3. Packets: In SCTP, the data is carried out in the form of data chunks and control information is

carried as control chunks. Data chunks and control chunks are packed together in the packet.

4. Multihoming: Multihoming allows both ends (sender and receiver) to define multiple IP addresses for communication. But, only one of these can be defined as primary address and the remaining can be used as alternative addresses.
5. It is suitable for Ethernet jumbo frames because of improved error detection.
6. It provides validation and acknowledgement mechanisms which protect against flooding attacks.
7. It provides notification of duplicated or missing data chunks.
8. It eliminates unnecessary head-of-line blocking by delivering chunks within independent data.
9. It provides path selection and monitors it.
10. It selects a primary data transmission path and tests its connectivity.

Advantages of SCTP:

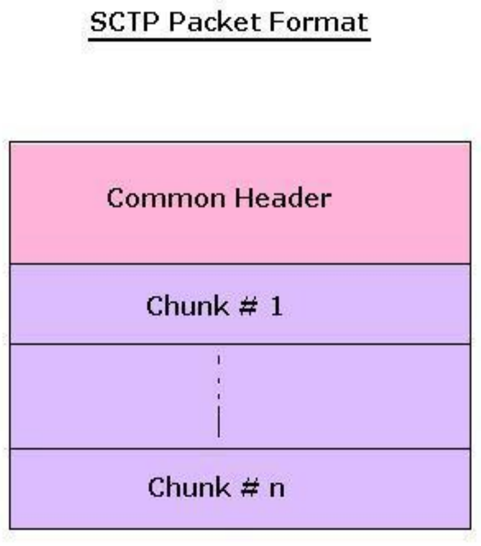
1. It is a full- duplex connection i.e. users can send and receive data simultaneously.
2. It allows half- closed connections.
3. The message's boundaries are maintained and application doesn't have to split messages.
4. It has properties of both TCP and UDP protocol.
5. It doesn't rely on IP layer for resilience of paths.

Disadvantages of SCTP:

1. One of key challenges is that it requires changes in transport stack on node.
2. Applications need to be modified to use SCTP instead of TCP/UDP.
3. Applications need to be modified to handle multiple simultaneous streams.

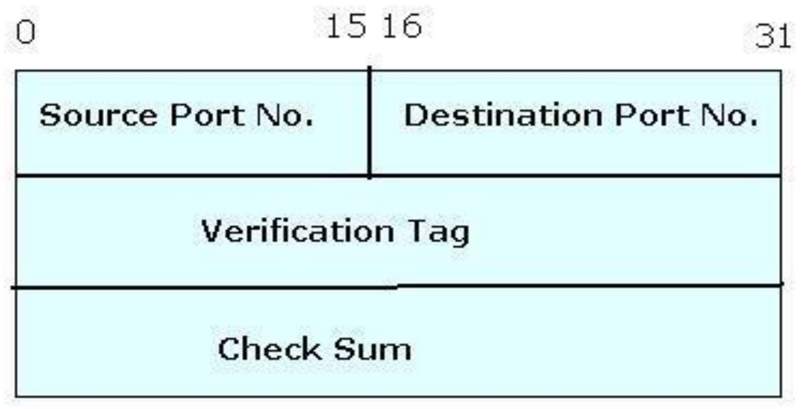
SCTP packet: An SCTP packet is a collection of information which may contain one or more chunks along with a header.

SCTP Packet Format : SCTP packet consists of a common header and chunks of information. These chunks can be either user defined data or some other controlled information
According to the Maximum Transmission Unit(MTU) size the SCTP packet can contain multiple chunks



.

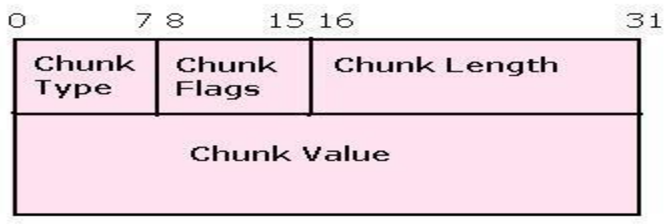
SCTP Header Format: The pictorial representation of the header of an SCTP packet is as follows:



SCTP Common Header Format

The header occupies the first 12 bytes of an SCTP packet and consists of a 16 bit (2 byte) Source port number, a 16 bit destination port number, 32 bit verification tag and a 32 bit checksum.
The port numbers specifies the source and the destination ports.
The checksum is a 32 bit no. which is used to check for erroneous packets received at the destination.

SCTP Chunk Field Format:



SCTP Chunk Field Format

The remaining part of the packet is filled up by chunks. Each chunk consists of an 8 bit type identifier which specifies the type of chunk that is being transmitted. There are 255 different types of chunks like 0 – payload, 1 – INIT etc.
Following the type is an 8 bit Chunk Flag followed by 16 bit chunk length and Chunk value. The chunk value is in multiple of 4 bytes and if it is not so it is padded with zeros at the end. These padding bits are not included in calculating the length of the chunk.

Transmission of Data by SCTP: The transmission of data takes place in the form of SCTP packets. The transmission takes place in the following steps:

1. Association Startup and Takedown - On receiving a request from an SCTP user the association/transmission begins and in the same way closes on user request at the end.

2. Path Management - At this step the availability and health of the path is checked/identified.

3. Packet validation - The first thing as the packet transfer begins is, to put the checksum and verification tag in the header and the first thing at the destination is the checking of erroneous packets. If the checksum and verification tag at destination do not match with the original ones in header, the packet is discarded.

4 .Chunk Bundling - Various chunks of data are assembled here to form the complete packet. Chunk bundling is also responsible for disassembling the packets at the receiving end.

5. Acknowledgment and congestion check - At this stage the packets are resend if a timely acknowledgment is not received from the destination.

6. User Data Fragmentation - Stream Control Transmission Protocol fragments user message so that the SCTP packets are in accordance to the MTU. At the destination, the fragments are reassembled before being passed to the user.

7. Sequenced delivery - Multiple streams of data are transmitted in parallel. This stage ensures that correct sequence of packet is received at the receiver end. This takes place at the destination only.

SCTP Socket Interfaces

When the **socket()** call creates a socket for IPPROTO_SCTP, it calls an SCTP-specific socket creation routine. Socket calls made on an SCTP socket call the appropriate SCTP socket routine automatically. In a one-to-one socket, each socket corresponds to one SCTP association. Create a one-to-one socket by calling this function:

socket(AF_INET[6], SOCK_STREAM, IPPROTO_STCP);

In a one-to-many style socket, each socket handles multiple SCTP associations. Each association has an association identifier called *sctp_assoc_t*. Create a one-to-many socket by calling this function:

socket(AF_INET[6], SOCK_SEQPACKET, IPPROTO_STCP);

i) sctp_bindx()

int sctp_bindx(int *sock,* void **addrs*, int *addrcnt,* int *flags*);

The **sctp_bindx()** function manages addresses on an SCTP socket. If the *sock* parameter is an IPv4 socket, the addresses passed to the **sctp_bindx()** function must be IPv4 addresses. If the *sock* parameter is an IPv6 socket, the addresses passed to the **sctp_bindx()** function can be either IPv4 or IPv6 addresses.

The value of the **addrs* parameter is a pointer to an array of one or more socket addresses. Each address is contained in its appropriate structure. If the addresses are IPv4 addresses, they are contained in either sockaddr_in structures or sockaddr_in6 structures. If the addresses are IPv6 addresses, they are contained in sockaddr_in6 structures. The address type's family distinguishes the address length. The caller specifies the number of addresses in the array with the *addrcnt* parameter.

The **sctp_bindx()** function returns 0 on success. The **sctp_bindx()** function returns -1 on failure and sets the value of errno to the appropriate error code.

The *flags* parameter is formed from performing the bitwise OR operation on zero or more of the following currently defined flags:

- SCTP_BINDX_ADD_ADDR
- SCTP_BINDX_REM_ADDR

SCTP_BINDX_ADD_ADDR directs SCTP to add the given addresses to the association. SCTP_BINDX_REM_ADDR directs SCTP to remove the given addresses from the association.

ii) sctp_opt_info():

int sctp_opt_info(int *sock*, sctp_assoc_id_t *id*, int *opt*, void **arg*, socklen_t **len*);

The **sctp_opt_info()** function returns the SCTP level options that are associated with the socket described in the *sock* parameter. If the socket is a one-to-many style SCTP socket the value of the *id* parameter refers to a specific association. The *id* parameter is ignored for one-to-one style SCTP sockets. The value of the *opt* parameter specifies the SCTP socket option to get. The value of the *arg* parameter is an option-specific structure buffer that is allocated by the calling program. The value of the **len* parameter is the length of the option.

The *opt* parameter can take on the following values:

SCTP_RTOINFO: Returns the protocol parameters that are used to initialize and bind the retransmission timeout (RTO) tunable.

SCTP_ASSOCINFO: Returns the association-specific parameters. The parameters use the following structure:

SCTP_PEER_ADDR_PARAMS: Returns the parameters for a specified peer address.

SCTP_STATUS: Returns the current status information about the association. The association can take on the following states:

- **SCTP_IDLE:** The SCTP endpoint does not have any association associated with it. Immediately after the call to the **socket()** function opens an endpoint, or after the endpoint closes, the endpoint is in this state.
- **SCTP_BOUND:** An SCTP endpoint is bound to one or more local addresses after calling the **bind()**.
- **SCTP_LISTEN:** This endpoint is waiting for an association request from any remote SCTP endpoint.
- **SCTP_COOKIE_WAIT:** This SCTP endpoint has sent an INIT chunk and is waiting for an INIT-ACK chunk.
- **SCTP_COOKIE_ECHOED:** This SCTP endpoint has echoed the cookie that it received from its peer's INIT-ACK chunk back to the peer.
- **SCTP_ESTABLISHED:** This SCTP endpoint can exchange data with its peer.
- **SCTP_SHUTDOWN_PENDING:** This SCTP endpoint has received a SHUTDOWN primitive from its upper layer. This endpoint no longer accepts data from its upper layer.
- **SCTP_SHUTDOWN_SEND:** An SCTP endpoint that was in the SCTP_SHUTDOWN_PENDING state has sent a SHUTDOWN chunk to its peer. The SHUTDOWN chunk is sent only after all outstanding data from this endpoint to its peer is acknowledged. When this endpoint's peer sends a SHUTDOWN ACK chunk, this endpoint sends a SHUTDOWN COMPLETE chunk and the association is considered closed.
- **SCTP_SHUTDOWN_RECEIVED:** An SCTP endpoint has received a SHUTDOWN chunk from its peer. This endpoint no longer accepts new data from its user.
- **SCTP_SHUTDOWN_ACK_SEND:** An SCTP endpoint in the SCTP_SHUTDOWN_RECEIVED state has sent the SHUTDOWN ACK chunk to its peer. The endpoint only sends the SHUTDOWN ACK chunk after the peer acknowledges all outstanding data from this endpoint. When this endpoint's peer sends a SHUTDOWN COMPLETE chunk, the association is closed.

iii) sctp_recvmsg()

ssize_t sctp_recvmsg(int *s*, void **msg*, size_t *len*, struct sockaddr **from*, socklen_t **fromlen*, struct sctp_sndrcvinfo **sinfo*, int **msg_flags*);

The **sctp_recvmsg()** function enables receipt of a message from the SCTP endpoint specified by the *s* parameter. The calling program can specify the following attributes:

msg

This parameter is the address of the message buffer.

len

This parameter is the length of the message buffer.

from

This parameter is a pointer to an address that contains the sender's address.

fromlen

This parameter is the size of the buffer associated with the address in the *from* parameter.

sinfo

This parameter is only active if the calling program enables sctp_data_io_events. To enable sctp_data_io_events, call the **setsockopt()** function with the socket option SCTP_EVENTS. When sctp_data_io_events is enabled, the application receives the contents of the sctp_sndrcvinfo structure for each incoming message. This parameter is a pointer to a sctp_sndrcvinfo structure. The structure is populated upon receipt of the message.

msg_flags

This parameter contains any message flags that are present.

The **sctp_recvmsg()** function returns the number of bytes it receives. The **sctp_recvmsg()** function returns -1 when an error occurs.

iv) sctp_sendmsg()

ssize_t sctp_sendmsg(int *s*, const void **msg*, size_t *len*, const struct sockaddr **to*, socklen_t *tolen*, uint32_t *ppid*, uint32_t *flags*, uint16_t *stream_no*, uint32_t *timetolive*, uint32_t *context*);

The **sctp_sendmsg()** function enables advanced SCTP features while sending a message from an SCTP endpoint.

s

This value specifies the SCTP endpoint that is sending the message.

msg

This value contains the message sent by the **sctp_sendmsg()** function.

len

This value is the length of the message. This value is expressed in bytes.

to

This value is the destination address of the message.

tolen

This value is the length of the destination address.

ppid

This value is the application-specified payload protocol identifier.

stream_no

This value is the target stream for this message.

timetolive

This value is the time period after which the message expires if it has not been successfully sent to the peer. This value is expressed in milliseconds.

context

This value is returned if an error occurs during the sending of the message.

flags

This value is formed from applying the logical operation OR in bitwise fashion on zero or more of the following flag bits:

MSG_UNORDERED

When this flag is set, the **sctp_sendmsg()** function delivers the message unordered.

MSG_ADDR_OVER

When this flag is set, the **sctp_sendmsg()** function uses the address in the *to* parameter instead of the association's primary destination address. This flag is only used with one-to-many style SCTP sockets.

MSG_ABORT

When this flag is set, the specified association aborts by sending an ABORT signal to its peer. This flag is only used with one-to-many style SCTP sockets.

MSG_EOF

When this flag is set, the specified association enters graceful shutdown. This flag is only used with one-to-many style SCTP sockets.

MSG_PR_SCTP

When this flag is set, the message expires when its transmission has not successfully completed within the time period specified in the timetolive parameter.

The **sctp_sendmsg()** function returns the number of bytes it sent. The **sctp_sendmsg()** function returns -1 when an error occurs.

v) sctp_send()

ssize_t sctp_send(int *s*, const void **msg*, size_t *len*, const struct sctp_sndrcvinfo **sinfo*, int *flags*);

The **sctp_send()** function is usable by one-to-one and one-to-many style sockets. The **sctp_send()** function enables advanced SCTP features while sending a message from an SCTP endpoint.

s

This value specifies the socket created by the **socket()** function.

msg

This value contains the message sent by the **sctp_send()** function.

len

This value is the length of the message. This value is expressed in bytes.

sinfo

This value contains the parameters used to send the message. For a one-to-many style socket, this value can contain the association ID to which the message is being sent.

flags

This value is identical to the flags parameter in the **sendmsg()** function.

The **sctp_send()** function returns the number of bytes it sent. The **sctp_send()** function returns -1 when an error occurs.