Internet socket

In computer networking, an Internet socket or network socket is an endpoint of a bidirectional inter-processcommunication flow across an Internet Protocol-based computer network, such as the Internet.

The term *Internet sockets* is also used as a name for an application programming interface (API) for the TCP/IP protocol stack, usually provided by the operating system. Internet sockets constitute a mechanism for delivering incoming data packets to the appropriate application process or thread, based on a combination of local and remote IP addresses and port numbers. Each socket is mapped by the operating system to a communicating application process or thread.

A socket address is the combination of an IP address (the location of the computer) and a port (which is mapped to the application program process) into a single identity, much like one end of a telephone connection is the combination of a phone number and a particular extension.

An Internet socket is characterized by a unique combination of the following:

- Local socket address: Local IP address and port number
- Remote socket address: Only for established TCP sockets. As discussed in the Client-Server section below, this is necessary since a TCP server may serve several clients concurrently. The server creates one socket for each client, and these sockets share the same local socket address.
- Protocol: A transport protocol (e.g., TCP, UDP), raw IP, or others. TCP port 53 and UDP port 53 are consequently different, distinct sockets.

Within the operating system and the application that created a socket, the socket is referred to by a unique integer number called *socket identifier* or *socket number*. The operating system forwards the payload of incoming IP packets to the corresponding application by extracting the socket address information from the IP and transport protocol headers and stripping the headers from the application data.

In IETFRequest for Comments, Internet Standards, in many textbooks, as well as in this article, the term *socket* refers to an entity that is uniquely identified by the socket number. In other textbooks[1], the socket term refers to a local socket address, i.e. a "combination of an IP address and a port number". In the original definition of *socket* given in RFC 147, as it was related to the ARPA network in 1971, *"the socket is specified as a 32 bit number with even sockets identifying*

receiving sockets and odd sockets identifying sending sockets." Today, however, socket communications are bidirectional.

On Unix-like and Microsoft Windows based operating systems the netstat command line tool may be used to list all currently established sockets and related information.

Socket types

There are several Internet socket types available:

- Datagram sockets, also known as connectionless sockets, which use User Datagram Protocol (UDP)
- Stream sockets, also known as connection-oriented sockets, which use Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP).
- Raw sockets (or *Raw IP sockets*), typically available in routers and other network equipment. Here the transport layer is bypassed, and the packet headers are not stripped off, but are accessible to the application. Application examples are Internet Control Message Protocol (ICMP, best known for the Ping suboperation), Internet Group Management Protocol (IGMP), and Open Shortest Path First (OSPF).[2]

There are also non-Internet sockets, implemented over other transport protocols, such as Systems Network Architecture (SNA).[3] See also Unix domain sockets (UDS), for internal inter-process communication.

Socket states and the client-server model

Computer processes that provide application services are called servers, and create sockets on start up that are in *listening state*. These sockets are waiting for initiatives from client programs. For a listening TCP socket, the remote address presented by netstat may be denoted 0.0.0.0 and the remote port number 0.

A TCP server may serve several clients concurrently, by creating a child process for each client and establishing a TCP connection between the child process and the client. Unique *dedicated sockets* are created for each connection. These are in *established* state, when a socket-to-socket virtual connection or virtual

circuit (VC), also known as a TCP session, is established with the remote socket, providing a duplex byte stream.

Other possible TCP socket states presented by the netstat command are Syn-sent, Syn-Recv, Fin-wait1, Fin-wait2, Time-wait, Close-wait and Closed which relate to various start up and shutdown steps.[4]

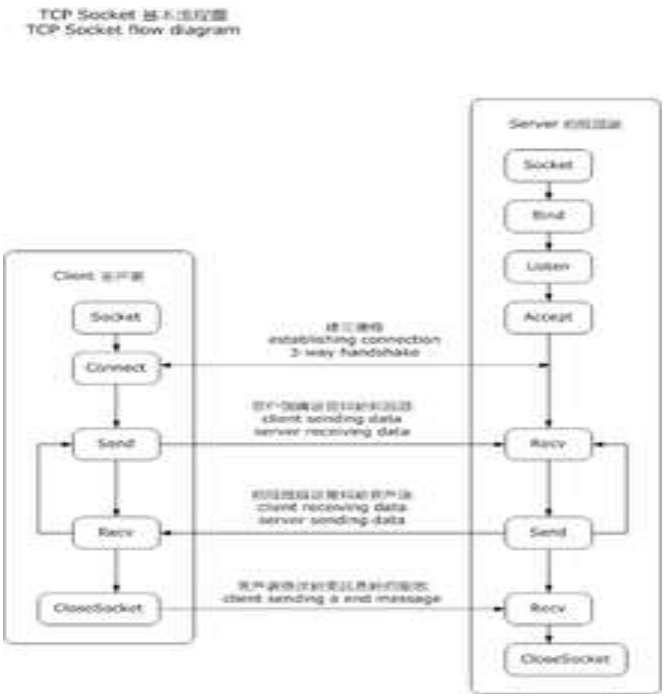
A server may create several concurrently established TCP sockets with the same local port number and local IP address, each mapped to its own server-child process, serving its own client process. They are treated as different sockets by the operating system, since the remote socket address (the client IP address and/or port number) are different; i.e. since they have different socket pair tuples (see below).

A UDP socket cannot be in an established state, since UDP is connectionless. Therefore, netstat does not show the state of a UDP socket. A UDP server does not create new child processes for every concurrently served client, but the same process handles incoming data packets from all remote clients sequentially through the same socket. This implies that UDP sockets are not identified by the remote address, but only by the local address, although each message has an associated remote address.

Socket pairs

Communicating local and remote sockets are called socket pairs. Each socket pair is described by a unique 4-tuplestruct consisting of source and destination IP addresses and port numbers, i.e. of local and remote socket addresses.[5][6] As seen in the discussion above, in the TCP case, each unique socket pair 4-tuple is assigned a socket number, while in the UDP case, each unique local socket address is assigned a socket number.

Implementation issues



TCP Socket flow diagram.

Sockets are usually implemented by an API library such as Berkeley sockets, first introduced in 1983. Most implementations are based on Berkeley sockets, for example Winsock introduced in 1991. Other socket API implementations exist, such as the STREAMS-based Transport Layer Interface (TLI).

Development of application programs that utilize this API is called socket programming or network programming.

These are examples of functions or methods typically provided by the API library:

- socket() creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
- bind() is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.
- listen() is used on the server side, and causes a bound TCP socket to enter listening state.
- connect() is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.

- accept() is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
- send() and recv(), or write() and read(), or recvfrom() and sendto(), are used for sending and receiving data to/from a remote socket.
- close() causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
- gethostbyname() and gethostbyaddr() are used to resolve host names and addresses.
- select() is used to prune a provided list of sockets for those that are ready to read, ready to write or have errors
- poll() is used to check on the state of a socket. The socket can be tested to see if it can be written to, read from or has errors.

Sockets in network equipment

The socket is primarily a concept used in the Transport Layer of the Internet model. Networking equipment such as routers and switches do not require implementations of the Transport Layer, as they operate on the Link Layer level (switches) or at the Internet Layer (routers). However, stateful network firewalls, network address translators, and proxy servers keep track of active socket pairs and require socket interfaces. Also in fair queuing, layer 3 switching and quality of service (QoS) support in routers, packet flows may be identified by extracting information about the socket pairs.

Raw sockets are typically available in network equipment, and used for routing protocols such as IGMP and OSPF, and in Internet Control Message Protocol (ICMP).

Sockets
A socket is made up of 3 identifyingproperties: Protocol Family, IP Address, Port Number
For TCP/IP Sockets:
 The protocol family is AF_INET (Address Family Internet)
 The IP Address identifies a host/service machine on the network
 Port defines the Service on the machine we’re communicating to/from

TCP Ports

The port numbers from 0 to 255 are well-known ports, and the use of these port numbersin your application is highly discouraged. Many well-known services you use haveassigned port numbers in this range.Service Name Port Numberftp 21telenet 23www-http 80irc 194.
In recent years the range for assigned ports managed by IANA (Internet AssignedNumbers Authority) has been expanded to the range of 0 – 1023. To get the most recentlisting of assigned port numbers, you can view the latest RFC 1700 at:http://www.faqs.org/rfcs/rfc1700.html.
In order for 2 machines to be able to communicate they need to be using the same type ofsockets, both have to be TCP or UDP. On Windows, the socket definition is defined inthe header file <winsock.h> or <winsock2.h>. Our program will be using Winsock 2.2 sowe will need to include <winsock2.h> and link with WS2_32.libOpening and closing a socket
To create a TCP/IP socket, we use the socket( ) API.SOCKET hSock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);If the socket API fails, it will return a value of INVALID_SOCKET, otherwise it willreturn a descriptor value that we will need to use when referring to this socket. Once weare done with the socket we must remember to call the closesocket( ) API.closesocket(hSock);
Note: On Unix/Linux the return type for socket( ) is an int, while the API to close thesocket is close(socket_discriptor);Before we can begin to use any socket API in Windows we need to initialize the socketlibrary, this is done by making a call to WSAStartup( ). This step is not required onUnix/Linux.
Initializing Winsock

```
// Initialize WinSock2.2 DLL
// low-word = major, hi-word = minor
WSADATA wsaData = {0};
WORD wVer = MAKEWORD(2,2);
int nRet = WSAStartup( wVer, &wsaData );
```

Before we exit our program we need to release the Winsock DLL with the following call.

```
// Release WinSock DLL
WSACleanup();
```

With the basics out of the way, the process of preparing the client and server application
is similar, but differ slightly in their steps. We will talk about how to write the server
code first.
Socket Address Structure for IPv4 the socket structure is defined as:

```
struct sockaddr
{
```

```
u_short sa_family; /* address family */
char sa_data[14]; /* up to 14 bytes of direct address */
};
```

This is the generic structure that most socket APIs accept, but the structure you will work
with is sockaddr_in (socket address internet).

```
struct sockaddr_in
{
short sin_family;
u_short sin_port;
struct in_addr sin_addr;
char sin_zero[8];
};
```

Note: sockaddr_in and the more generic sockaddr struct are the same size. Padding is
used to maintain the size by sin_zero. You will need to typecast between the two in your program.

struct in_addr found inside sockaddr_in is a union defined as:

```
struct in_addr
{
union
{
struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
struct { u_short s_w1,s_w2; } S_un_w;
u_long S_addr;
} S_un;
#define s_addr S_un.S_addr
#define s_host S_un.S_un_b.s_b2
#define s_net S_un.S_un_b.s_b1
#define s_imp S_un.S_un_w.s_w2
#define s_impno S_un.S_un_b.s_b4
#define s_lh S_un.S_un_b.s_b3
};
```

This structure holds the IP address which can be accessed in many ways. You can use theinet_addr( ) API to convert a IP dotted-numerical string into a 32bit IP address andassign it to the s_addr member. The use of this API is shown when we discuss the client code. To do an opposite conversion from network to IP string, use the inet_ntoa( ) API.

TCP Server

The steps to get a server up and running are shown below (read from top to bottom). This
is how our sample code is written, so it's a good idea to get familiar with the process.

```
socket( )
bind( )
+---->listen( )
| accept( )
| (block until connection from client )
| read( )
| write( )
+-----close( )
close( )
```

1. Create a server socket
2. Name the socket
3. Prepare the socket to listen
4. Wait for a request to connect, a new client socket is created here
5. Read data sent from client
6. Send data back to client
7. Close client socket
8. Loop back if not told to exit
9. Close server socket is exit command given by client

Naming the socket

When we prepare the socket for the server, we use INADDR_ANY for the IP address totell the TCP stack to assign an IP address to listen on for incoming connection requests.Do not assign a hard-coded value, otherwise the program will only run on the serverdefined by the IP address, and if the server is multi-homed then we are restricting it tolisten on only one IP address rather than allow the administrator to set the default IPaddress.

```
// name socket
sockaddr_in saListen = {0};
saListen.sin_family = PF_INET;
saListen.sin_port = htons( 10000 );
saListen.sin_addr.s_addr = htonl( INADDR_ANY );
```

Once we have initialize the sockaddr_in struct, we need to name the socket by callingthe bind( ) API.

```
bind( hSock, (sockaddr*)&saListen, sizeof(sockaddr) );
```

Notice how we are required to typecast the socket address to (sockaddr*). Here hSock isthe server socket we created earlier and this binds (names) the socket with theinformation provided in the socket structure. After the socket has been named, we thenwait for incoming connection requests by making the next 2 calls.

```
listen( hSock, 5 );
sockaddr_in saClient = {0};
int nSALen = sizeof( sockaddr );
SOCKET hClient = accept( hSock, (sockaddr*)&saClient, &nSALen );
```

The values of '5' passed to listen( ) is the backlog of connection request that will getqueued waiting to be processed. Once this limit is exceeded, all new calls will fail. In thecall to accept( ), we do not need to initialize saClient, the sockaddr struct because whenthis calls returns it will fill saClient with the name of the client's socket.The sample echo server provided is a single thread (iterative) application, it can onlyhandle and process one connection at a time. In a real world application, one wouldgeneral use multithreading to handle incoming client requests. The primary thread wouldlisten to incoming calls and block on the call to accept( ). When a connection requestcame in, the main thread would be given the client's socket descriptor. At this point anew thread should be created and the client socket passed to it for processing the request.The main thread would then loop back and listen for the next connection.