SOCKET PROGRAMMING

Sockets:

- 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** in computer networks are used for allowing the transmission of information between two processes of the same machines or different machines in the network.
- The socket is the combination of IP address and software port number used for communication between multiple processes.

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

Datagram Socket : This is a type of network that has a connectionless point for sending and receiving packets.

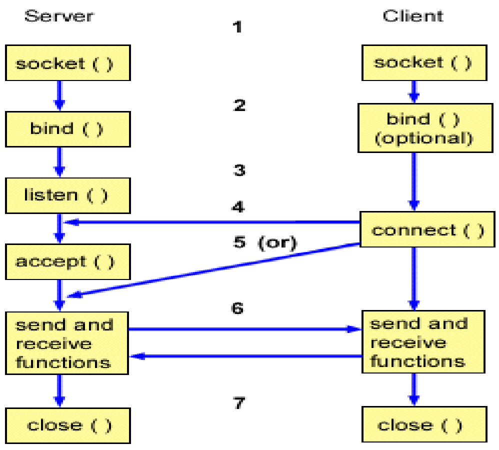
Stream Socket :A stream socket is a type of interprocess communications socket or network socket that provides a connection-oriented, sequenced, and unique flow of data without record boundaries with well-defined mechanisms for creating and destroying connections and for detecting errors.

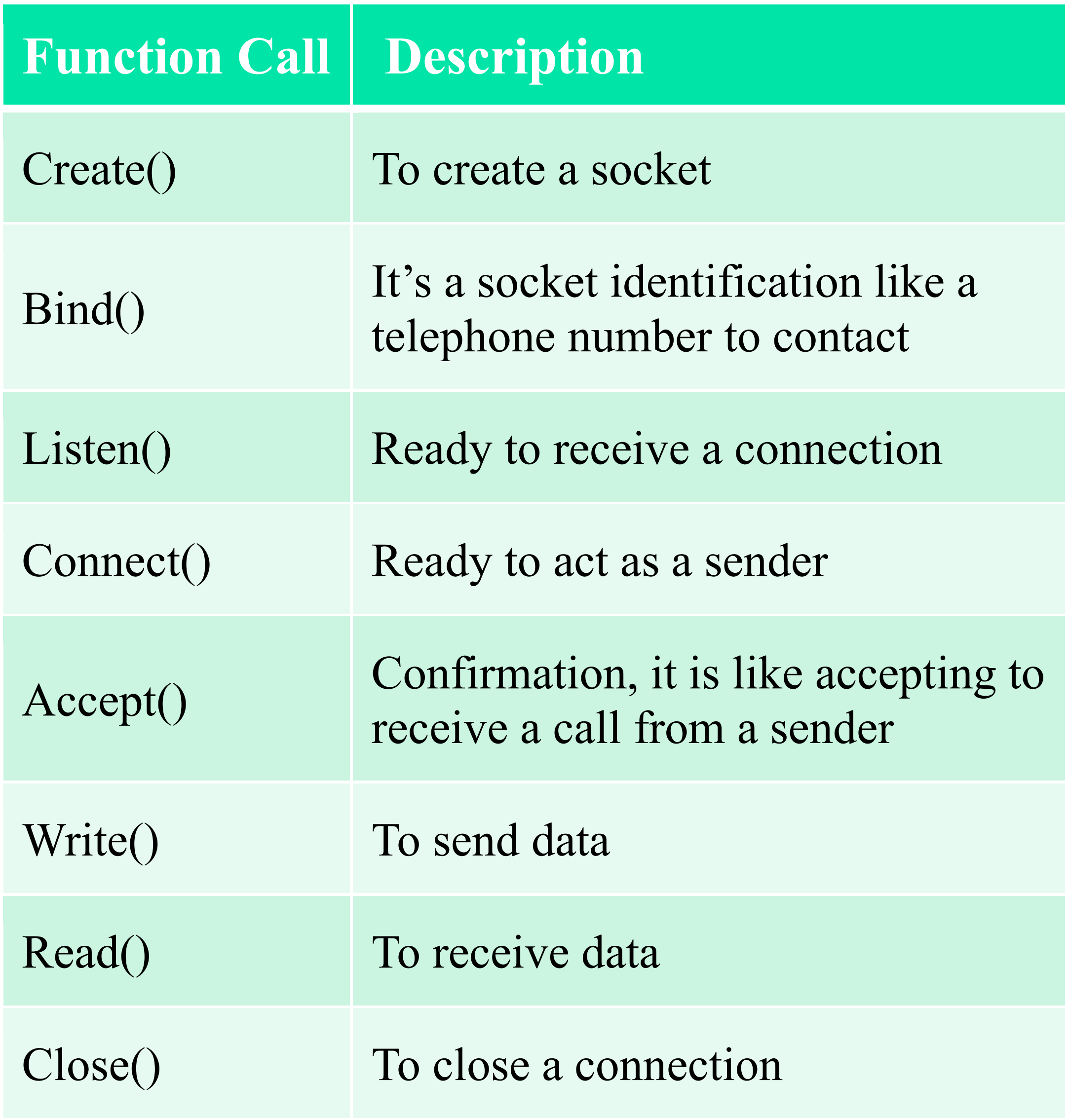
The socket API is a collection of socket calls that enables you to perform the following primary communication functions between application programs:

- Set up and establish connections to other users on the network
- Send and receive data to and from other users
- Close down connections

How sockets work:

- A socket has a typical flow of events.
- The following figure shows the typical flow of events(and the sequence of issued APIs)for a connection-oriented socket session. An explanation of each event follows the figure.





Function Call	Description
Create()	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

This is a typical flow of events for a connection-oriented socket:

1. The socket() API creates an endpoint for communications and returns a socket descriptor that represents the endpoint.
2. When an application has a socket descriptor, it can bind a unique name to the socket. Servers must bind a name to be accessible from the network.
3. The listen() API indicates a willingness to accept client connection requests. When a listen() API is issued for a socket, that socket cannot actively initiate connection requests. The listen() API is issued after a socket is allocated with a socket() API and the bind() API binds a name to the socket. A listen() API must be issued before an accept() API is issued.
4. The client application uses a connect() API on a stream socket to establish a connection to the server.
5. The server application uses the accept() API to accept a client connection request. The server must issue the bind() and listen() APIs successfully before it can issue an accept() API.

6. When a connection is established between stream sockets (between client and server), you can use any of the socket API data transfer APIs. Clients and servers have many data transfer APIs from which to choose, such as send(), recv(), read(), write(), and others.
7. When a server or client wants to stop operations, it issues a close() API to release any system resources acquired by the socket.

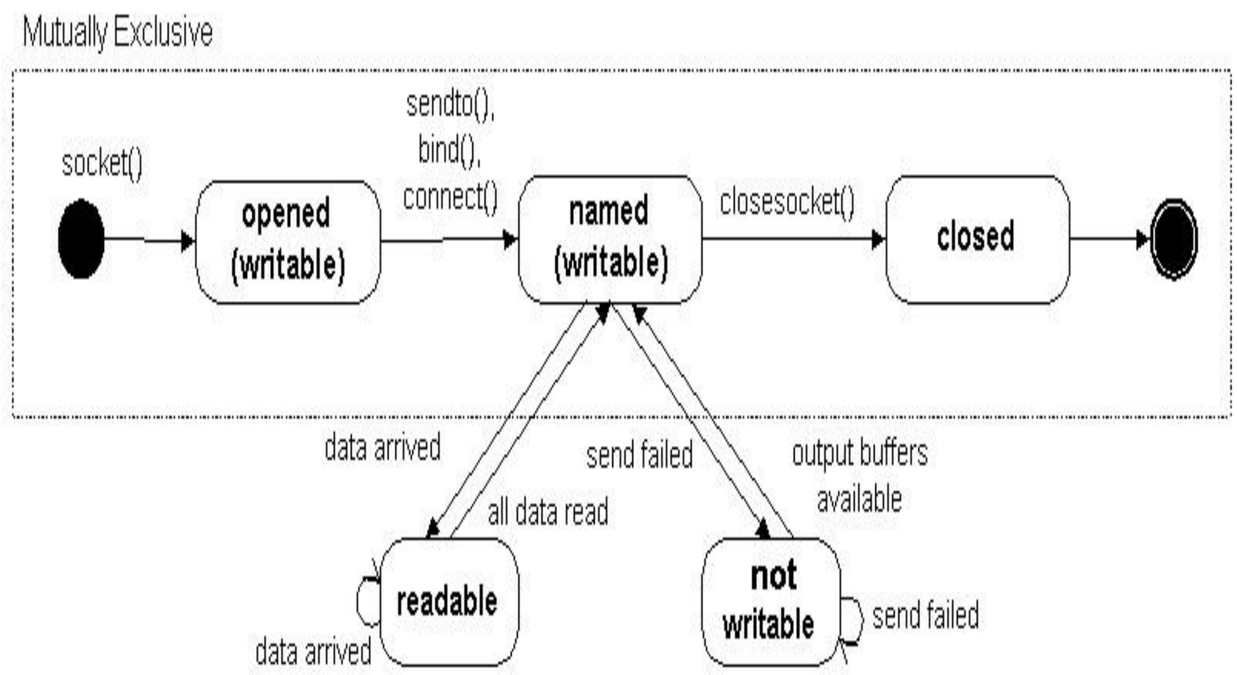
Socket characteristics

- A socket is represented by an integer. That integer is called a *socket descriptor*.
- A socket exists as long as the process maintains an open link to the socket.
- You can name a socket and use it to communicate with other sockets in a communication domain.
- Sockets perform the communication when the server accepts connections from them, or when it exchanges messages with them.

Socket States:

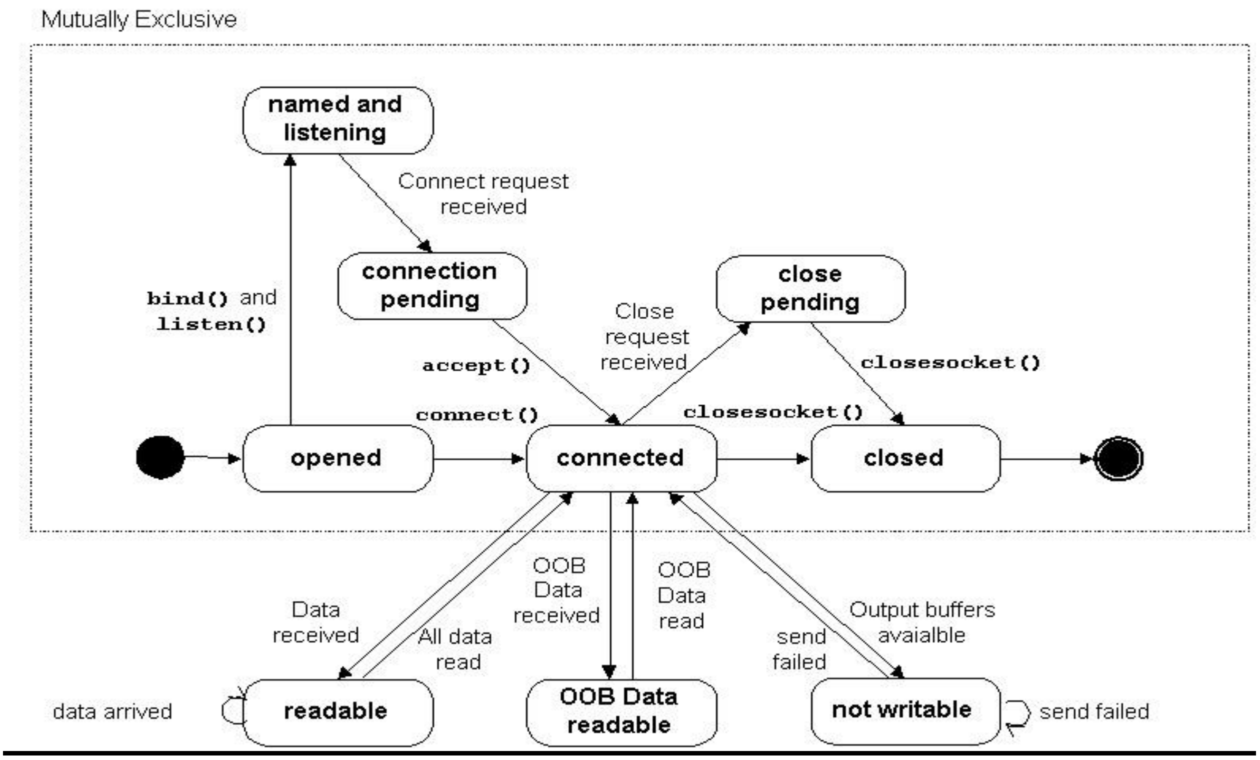
The state of a socket determines which network operations will succeed, which operations will block, and which operations with will fail (the socket state even determines the error code). Sockets have a finite number of states, and the WinSock API clearly defines the conditions that trigger a transition from one state to another.

DATAGRAM SOCKET STATES*:*



Socket State	Meaning
opened	**socket()** returned an unnamed socket (an unnamed socket is one that is not bound to a local address and port). The socket can be named explicitly with bind or implicitly with **sendto()** or **connect()**.
named	A named socket is one that is bound to a local address and a port. The socket can now send and/or receive.
readable	The network system received data and is ready to be read by the application using **recv()** or **recvfrom()**.
not writable	The network system does not have enough buffers to accommodate outgoing data.
closed	The socket handle is invalid.

STREAM SOCKET STATES:



Socket State	Meaning
opened	**socket()** returned an unnamed socket (an unnamed socket is one that is not bound to a local address and port). The socket can be named explicitly with **bind()** or implicitly with **conenct()**.
named and listening	The socket is named (bound to a local address and port) and is ready to accept incoming connection requests.
connection pending	The network system received an incoming connection requests and is waiting for the application to respond.
connected	An association (virtual circuit) has been established between a local and remote host. Sending and receiving data is now possible
readable	The network system received data and is ready to be read by the application using **recv()** or **recvfrom()**.
OOB readable	Out Of Band data received by the network system received data and is ready to be read by the application (using **recv()** or **recvfrom()**.)

Not writable	The network system does not have enough buffers to accommodate outgoing data.
close pending	The virtual circuit is close.
closed	The socket handle is invalid.

Socket Options

Various types of options are available in a socket. There are various ways to get and set the options that affect a socket. They include,

- getsockopt and setsockopt functions
- fcntl function
- ioctl function

(a) getsockopt and setsockopt

Syntax

```
int getsockopt (int sockfd, int level, int optname, void *optval, socklen_t *optlen);

int setsockopt (int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

Return Value : Both return 0 on OK, -1 on error.

sockfd – refer to an open socket descriptor

level –specifies the code in the system that interprets the option. (i.e) general socket code or protocol specific code (IPv4, IPv6,TCP)

optname – name of the option

optval – It is a pointer to a variable from which the new value of the option is fetched by setsockopt or into which the current value of the option is stored by getsockopt.

Optlen – specifies the size of this variable.

(b) fcntl

fcntl stands for "file control". This function performs various descriptor control operations.

Syntax

```
int fcntl (int fd, int cmd, ….int arg);

Return value: 0 if OK, -1 on error
```

(c) ioctl

ioctl stands for “IO control”.

Syntax

```
int ioctl (int fd, int request, ….void *arg);

Return value: 0 if OK, -1 on error
```

Generic Socket Options

These options are protocol independent options. These options are as follows.

(1) SO_BROADCAST

- This option enables or disables the ability of the process to send broadcast messages.
- Broadcasting is supported only for datagram sockets and only on networks that support the concept of broadcast message.
- An application must set this socket option before sending any broadcast message.
- If the destination address is a broadcast address and this socket option is not set, EACCESS is returned.

(2) SO_DEBUG

- This option is supported only by TCP.
- When this option is enabled for a TCP socket, the kernel keeps track of the detailed information about all the packets send and received by the TCP for the socket.
- These are kept in a circular buffer and can be examined with the trpt program.

(3) SO_DONTROUTE

- This option specifies that the outgoing packets are to bypass the normal routing mechanisms of the underlying protocol.
- According to the destination address given, the packets will be routed.
- So, a local interface will be identified and then the packet is routed.
- If the local interface cannot be identified, ENETUNREACH is returned.

- This option can also be applied to individual datagrams using MSG_DONTROUTE flag.
- This option is often used by the routing daemons to bypass the routing table and force a packet to be sent out a particular interface.

(4) SO_ERROR

- When an error occurs on a socket, the protocol module sets a variable named so_error for that socket to one of the standard unix Exxx values. This is called the pending error for the socket.
- This option can be fetched but cannot be set.
- The process can be notified about the error in one of the two ways.
  - If the process is blocked in a call to select on the socket, for either readability or writability.
  - If the process is using signal driven I/O, the SIGIO signal is generated for either the process or the process group.

(5) SO_KEEPALIVE

- The purpose of this option is to detect if the peer host crashes or become unreachable.
- When the keepalive option is set for a TCP socket and no data has been exchanged across the socket in either direction for 2 hours, TCP automatically sends a keep-alive probe to the peer.
- This probe is a TCP segment to which the peer must respond. One of the three scenarios result.
  - The peer responds with the expected ACK (If the peer is active)
  - The peer responds with an RST, which tells the local TCP that the peer host has been crashed and rebooted. So, the sockets pending error is set to ECONNRESET and the socket is closed.
  - There is no response from the peer to the keep-alive probe. TCP sends 8 additional probes, 75 seconds apart, trying to get a response from the peer. It will give up if there is no response within 11 minutes and 15 seconds from the time of sending the first probe. If there is no response, the sockets pending error is set to ETIMEDOUT and the socket are closed. If the peer host is unreachable, the pending error is set to EHOSTUNREACH.

- This option is normally used by servers, although clients can also use this option.

- Servers use this option because after establishment of connection, there may be a situation where the server may wait for the client request.
- But, if the client hosts crashes, powered off or connection drops, the server never knows about it and waits for the input that can never arrive. This is called a half open connection. The keep-alive option will detect these half open connections and terminate them.

(6) SO_LINGER

- This option specifies how the close function operates for a connection oriented protocol.
- By default, close returns immediately. But, if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer.
- But, the SO_LINGER changes this default case.
- It requires the following structure to be passed between the user process and the kernel.

```
Struct linger
{
        int l_onoff;     /* 0 = off, non-zero = on */
        int l_linger     /* linger time */
}
```

- When this socket option is set, any one of the following three scenarios takes place, depending on the values of the two structure members.
    - If l_onoff=0, the option is turned off. So, the value of l_linger is ignored and the TCP default applies (i.e) close returns immediately.
    - If l_onoff=nonzero and l_linger = 0, TCP aborts the connection when it is closed. (ie) TCP discards any data still remaining in the socket send buffer and sends a RST to its peer.
    - If l_onoff=nonzero and l_linger = nonzero, then the kernel will linger when the socket is closed. (i.e) If there is any data still remaining in the socket send buffer, the process is put to sleep until either,
        - All data is send and acknowledged by the peer TCP.
        - The linger time expires.
- Assume that the client writes data to the socket and then calls close. The following diagrams depict the various scenarios.

(a) Default situation

- By default, close returns immediately.

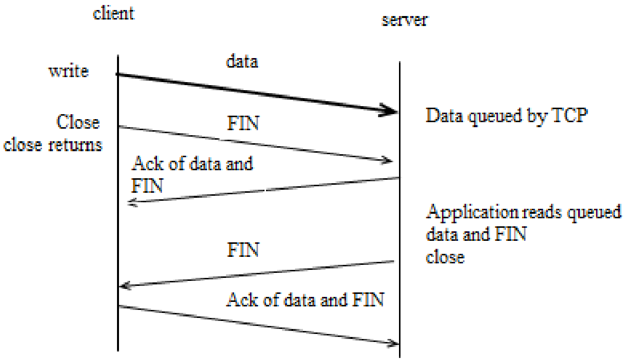


Fig 3.1 Default operation of close

We now need to look at exactly when *close* on a socket returns and what the actions and consequences are. In these cases, we assume that the client writes data to the socket and then calls close.

Fig. 3.1 shows the default scenario. Assume that when the client's data arrives, the server is temporarily busy, so the data is added to the socket receive buffer by its TCP. Similarly, the next segment, the client's FIN is also added.

But by default, the client's close returns immediately. As we see here, the client's close can return before the server reads the remaining data in its socket receive buffer. Therefore it is possible for the server host to crash before the server application reads this remaining data, and the client application will never know.

(b) SO_LINGER socket option is set and l_linger set to a positive value

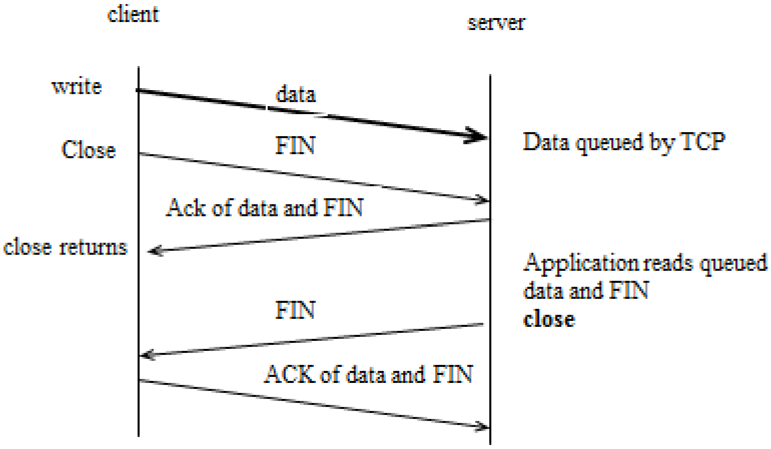


Fig 3.2 l_linger set to a positive value

In this scenario, the client sets the SO_LINGER option, specifying some positive linger time. When this occurs, the client's close does not return until all the client's data and its FIN have been acknowledged by the server TCP as shown in Fig 3.2.

The server host can crash before the server application reads its remaining data, and the client application will never know.

(c) SO_LINGER socket option set with l_linger set to small positive value

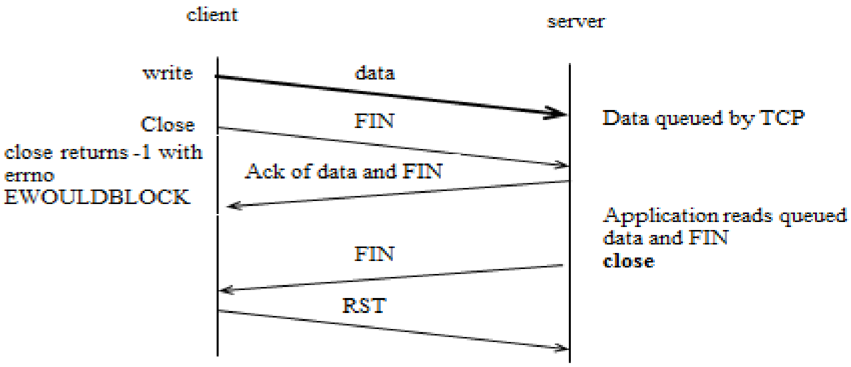


Fig 3.3 l_linger set to small positive value

Fig. 3.3 shows what can happen if the SO_LINGER option is set to a value that is too low. The basic principle here is that a successful return from close, with the SO_LINGER option set, only tells us that the data we sent (and our FIN) have been acknowledge by the peer TCP. It does not tell us whether the peer application has read the data. If we do not set the SO_LINGER option, we do not know whether the peer TCP has acknowledged the data.

(d) Using shutdown to show the peer has received the data

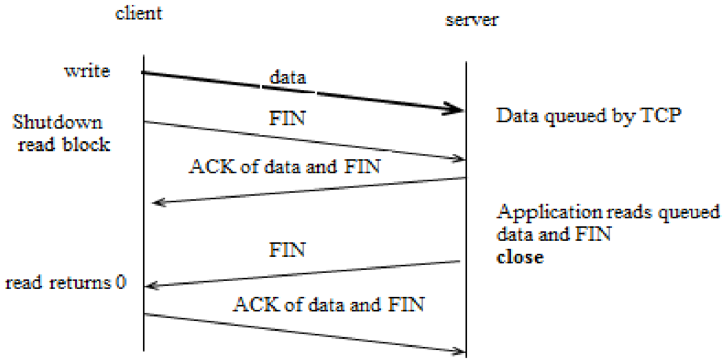


Fig. 3.4 Using shutdown

One way for the client to know that the server has read its data is to call shutdown (with SHUT_WR) instead of close and wait for the peer to close its end of the connection as shown in Fig. 3.4.

(e) Application ACK

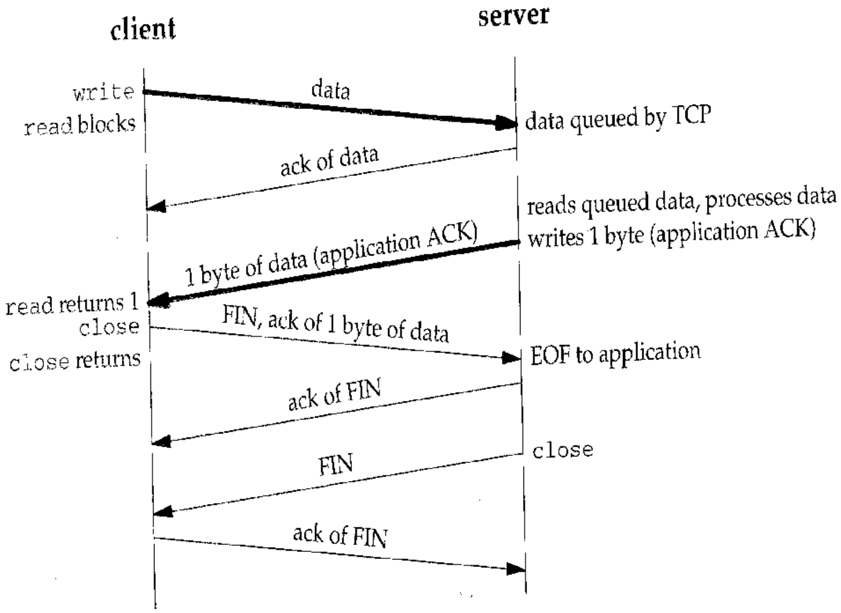


Fig. 3.5 Application ACK

Another way to know that the peer application has read our data is to use an application-level acknowledgment which requires coding in the server and client. In this case, the client waits for a 1 byte acknowledgment for each packet sent. Fig 3.5 shows the possible packet exchange.

(7) SO_OOBINLINE

- When this option is enabled, the out of band data will be placed in the normal input queue.
- When this occurs, the MSG_OOB flag to the receive functions cannot be used to read the out of band data.

(8) SO_RCVBUF and SO_SNDBUF

- Every socket has a send buffer and a receive buffer.
- Receive buffer - It is used to hold the received data until it is read by the application.
- Send buffer – It is used to hold the data to be send.

- The socket receive buffer has a limit in its window size. And, the peer can send data only upto the window size limit. The window size will be advertised to its peer while sending the SYN segment during connection establishment. This is TCPs flow control.
- If the peer ignores the advertised window and if it sends data beyond the window, the receiving TCP discards it.
- The default size of TCP send and receive buffers is 4096 bytes. But, newer systems use larger values from 8192 to 61440 bytes.
- The default size of UDP send buffer is 9000 bytes.
- The default size of UDP receive buffer is 40000 bytes.
- The main goal of this option is that these two options let us change the default sizes.

(9) SO_RCVLOWAT and SO_SNDLOWAT

- Every socket has a receive low water mark and send low water mark.
- These are used by the select function.
- Receive low water mark – It is the amount of data that must be in the socket receive buffer for the select to return readable.
- Send low water mark – It is the amount of available space that must exist in the socket send buffer for select to return writable.
- Default receive low water mark is 1.
- Default send low water mark is 2048.
- These two socket options, let us change these two low water marks.

(10) SO_RCVTIMEO and SO_SNDTIMEO

- These two socket options allow us to place a timeout on socket receive and send.
- This let us specify the timeout in seconds and microseconds.
- The timeout can be disabled by setting its value to 0 seconds and 0 microseconds.
- Both timeouts are disabled by default.
- The receive timeout affects the five input functions namely read, readv, recv, recvfrom and recvmsg.
- The send timeout affects the five output functions namely write, writev, send, sendto and sendmsg.

(11) S0_REUSEADDR and SO_REUSEPORT

- SO_REUSEADDR serves four different purposes.
    - It allows a listening server to start and bind its well known port even if previously established connections exist that use this port as their local port. This condition is typically encountered as follows.
        - A listening server is started.
        - A connection request arrives and a child process is spawned to handle that client.
        - The listening server terminates, but the child continues to service the client on the existing connection.
        - The listening server is restarted.
    - It allows a new server to be started on the same port as an existing server that is bound to the wildcard address as long as each instance binds a different local IP address.
    - It allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address.
    - It allows completely duplicate bindings : A bind of an IP address and port, when the same IP address and port are already bound to another socket, if the transport protocol supports it.
- This feature is supported only for UDP sockets.
- This feature is used with multicasting to allow the same application to be run multiple times on the same host.
- SO_REUSEADDR does the following.
    - It allows completely duplicate bindings, but only if each socket that wants to bind the same IP address and port specify this socket option.
    - It is considered equivalent to SO_REUSEPORT if the IP address being bound is a multicast address.
- Limitation : It is not supported by all systems.

(12) SO_TYPE

- This option returns the socket type.
- The integer value returned is a value SOCK_STREAM or SOCK_DGRAM or SOCK_RAW.

(13) SO_USELOOPBACK

- This option applies only to sockets in the routing domain.

- By default, this set to ON.
- When this option is enabled, the socket receives a copy of everything sent on the socket.

IPv4 Socket Options

- These socket options are processed by IPv4. These options include the following.

(1) IP_HDRINCL

  - If this option is set for a raw IP socket, we must build our own IP header for all the datagrams we send on the raw socket.
  - Normally kernel builds the IP header for all datagrams, but some applications require to build their own IP header.
  - When this option is set, we build a complete IP header, with the following exceptions.
    - IP always calculates and stores the IP header checksum.
    - If we set the IP identification field to 0, the kernel will set the field.
    - If the source IP address is INADDR_ANY, IP sets it to the primary IP address of the outgoing interface.
    - Setting IP options is implementation dependent.
    - Some fields must be in host byte order and some in network byte order. This is implementation dependent.

(2) IP_OPTIONS

  - Setting this option allows us to set IP options in the IPv4 header.
  - This requires intimate knowledge of the format of IP options in the IP header.

(3) IP_RECVDSTADDR

  - This option causes the destination IP address of a received UDP datagram to be returned as ancillary data by recvmsg.

(4) IP_RECVIF

  - This option causes the index of the interface on which a UDP datagram is received to be returned as ancillary data by recvmsg.

(5) IP_TOS

  - This option let us set the type of service in the IP header for a TCP, UDP socket.
  - TOS can be,
    - T – Throughput
    - R – Reliability

        - D – Delay
        - C - Cost

(6) IP_TTL

    - TTL stands for Time to Live
    - This option let us set and fetch the default TTL.

3.2 ICMPv6 Socket Option

- This socket option is processed by ICMPv6.

(1) ICMP_FILTER

    - This option let us fetch and set an icmp6_filter structure that specifies which of the 256 possible ICMPv6 message types will be passed to the process on a raw socket.

IPv6 Socket Option

- These socket options are processed by IPv6. These options include the following.

(1) IPv6_CHECKSUM

    - This option specifies the byte offset into the user data where the checksum field is located.
    - If this value is non-negative, the kernel will,
        - Compute and store a checksum for all outgoing packets.
        - Verify the received checksum on input, discarding packets with an invalid checksum.
    - If the value is -1 (default), the kernel will not calculate and store the checksum for outgoing packets on this raw socket and will not verify the checksum for received packets.

(2) IPv6_DONTFRAG

    - Setting this option disables the automatic insertion of a fragment header for UDP and raw sockets.
    - When this option is set, output packets larger than Maximum Transfer Unit (MTU) of the outgoing interface will be dropped.

(3) IPv6_NEXTHOP

    - This option specifies the next hop address for a datagram as a socket address structure and is a privileged operation.

(4) IPv6_PATHMTU

- This option cannot be set, only retrieved.
- When this option is retrieved, the current MTU as determined by PATH_MTU discovery is returned.

(5) IPv6_RECVDSTOPTS

- Setting this option specifies that any received IPv6 destination options are to be returned as ancillary data by recvmsg.

(6) IPv6_RECVHOPLIMIT

- Setting this option specifies that the received hop limit field is to be returned as ancillary data by recvmsg.

(7) IPv6_RECVHOPOPTS

- Setting this option specifies that any received IPv6 hop-by-hop options are to be returned as ancillary data by recvmsg.

(8) IPv6_RECVPATHMTU

- Setting this option specifies that the path MTU of a path is to be returned as ancillary data by recvmsg.

(9) IPv6_RECVPKTINFO

- Setting this option specifies that the following two pieces of information about a received IPv6 datagram are to be returned as ancillary data by recvmsg.
  - The destination IPv6 address
  - Arriving interface index

(10) IPv6_RECVRTHDR

- Setting this option specifies that a received IPv6 routing header is to be returned as an ancillary date by recvmsg.

(11) IPv6_RECVTCLASS

- Setting this option specifies that the received traffic class is to be returned as ancillary data by recvmsg.

(12) IPv6_UNICAST_HOPS

- Setting this option specifies the default hop limit for outgoing datagrams sent on the socket, while fetching the socket option returns the value of the hop limit that the kernel will use for the socket.

(13) IPv6_USE_MIN_MTU

- Setting this option avoids fragmentation.
  - When this option is set to 1, path MTU discovery is not performed and packets are sent using minimum MTU.

  - When this option is set to 0, causes path MTU discovery to occur for all destinations.
  - When this option is set to -1, path MTU discovery is performed.

(14) IPv6_V6ONLY

- Setting this option restricts it to IPv6 communication only.

(15) IPv6_XXX

- UDP socket uses, recvmsg and sendmsg
- TCP socket uses, getsockopt and setsockopt

TCP Socket Options

(1) TCP_MAXSEG

- This socket option allows us to fetch or set the Maximum Segment Size (MSS) for a TCP connection.
- The value returned is the maximum amount of data that the TCP will send to the other end.
- The MSS is set while sending the SYN segment to the peer during connection establishment.
- The maximum amount of data that our TCP will send per segment can also change during the life of the connection if TCP supports path MTU discovery.
- If the route of the peer changes, this value will go up or down.

(2) TCP_NODELAY

- If this option is set, it disables TCP's Nagle algorithm.
- By default, this algorithm is enabled.
- Nagles algorithm avoids the syndrome caused in the sender side (i.e) if the sending side sends data too slowly, by sending each byte as a packet and waiting for the acknowledgment.

Nagle's Algorithm

(1) It sends the first byte as it is as a packet and waits for an acknowledgment.
(2) When it receives the ACK, it does not send the further byte as it is, provided it waits until a certain number of bytes gets accumulated or till the ACK for the previous is arrived.

- The purpose of Nagle's algorithm is to reduce the number of small packets in WAN.
- Small packet is any packet smaller than MSS.
- The two common generators of small packets are the Rlogin and Telnet clients, since they send each keystroke as a separate packet.
- In a fast LAN, we normally donot notice a Nagle's algorithm because the time required for a small packet to be acknowledged is typically a few milliseconds, far less than the time between two successive characters that we type.
- But in a WAN, it takes nearly a second to acknowledge a small packet, so we can notice a delay in the character echoing and this delay is often exaggerated by the Nagle's algorithm.
- Consider the following example,
    - We type the six character string "hello!" with exactly 250 ms between each character.
    - The Round Trip Time (RTT) to the server is 600 ms and the server immediately sends back the echo of each character.
    - Assuming the Nagle's algorithm is disabled, we have the 12 packets as shown below.

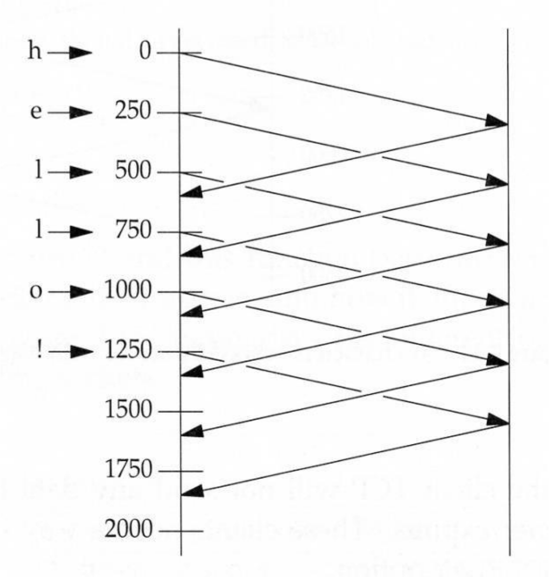


Fig 3.6 Nagle's algorithm (disabled)

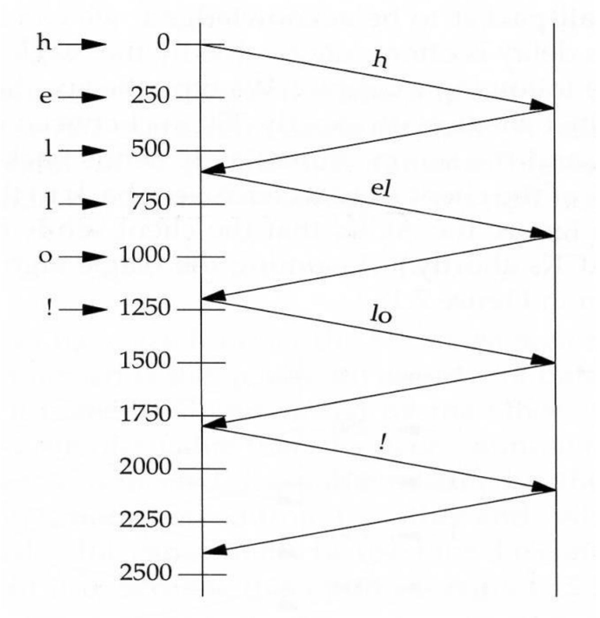


Fig. 3.7 Nagle’s Algorithm (Enabled)

- The purpose of the Nagle algorithm is to reduce the number of small packets on a WAN.
- The algorithm states that if a given connection has outstanding data then no small packets will be sent on the connection in response to a user write operation until the existing data is acknowledged.
- Two common generators of small packets are the rlogin and telnet clients, since they normally send each keystroke as a separate packet.
- Fig. 3.6 shows the algorithm disabled. Fig. 3.7 shows the algorithm enabled.
- The characters are typed with 250 milliseconds between each.
- Round Trip Time (RTT) is 600 milliseconds.

ICMPv6 Socket Option

- This socket option is processed by ICMPv6.

(1) ICMP_FILTER

  - This option let us fetch and set an icmp6_filter structure that specifies which of the 256 possible ICMPv6 message types will be passed to the process on a raw socket.

IPv6 Socket Option

- These socket options are processed by IPv6. These options include the following.

(1) IPv6_CHECKSUM

  - This option specifies the byte offset into the user data where the checksum field is located.
  - If this value is non-negative, the kernel will,
    - Compute and store a checksum for all outgoing packets.
    - Verify the received checksum on input, discarding packets with an invalid checksum.
  - If the value is -1 (default), the kernel will not calculate and store the checksum for outgoing packets on this raw socket and will not verify the checksum for received packets.

(2) IPv6_DONTFRAG

  - Setting this option disables the automatic insertion of a fragment header for UDP and raw sockets.
  - When this option is set, output packets larger than Maximum Transfer Unit (MTU) of the outgoing interface will be dropped.

(3) IPv6_NEXTHOP

  - This option specifies the next hop address for a datagram as a socket address structure and is a privileged operation.

(4) IPv6_PATHMTU

- This option cannot be set, only retrieved.
- When this option is retrieved, the current MTU as determined by PATH_MTU discovery is returned.

(5) IPv6_RECVDSTOPTS

- Setting this option specifies that any received IPv6 destination options are to be returned as ancillary data by recvmsg.

(6) IPv6_RECVHOPLIMIT

- Setting this option specifies that the received hop limit field is to be returned as ancillary data by recvmsg.

(7) IPv6_RECVHOPOPTS

- Setting this option specifies that any received IPv6 hop-by-hop options are to be returned as ancillary data by recvmsg.

(8) IPv6_RECVPATHMTU

- Setting this option specifies that the path MTU of a path is to be returned as ancillary data by recvmsg.

(9) IPv6_RECVPKTINFO

- Setting this option specifies that the following two pieces of information about a received IPv6 datagram are to be returned as ancillary data by recvmsg.
    - The destination IPv6 address
    - Arriving interface index

(10) IPv6_RECVRTHDR

- Setting this option specifies that a received IPv6 routing header is to be returned as an ancillary date by recvmsg.

(11) IPv6_RECVTCLASS

- Setting this option specifies that the received traffic class is to be returned as ancillary data by recvmsg.

(12) IPv6_UNICAST_HOPS

- Setting this option specifies the default hop limit for outgoing datagrams sent on the socket, while fetching the socket option returns the value of the hop limit that the kernel will use for the socket.

(13) IPv6_USE_MIN_MTU

- Setting this option avoids fragmentation.
    - When this option is set to 1, path MTU discovery is not performed and packets are sent using minimum MTU.

        - When this option is set to 0, causes path MTU discovery to occur for all destinations.
        - When this option is set to -1, path MTU discovery is performed.

(14) IPv6_V6ONLY

- Setting this option restricts it to IPv6 communication only.

(15) IPv6_XXX

- UDP socket uses, recvmsg and sendmsg
- TCP socket uses, getsockopt and setsockopt

TCP Socket Options

(1) TCP_MAXSEG

- This socket option allows us to fetch or set the Maximum Segment Size (MSS) for a TCP connection.
- The value returned is the maximum amount of data that the TCP will send to the other end.
- The MSS is set while sending the SYN segment to the peer during connection establishment.
- The maximum amount of data that our TCP will send per segment can also change during the life of the connection if TCP supports path MTU discovery.
- If the route of the peer changes, this value will go up or down.

(2) TCP_NODELAY

- If this option is set, it disables TCP's Nagle algorithm.
- By default, this algorithm is enabled.
- Nagles algorithm avoids the syndrome caused in the sender side (i.e) if the sending side sends data too slowly, by sending each byte as a packet and waiting for the acknowledgment.

Nagle's Algorithm

(1) It sends the first byte as it is as a packet and waits for an acknowledgment.
(2) When it receives the ACK, it does not send the further byte as it is, provided it waits until a certain number of bytes gets accumulated or till the ACK for the previous is arrived.

- The purpose of Nagle's algorithm is to reduce the number of small packets in WAN.
- Small packet is any packet smaller than MSS.
- The two common generators of small packets are the Rlogin and Telnet clients, since they send each keystroke as a separate packet.
- In a fast LAN, we normally donot notice a Nagle's algorithm because the time required for a small packet to be acknowledged is typically a few milliseconds, far less than the time between two successive characters that we type.
- But in a WAN, it takes nearly a second to acknowledge a small packet, so we can notice a delay in the character echoing and this delay is often exaggerated by the Nagle's algorithm.
- Consider the following example,
  - We type the six character string "hello!" with exactly 250 ms between each character.
  - The Round Trip Time (RTT) to the server is 600 ms and the server immediately sends back the echo of each character.
  - Assuming the Nagle's algorithm is disabled, we have the 12 packets as shown below.

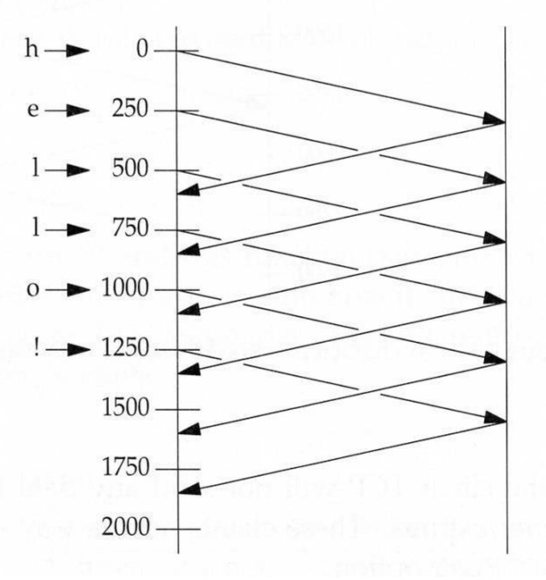


Fig 3.6 Nagle's algorithm (disabled)

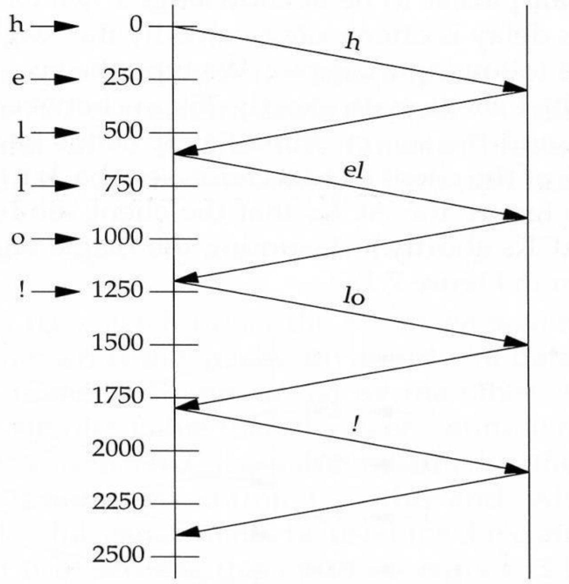


Fig. 3.7 Nagle's Algorithm (Enabled)

Web sockets

Web sockets are defined as a two-way communication between the servers and the clients, which mean both the parties, communicate and exchange data at the same time.

This protocol defines a full duplex communication from the ground up. Web sockets take a step forward in bringing desktop rich functionalities to the web browsers.

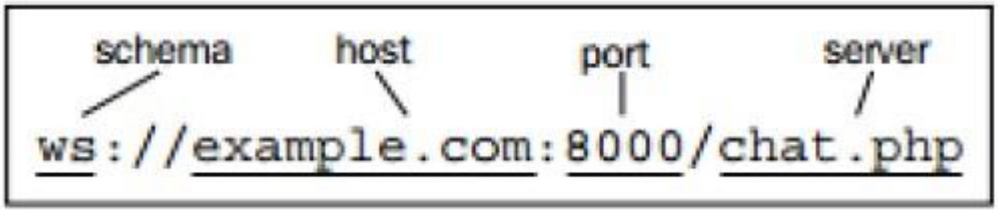
The main features of web sockets are as follows:

- Web socket protocol is being standardized, which means real time communication between web servers and clients is possible with the help of this protocol.
- Web sockets are transforming to cross platform standard for real time communication between a client and the server.
- This standard enables new kind of the applications. Businesses for real time web application can speed up with the help of this technology.
- The biggest advantage of Web Socket is it provides a two-way communication (fullduplex) over a single TCP connection.

URL

HTTP has its own set of schemas such as http and https. Web socket protocol also has similar schema defined in its URL pattern.

The following image shows the Web Socket URL in tokens.



Browser Support

The latest specification of Web Socket protocol is defined as **RFC 6455** – a proposed standard.

RFC 6455 is supported by various browsers like Internet Explorer, Mozilla Firefox, Google Chrome, Safari, and Opera.

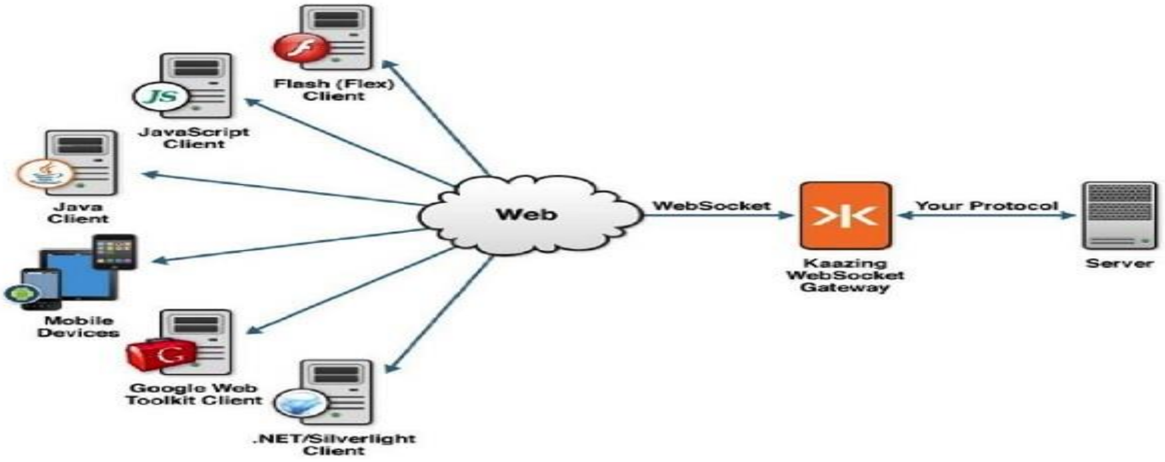
Functionalities

Web Socket represents a major upgrade in the history of web communications. Before its existence, all communication between the web clients and the servers relied only on HTTP.

Web Socket helps in dynamic flow of the connections that are persistent full duplex. Full duplex refers to the communication from both the ends with considerable fast speed.

It is termed as a game changer because of its efficiency of overcoming all the drawbacks of existing protocols.

The following diagram describes the functionalities of Web Sockets −



Web Socket connections are initiated via HTTP; HTTP servers typically interpret Web Socket handshakes as an Upgrade request.

Web Sockets can both be a complementary add-on to an existing HTTP environment and can provide the required infrastructure to add web functionality. It relies on more advanced, full duplex protocols that allow data to flow in both directions between client and server.

Events

There are four main Web Socket API **events** −

- Open
- Message
- Close
- Error

Each of the events are handled by implementing the functions like **onopen, onmessage**, **onclose** and **onerror** functions respectively. It can also be implemented with the help of addEventListener method.

The brief overview of the events and functions are described as follows −

Open

Once the connection has been established between the client and the server, the open event is fired from Web Socket instance. It is called as the initial handshake between client and server. The event, which is raised once the connection is established, is called **onopen**.

Message

Message event happens usually when the server sends some data. Messages sent by the server to the client can include plain text messages, binary data or images. Whenever the data is sent, the **onmessage** function is fired.

Close

Close event marks the end of the communication between server and the client. Closing the connection is possible with the help of **onclose** event. After marking the end of communication with the help of **onclose** event, no messages can be further transferred between the server and the client. Closing the event can happen due to poor connectivity as well.

Error

Error marks for some mistake, which happens during the communication. It is marked with the help of **onerror** event. **Onerror** is always followed by termination of connection. The detailed description of each and every event is discussed in further chapters.

Actions

Events are usually triggered when something happens. On the other hand, actions are taken when a user wants something to happen. Actions are made by explicit calls using functions by users.

The Web Socket protocol supports two main actions, namely −

- send( )
- close( )

send ( )

This action is usually preferred for some communication with the server, which includes sending messages, which includes text files, binary data or images.

A chat message, which is sent with the help of send() action, is as follows −

```
// get text view and button for submitting the message
vartextsend=document.getElementById(“text-view”);
varsubmitMsg=document.getElementById(“tsend-button”);

//Handling the click event
submitMsg.onclick=function(){
// Send the data
socket.send(textsend.value);
}
```

Note − Sending the messages is only possible if the connection is open.

close ( )

This method stands for goodbye handshake. It terminates the connection completely and no data can be transferred until the connection is re-established.

```
vartextsend=document.getElementById(“text-view”);
varbuttonStop=document.getElementById(“stop-button”);

//Handling the click event
buttonStop.onclick=function(){
// Close the connection if open
if(socket.readyState===WebSocket.OPEN){
socket.close();
}
}
```

It is also possible to close the connection deliberately with the help of following code snippet −

```
socket.close(1000,”DeliberateConnection”);
```

Opening Connections

Once a connection has been established between the client and the server, the open event is fired from Web Socket instance. It is called as the initial handshake between client and server.

The event, which is raised once the connection is established, is called the **onopen**. Creating Web Socket connections is really simple. All you have to do is call the **WebSocket constructor** and pass in the URL of your server.

The following code is used to create a Web Socket connection −

```
// Create a new WebSocket.
var socket =newWebSocket('ws://echo.websocket.org');
```

Once the connection has been established, the open event will be fired on your Web Socket instance.

onopen refers to the initial handshake between client and the server which has lead to the first deal and the web application is ready to transmit the data.

The following code snippet describes opening the connection of Web Socket protocol −

```
socket.onopen=function(event){
console.log(“Connection established”);
// Display user friendly messages for the successful establishment of connection
var.label=document.getElementById(“status”);
label.innerHTML=”Connection established”;
}
```

It is a good practice to provide appropriate feedback to the users waiting for the Web Socket connection to be established. However, it is always noted that Web Socket connections are comparatively fast.

Handling Errors

Once a connection has been established between the client and the server, an **open** event is fired from the Web Socket instance. Error are generated for mistakes, which take place during the communication. It is marked with the help of **onerror** event. **Onerror** is always followed by termination of connection.

The **onerror** event is fired when something wrong occurs between the communications. The event **onerror** is followed by a connection termination, which is a **close** event.

A good practice is to always inform the user about the unexpected error and try to reconnect them.

```
socket.onclose=function(event){
console.log("Error occurred.");

// Inform the user about the error.
var label =document.getElementById("status-label");
label.innerHTML="Error: "+event;
}
```

When it comes to error handling, you have to consider both internal and external parameters.

- Internal parameters include errors that can be generated because of the bugs in your code, or unexpected user behavior.
- External errors have nothing to do with the application; rather, they are related to parameters, which cannot be controlled. The most important one is the network connectivity.
- Any interactive bidirectional web application requires, well, an active Internet connection.

Send & Receive Messages

The **Message** event takes place usually when the server sends some data. Messages sent by the server to the client can include plain text messages, binary data, or images. Whenever data is sent, the **onmessage** function is fired.

This event acts as a client's ear to the server. Whenever the server sends data, the **onmessage** event gets fired.

The following code snippet describes opening the connection of Web Socket protocol.

```
connection.onmessage=function(e){
varserver_message=e.data;
  console.log(server_message);
}
```

It is also necessary to take into account what kinds of data can be transferred with the help of Web Sockets. Web socket protocol supports text and binary data. In terms of Javascript, **text** refers to as a string, while binary data is represented like **ArrayBuffer**.

Web sockets support only one binary format at a time. The declaration of binary data is done explicitly as follows −

```
socket.binaryType=”arrayBuffer”;
socket.binaryType=”blob”;
```

Strings

Strings are considered to be useful, dealing with human readable formats such as XML and JSON. Whenever **onmessage** event is raised, client needs to check the data type and act accordingly.

The code snippet for determining the data type as String is mentioned below −

```
socket.onmessage=function(event){

if(typeOfevent.data===String){
console.log(“Received data string”);
}
}
```

JSON (JavaScript Object Notation)

It is a lightweight format for transferring human-readable data between the computers. The structure of JSON consists of key-value pairs.

Example

```
{
  name:“JamesDevilson”,
  message:“HelloWorld!”
}
```

The following code shows how to handle a JSON object and extract its properties −

```
socket.onmessage=function(event){
if(typeOfevent.data===String){
//create a JSON object
varjsonObject=JSON.parse(event.data);
var username = jsonObject.name;
var message =jsonObject.message;

console.log(“Received data string”);
}
}
```

XML

Parsing in XML is not difficult, though the techniques differ from browser to browser. The best method is to parse using third party library like jQuery.

In both XML and JSON, the server responds as a string, which is being parsed at the client end.

ArrayBuffer

It consists of a structured binary data. The enclosed bits are given in an order so that the position can be easily tracked. ArrayBuffers are handy to store the image files.

Receiving data using ArrayBuffers is fairly simple. The operator **instanceOf** is used instead of equal operator.

The following code shows how to handle and receive an ArrayBuffer object −

```
socket.onmessage=function(event){
if(event.datainstanceofArrayBuffer){
var buffer =event.data;
console.log(“Receivedarraybuffer”);
}
}
```

Closing a Connection

Close event marks the end of a communication between the server and the client. Closing a connection is possible with the help of **onclose** event. After marking the end of communication with the help of **onclose** event, no messages can be further transferred between the server and the client. Closing the event can occur due to poor connectivity as well.

The **close()** method stands for **goodbye handshake**. It terminates the connection and no data can be exchanged unless the connection opens again.

Similar to the previous example, we call the **close()** method when the user clicks on the second button.

```
vartextView=document.getElementById("text-view");
varbuttonStop=document.getElementById("stop-button");

buttonStop.onclick=function(){
// Close the connection, if open.
if(socket.readyState===WebSocket.OPEN){
socket.close();
}
}
```

It is also possible to pass the code and reason parameters we mentioned earlier as shown below.

```
socket.close(1000, "Deliberate disconnection");
```