


SOCKETS - SERVER & CLIENT - 2018



(<http://www.addthis.com/bookmark.php?v=250&username=khong7>)

K Hong
google.com/+KHongSanF
Francisc...

 Follow

4,583 followers

Ph.D. / Golden Gate Ave, San Francisco / Seoul National Univ / Carnegie Mellon / UC Berkeley / DevOps / Deep Learning / Visualization

Sponsor Open Source development activities and free contents for everyone.



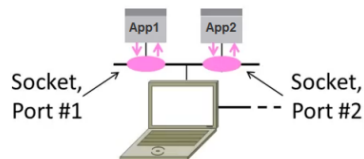
Thank you.

- K Hong
(http://bogotobogo.com/about_us.php)

bogotobogo.com site search:

Socket

Sockets let apps attach to the local network at different **ports**



Server/Client Applications

The basic mechanisms of client-server setup are:

1. A client app send a request to a server app.
2. The server app returns a reply.
3. Some of the basic data communications between client and server are:
 1. File transfer - sends name and gets a file.
 2. Web page - sends url and gets a page.
 3. Echo - sends a message and gets it back.

Sponsor Open Source development activities and free contents for everyone.



Thank you.

- K Hong
(http://bogotobogo.com/about_us.php)

Server Socket

1. create a **socket** - Get the file descriptor!

- 2. **bind** to an address -What port am I on?
- 3. **listen** on a port, and wait for a connection to be established.
- 4. **accept** the connection from a client.
- 5. **send/recv** - the same way we read and write for a file.
- 6. **shutdown** to end read/write.
- 7. **close** to releases data.

Client Socket

- 1. create a **socket**.
- 2. **bind*** - this is probably be unnecessary because you're the client, not the server.
- 3. **connect** to a server.
- 4. **send/recv** - repeat until we have or receive data
- 5. **shutdown** to end read/write.
- 6. **close** to releases data.

Socket and network programming links

For socket programming with **Boost.Asio**, please visit:

- 1. Boost.Asio - 1. Blocking and non-blocking wait with timers
(http://www.bogotobogo.com/cplusplus/Boost/boost_AsyncIOasio_tcpip_socket_server_client_timer_A.php)
- 2. Boost.Asio - 2. Binding arguments to a callback handler member function
(http://www.bogotobogo.com/cplusplus/Boost/boost_AsyncIOasio_tcpip_socket_server_client_timer_bind_handler_member_function_B...
- 3. Boost.Asio - 3. Multithreading, synchronizing, and handler Application (UI) -
(http://www.bogotobogo.com/cplusplus/Boost/boost_AsyncIOasio_tcpip_socket_server_client_timer_bind_handler_multithreading_sync...
- 4. Boost.Asio - 4. TCP Socket Programming
(http://www.bogotobogo.com/cplusplus/Boost/boost_AsyncIOasio_tcpip_socket_server_client_timer_bind_handler_multithreading_sync...

For socket programming with Qt, please visit

http://www.bogotobogo.com/cplusplus/sockets_server_client_QT.php
(/cplusplus/sockets_server_client_QT.php).

Qt 5 Tutorials:

- 1. Qt QHttp - Downloading Files
(/Qt/Qt5_QHttp_Downloading_Files_Network_Programming.php)
- 2. Qt 5 QNetworkAccessManager and QNetworkRequest -
Downloading Files
(/Qt/Qt5_Downloading_Files_QNetworkAccessManager_QNetworkRequest.php)
- 3. Qt 5 QTcpSocket (/Qt/Qt5_QTcpSocket.php)
- 4. Qt 5 QTcpSocket with Signals and Slots
(/Qt/Qt5_QTcpSocket_Signals_Slots.php)
- 5. Qt 5 QTcpServer - Client and Server
(/Qt/Qt5_QTcpServer_Client_Server.php)
- 6. Qt 5 QTcpServer - Client and Server using MultiThreading
(/Qt/Qt5_QTcpServer_Multithreaded_Client_Server.php)
- 7. Qt 5 QTcpServer - Client and Server using QThreadPool
(/Qt/Qt5_QTcpServer_QThreadPool_Multithreaded_Client_Server.php)

C++
Tutorials

C++ Home
(/cplusplus/cpptut.php)

Algorithms & Data Structures in C++
...
(/Algorithms/algorithms.php)
asio_tcpip_socket_server_client_timer_bind_handler_member_function_B
using Windows Forms (Visual Studio 2013/2012)
asio_tcpip_socket_server_client_timer_bind_handler_multithreading_sync
(/cplusplus/application_visual_studio_2013.php)

auto_ptr
(/cplusplus/autoptr.php)

Binary Tree
Example Code
(/cplusplus/binarytree.php)

Blackjack with Qt
(/cplusplus/blackjackQT.php)

Boost - shared_ptr,
weak_ptr, etc.
(/cplusplus/boost.php)

Boost.Asio (Socket Programming - Asynchronous TCP/IP)...
(/cplusplus/Boost/boost_AsyncIOasio_tcpip_socket_server_client_timer_...

Classes and Structs
(/cplusplus/class.php)

Constructor
(/cplusplus/constructor.php)

C++11(C++0x):
rvalue references,

Socket - summary

Here is the summary of key concepts:

1. Socket is a way of speaking to other programs using standard **file descriptors**.
2. Where do we get the file descriptor for network communication?
Well, we make a call to the **socket()** system routine. After the **socket()** returns the socket descriptor, we start communicate through it using the specialized **send()/recv()** socket API calls.
3. A TCP socket is an **endpoint instance**
4. A TCP socket is **not a connection**, it is the **endpoint** of a specific connection.
5. A TCP **connection** is defined by **two endpoints** aka sockets.
6. The purpose of **ports** is to **differentiate multiple endpoints** on a given network address.
7. The port numbers are encoded in the transport protocol packet header, and they can be readily interpreted not only by the sending and receiving computers, but also by other components of the networking infrastructure. In particular, firewalls are commonly configured to differentiate between packets based on their source or destination **port numbers** as in **port forwarding**.
8. It is the **socket pair** (the **4-tuple** consisting of the client IP address, client port number, server IP address, and server port number) that specifies the two endpoints that uniquely identifies each **TCP connection** in an internet.
9. Only **one process** may bind to a specific **IP address** and **port** combination using the **same transport protocol**. Otherwise, we'll have **port conflicts**, where multiple programs attempt to bind to the same port numbers on the same IP address using the same protocol.

To connect to another machine, we need a **socket** connection.

What's a connection?

A **relationship** between two machines, where **two pieces of software know about each other**. Those two pieces of software know how to communicate with each other. In other words, they know how to send **bits** to each other. A socket connection means the two machines have information about each other, including **network location (IP address)** and **TCP port**. (If we can use analogy, IP address is the **phone number** and the TCP port is the **extension**).

A socket is an object similar to a file that allows a program to accept incoming connections, make outgoing connections, and send and receive data. **Before two machines can communicate, both must create a socket object.**

A socket is a **resource** assigned to the server process. The server creates it using the system call **socket()**, and it can't be shared with other processes.

TCP vs UDP

There are several different types of socket that determine the structure of the transport layer. The most common types are **stream** sockets and **datagram** sockets.

move constructor,
and lambda, etc.
(/cplusplus/cplusplus11.php)

C++ API Testing
(/cplusplus/cpptesting.php)

C++ Keywords -
const, volatile, etc.
(/cplusplus/cplusplus_keywords.php)

Debugging Crash &
Memory Leak
(/cplusplus/CppCrashDebuggingMemoryLeak.php)

Design Patterns in
C++ ...
(/DesignPatterns/introduction.php)

Dynamic Cast
Operator
(/cplusplus/dynamic_cast.php)

Eclipse CDT / JNI
(Java Native
Interface) / MinGW
(/cplusplus/eclipse_CDT_JNI_MinGW_64bit.php)

Embedded
Systems
Programming I -
Introduction
(/cplusplus/embeddedSystemsProgramming.php)

Embedded
Systems
Programming II -
gcc ARM Toolchain
and Simple Code
on Ubuntu and
Fedora
(/cplusplus/embeddedSystemsProgramming_gnu_toolchain_ARM_cross_c

Embedded
Systems
Programming III -
Eclipse CDT Plugin
for gcc ARM
Toolchain
(/cplusplus/embeddedSystemsProgramming_GNU_ARM_ToolChain_Eclips

Exceptions
(/cplusplus/exceptions.php)

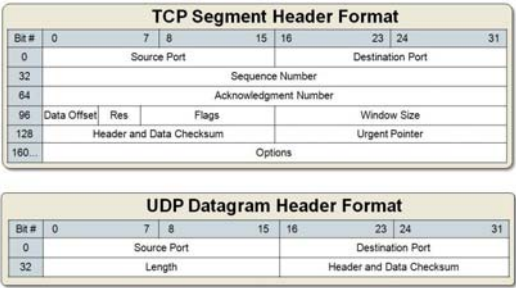
Friend Functions
and Friend Classes
(/cplusplus/friendclass.php)

fstream: input &
output
(/cplusplus/fstream_input_output.php)

Function
Overloading
(/cplusplus/function_overloading.php)

Functors (Function
Objects) I -
Introduction
(/cplusplus/functor_function_object_stl_intro.php)

Functors (Function
Objects) II -
Converting
function to functor
(/cplusplus/functor_function_object_stl_2.php)



Functors (Function Objects) - General
(/cplusplus/functors.php)

Git and GitHub
Express...
(/cplusplus/Git/Git_GitHub_Express.php)

GTest (Google Unit Test) with Visual Studio 2012
(/cplusplus/google_unit_test_gtest.php)

| | |
|--|---|
| TCP (Streams) | UDP (Datagrams) |
| Connections | Connectionless sockets We don't have to maintain an open connection as we do with stream sockets. We just build a packet, put an IP header on it with destination information, and send it out. No connection needed: datagram sockets also use IP for routing, but they don't use TCP *note: can be connect()'d if we really want. |
| SOCK_STREAM | SOCK_DGRAM |
| If we output two items into the socket in the order "A, B", they will arrive in the order "A, B" at the opposite end. They will also be error-free. | If we send a datagram, it may arrive. But it may arrive out of order. If it arrives, however, the data within the packet will be error-free. |
| | Why would we use an unreliable protocol? Speed! We just ignore the dropped packets. |
| Arbitrary length content | Limited message size |
| Flow control matches sender to receiver | Can send regardless of receiver state |
| Congestion control matches sender to network | Can send regardless of network state |
| http, telnet | ftp (trivial file transfer protocol), dhcpd (a DHCP client), multiplayer games, streaming audio, video conferencing *note: They use complementary protocol on top of UDP to get more reliability |

Inheritance & Virtual Inheritance (multiple inheritance)
(/cplusplus/multipleinheritance.php)

Libraries - Static, Shared (Dynamic)
(/cplusplus/libraries.php)

Linked List Basics
(/cplusplus/linked_list_basics.php)

Linked List Examples
(/cplusplus/linkedlist.php)

make & CMake
(/cplusplus/make.php)

make (gnu)
(/cplusplus/gnumake.php)

Memory Allocation
(/cplusplus/memoryallocation.php)

Multi-Threaded Programming - Terminology - Semaphore, Mutex, Priority Inversion etc.
(/cplusplus/multithreaded.php)

Multi-Threaded Programming II - Native Thread for Win32 (A)
(/cplusplus/multithreading_win32A.php)

1. Stream Sockets

Stream sockets provide **reliable two-way** communication similar to when we call someone on the phone. One side initiates the connection to the other, and after the connection is established, either side can communicate to the other.

In addition, there is immediate confirmation that what we said actually reached its destination.

Stream sockets use a **Transmission Control Protocol (TCP)**, which exists on the transport layer of the Open Systems Interconnection (OSI) model. The data is usually transmitted in packets. TCP is designed so that the packets of data will arrive without errors and in sequence.

Webservers, mail servers, and their respective client applications all use TCP and stream socket to communicate.

2. Datagram Sockets

Communicating with a datagram socket is more like mailing a letter than making a phone call. The connection is **one-**

Multi-Threaded Programming II - Native Thread for Win32 (B)
(/cplusplus/multithreading_win32B.php)

Multi-Threaded Programming II - Native Thread for Win32 (C)
(/cplusplus/multithreading_win32C.php)

Multi-Threaded Programming II - C++ Thread for Win32
(/cplusplus/multithreading_win32.php)

Multi-Threaded Programming III -

way only and **unreliable**.

If we mail several letters, we can't be sure that they arrive in the same order, or even that they reached their destination at all. Datagram sockets use **User Datagram Protocol (UDP)**. Actually, it's not a real connection, just a basic method for sending data from one point to another. Datagram sockets and UDP are commonly used in networked games and streaming media.

Though in this section, we mainly put focus on applications that maintain connections to their clients, using connection-oriented TCP, there are cases where the overhead of establishing and maintaining a socket connection is unnecessary.

For example, just to get the data, a process of creating a socket, making a connection, reading a single response, and closing the connection, is just too much. In this case, we use UDP.

Services provided by UDP are typically used where a client needs to make a short query of a server and expects a single short response. To access a service from UDP, we need to use the UDP specific system calls, **sendto()** and **recvfrom()** instead of **read()** and **write()** on the socket.

UDP is used by app that doesn't want reliability or bytestreams.

1. Voice-over-ip (unreliable) such as conference call. (visit VoIP (<http://www.bogotobogo.com/VideoStreaming/VoIP.php>))
2. DNS, RPC (message-oriented)
3. DHCP (bootstrapping)

C/C++ Class
Thread for
Pthreads
(/cplusplus/multithreading_pthread.php)

MultiThreading/Parallel
Programming - IPC
(/cplusplus/multithreading_ipc.php)

Multi-Threaded
Programming with
C++11 Part A (start,
join(), detach(), and
ownership)
(/cplusplus/multithreaded4_cplusplus11.php)

Multi-Threaded
Programming with
C++11 Part B
(Sharing Data -
mutex, and race
conditions, and
deadlock)
(/cplusplus/multithreaded4_cplusplus11B.php)

Multithread
Debugging
(</cplusplus/multithreadedDebugging.php>)

Object Returning
(/cplusplus/object_returning.php)

Object Slicing and
Virtual Table
(</cplusplus/slicing.php>)

OpenCV with C++
(</cplusplus/opencv.php>)

Operator
Overloading I
(</cplusplus/operatoroverloading.php>)

Operator
Overloading II -
self assignment
(/cplusplus/operator_oveading_self_assignment.php)

Pass by Value vs.
Pass by Reference
(</cplusplus/valuesreference.php>)

Pointers
(</cplusplus/pointers.php>)

Pointers II - void
pointers & arrays
(/cplusplus/pointers2_voidpointers_arrays.php)

Pointers III -
pointer to function
& multi-
dimensional arrays
(/cplusplus/pointers3_function_multidimensional_arrays.php)

Preprocessor -
Macro
(/cplusplus/preprocessor_macro.php)

Private Inheritance
(/cplusplus/private_inheritance.php)

Python & C++ with
SIP
(/python/python_cpp_sip.php)

Client/Server

The **client-server model** distinguishes between applications as well as devices. **Network clients make requests to a server by sending messages, and servers respond to their clients by acting on each request and returning results.**

For example, let's talk about **telnet**.

When we connect to a remote host on port 23 with telnet (the client), a program on that host (called **telnetd**, the server) springs to life. It handles the incoming telnet connection, sets us up with a login prompt, etc.

One server generally supports numerous clients, and multiple servers can be networked together in a pool to handle the increased processing load as the number of clients grows.

Some of the most popular applications on the Internet follow the client-server model including email, FTP and Web services. Each of these clients features a user interface and a client application that allows the user to connect to servers. In the case of email and FTP, users enter a computer name (or an IP address) into the interface to set up connections to the server.

The steps to establish a socket on the **server** side are:

1. Create a socket with the **socket()** system call.
2. The server process gives the socket a name. In linux file system, local sockets are given a filename, under /tmp or /usr/tmp directory. For network sockets, the filename will be a service identifier, port number, to which the clients can

make connection. This identifier allows to route incoming connections (which has that the port number) to connect server process. A socket is named using **bind()** system call.

3. The server process then waits for a client to connect to the named socket, which is basically listening for connections with the **listen()** system call. If there are more than one client are trying to make connections, the **listen()** system call make a queue.

The machine receiving the connection (the server) must bind its socket object to a known port number. A port is a 16-bit number in the range 0-65535 that's managed by the operating system and used by clients to uniquely identify servers. Ports 0-1023 are reserved by the system and used by common network protocols.

4. Accept a connection with the **accept()** system call. At **accept()**, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client. For TCP servers, the socket object used to receive connections is not the same socket used to perform subsequent communication with the client. In particular, the **accept()** system call returns a new socket object that's actually used for the connection. This allows a server to manage connections from a large number of clients simultaneously.
5. Send and receive data.
6. The named socket remains for further connections from other clients. A typical web server can take advantage of multiple connections. In other words, it can serve pages to many clients at once. But for a simple server, further clients wait on the listen queue until the server is ready again.

The steps to establish a socket on the **client** side are:

1. Create a socket with the **socket()** system call.
2. Connect the socket to the address of the server using the **connect()** system call.
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the **read()** and **write()** system calls.

Only needed for Streams

To/From forms for Datagrams

| Primitive | Meaning |
|---------------|--|
| SOCKET | Create a new communication endpoint |
| BIND | Associate a local address (port) with a socket |
| LISTEN | Announce willingness to accept connections |
| ACCEPT | Passively establish an incoming connection |
| CONNECT | Actively attempt to establish a connection |
| SEND(TO) | Send some data over the socket |
| RECEIVE(FROM) | Receive some data over the socket |
| CLOSE | Release the socket |

TCP communication

(Pseudo)-random numbers in C++
(/cplusplus/RandomNumbers.php)

References for Built-in Types
(/cplusplus/references.php)

Socket - Server & Client
(/cplusplus/sockets_server_client.php)

Socket - Server & Client with Qt (Asynchronous / Multithreading / ThreadPool etc.)
(/cplusplus/sockets_server_client_QT.php)

Stack Unwinding
(/cplusplus/stackunwinding.php)

Standard Template Library (STL) I - Vector & List
(/cplusplus/stl_vector_list.php)

Standard Template Library (STL) II - Maps
(/cplusplus/stl2_map.php)

Standard Template Library (STL) II - unordered_map
(/cplusplus/stl2_unorderd_map_cpp11_hash_table_hash_function.php)

Standard Template Library (STL) II - Sets
(/cplusplus/stl2B_set.php)

Standard Template Library (STL) III - Iterators
(/cplusplus/stl3_iterators.php)

Standard Template Library (STL) IV - Algorithms
(/cplusplus/stl4_algorithms.php)

Standard Template Library (STL) V - Function Objects
(/cplusplus/stl5_function_objects.php)

Static Variables and Static Class Members
(/cplusplus/statics.php)

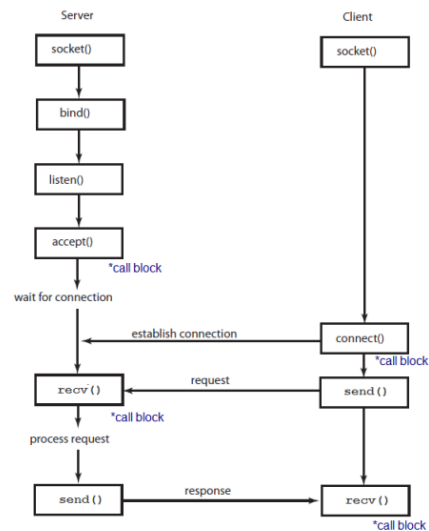
String
(/cplusplus/string.php)

String II - stringstream etc.
(/cplusplus/string2.php)

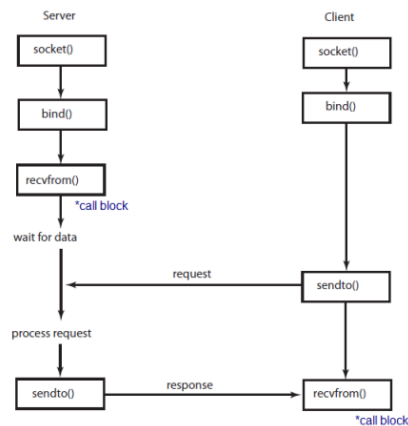
Taste of Assembly
(/cplusplus/assembly.php)

Templates
(/cplusplus/templates.php)

Template



UDP communication - clients and servers don't establish a **connection** with each other



*call block, go to Blocking socket vs non-blocking socket .

Specialization
(/cplusplus/template_specialization_function_class.php)

Template
Specialization -
Traits
(/cplusplus/template_specialization_traits.php)

Template
Implementation &
Compiler (.h or
.cpp?)
(/cplusplus/template_declaration_definition_header_implementation_file.php)

The this Pointer
(/cplusplus/this_pointer.php)

Type Cast
Operators
(/cplusplus/typecast.php)

Upcasting and
Downcasting
(/cplusplus/upcasting_downcasting.php)

Virtual Destructor
&
boost::shared_ptr
(/cplusplus/virtual_destructors_shared_ptr.php)

Virtual Functions
(/cplusplus/virtualfunctions.php)

*Programming
Questions and
Solutions ↓*

Strings and Arrays
(/cplusplus/quiz_strings_arrays.php)

Linked List
(/cplusplus/quiz_linkedlist.php)

Recursion
(/cplusplus/quiz_recursion.php)

Bit Manipulation
(/cplusplus/quiz_bit_manipulation.php)

Small Programs
(string, memory
functions etc.)
(/cplusplus/smallprograms.php)

Math & Probability
(/cplusplus/quiz_math_probability.php)

Multithreading
(/cplusplus/quiz_multithreading.php)

140 Questions by
Google
(/cplusplus/google_interview_questions.php)

Qt 5 EXPRESS...
(/Qt/Qt5_Creating_QtQuick2_QML_Application_Animation_A.php)

Win32 DLL ...
(/Win32API/Win32API_DLL.php)

Articles On C++
(/cplusplus/cppNews.php)

Socket Functions

Sockets, in C, behaves like files because they use file descriptors to identify themselves. Sockets behave so much like files that we can use the **read()** and **write()** to receive and send data using **socket file descriptors**.

There are several functions, however, specifically designed to handle sockets. These functions have their prototypes defined in **/usr/include/sys/sockets.h**.

1. **int socket(int domain, int type, int protocol)**

Used to create a new socket, returns a file descriptor for the socket or -1 on error.

It takes three parameters:

1. domain: the protocol family of socket being requested
2. type: the type of socket within that family
3. and the protocol.

The parameters allow us to say what kind of socket we want (IPv4/IPv6, stream/datagram(TCP/UDP)).

1. The protocol family should be **AF_INET** or **AF_INET6**
2. and the protocol type for these two families is either **SOCK_STREAM** for TCP/IP or **SOCK_DGRAM** for UDP/IP.
3. The protocol should usually be set to zero to indicate that the default protocol should be used.

What's new in

C++11...

(/cplusplus/C11/C11_initializer_list.php)

C++11 Threads

EXPRESS...

(/cplusplus/C11/1_C11_creating_thread.php)

OpenCV...

(/OpenCV/opencv_3_tutorial_imgproc_gaussian_median_blur_bilateral_filt

2. `int bind(int fd, struct sockaddr *local_addr, socklen_t addr_length)`

Once we have a socket, we might have to associate that socket with a port on our local machine.

The port number is used by the kernel to match an incoming packet to a certain process's socket descriptor.

A server will call **bind()** with the address of the local host and the port on which it will listen for connections.

It takes file descriptor (previously established socket), a pointer to (the address of) a structure containing the details of the address to bind to, the value **INADDR_ANY** is typically used for this, and the length of the address structure.

The particular structure that needs to be used will depend on the protocol, which is why it is passed by the pointer.

So, this **bind()** call will bind the socket to the current IP address on port, portno

Returns 0 on success and -1 on error.

3. `int listen(int fd, int backlog_queue_size)`

Once a server has been bound to an address, the server can then call **listen()** on the socket.

The parameters to this call are the socket (fd) and the maximum number of queued connections requests up to **backlog_queue_size**.

Returns 0 on success and -1 on error.

4. `int accept(int fd, struct sockaddr *remote_host, socklen_t addr_length)`

Accepts an incoming connection on a bound socket. The address information from the remote host is written into the **remote_host** structure and the actual size of the address structure is written into ***addr_length**.

In other words, this **accept()** function will write the connecting client's address info into the address structure. Then, returns a new socket file descriptor for the accepted connection.

So, the original socket file descriptor can continue to be used for accepting new connections while the new socket file descriptor is used for communicating with the connected client.

This function returns a new socket file descriptor to identify the connected socket or -1 on error.

Here is the description from the man page:

"It extracts the first connection request on the queue of pending connections for the listening socket, **sockfd**, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call".

If no pending connections are present on the queue, and the socket is not marked as nonblocking, `accept()` blocks the caller until a connection is present.

5. `int connect(int fd, struct sockaddr *remote_host, socklen_t addr_length)`

Connects a socket (described by file descriptor **fd**) to a remote host.

Returns 0 on success and -1 on error.

This is a blocking call. That's because when we issue a call to `connect()`, our program doesn't regain control until either the connection is made, or an error occurs. For example, let's say that we're writing a web browser. We try to connect to a web server, but the server isn't responding. So, we now want the `connect()` API to stop trying to connect by clicking a stop button. But that can't be done. It waits for a return which could be 0 on success or -1 on error.

6. `int send(int fd, void *buffer, size_t n, int flags)`

Sends `n` bytes from ***buffer** to **socket fd**.

Returns the number of bytes sent or -1 on error.

7. `int receive(int fd, void *buffer, size_t n, int flags)`

Receives `n` bytes from **socket fd** into ***buffer**.

Returns the number of bytes received or -1 on error.

This is another blocking call. In other words, when we call **recv()** to read from a stream, control isn't returned to our program until at least one byte of data is read from the remote site. This process of waiting for data to appear is referred to as **blocking**. The same is true for the `write()` and the `connect()` APIs, etc. When we run those blocking APIs, the connection "blocks" until the operation is complete.

The following server code listens for TCP connections on port 20001. When a client connects, it sends the message "Hello world!", and then it receives data from client.

server.c

```

/* The port number is passed as an argument */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno;
    socklen_t clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;
    if (argc < 2) {
        fprintf(stderr,"ERROR, no port provided\n");
        exit(1);
    }
    // create a socket
    // socket(int domain, int type, int protocol)
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");

    // clear address structure
    bzero((char *) &serv_addr, sizeof(serv_addr));

    portno = atoi(argv[1]);

    /* setup the host_addr structure for use in bind call
    // server byte order
    serv_addr.sin_family = AF_INET;

    // automatically be filled with current host's IP address
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    // convert short integer value for port must be converted
    serv_addr.sin_port = htons(portno);

    // bind(int fd, struct sockaddr *local_addr, socklen_t
    // bind() passes file descriptor, the address structure
    // and the length of the address structure
    // This bind() call will bind the socket to the current
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
        error("ERROR on binding");

    // This listen() call tells the socket to listen to
    // The listen() function places all incoming connections
    // until accept() call accepts the connection.
    // Here, we set the maximum size for the backlog queue
    listen(sockfd,5);

    // The accept() call actually accepts an incoming connection
    clilen = sizeof(cli_addr);

    // This accept() function will write the connecting
    // into the the address structure and the size of the
    // The accept() returns a new socket file descriptor
    // So, the original socket file descriptor can continue
    // for accepting new connections while the new socket
    // communicating with the connected client.
    newsockfd = accept(sockfd,
        (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");

    printf("server: got connection from %s port %d\n",
        inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port));

    // This send() function sends the 13 bytes of the string
    send(newsockfd, "Hello, world!\n", 13, 0);

    bzero(buffer,256);

    n = read(newsockfd,buffer,255);
    if (n < 0) error("ERROR reading from socket");
    printf("Here is the message: %s\n",buffer);

    close(newsockfd);

```

```

        close(sockfd);
        return 0;
    }

```

When a socket is created with the **socket()** function, the **domain**, **type**, and **protocol** of the socket must be specified. The domain refers to the protocol family of the socket.

```

/* Address families. */
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL PF_LOCAL
#define AF_UNIX PF_UNIX
#define AF_FILE PF_FILE
#define AF_INET PF_INET
#define AF_AX25 PF_AX25
#define AF_IPX PF_IPX
#define AF_APPLETALK PF_APPLETALK
#define AF_NETROM PF_NETROM
#define AF_BRIDGE PF_BRIDGE
#define AF_ATMPVC PF_ATMPVC
#define AF_X25 PF_X25
#define AF_INET6 PF_INET6
#define AF_ROSE PF_ROSE
#define AF_DECnet PF_DECnet
#define AF_NETBEUI PF_NETBEUI
#define AF_SECURITY PF_SECURITY
#define AF_KEY PF_KEY
#define AF_NETLINK PF_NETLINK
#define AF_ROUTE PF_ROUTE
#define AF_PACKET PF_PACKET
#define AF_ASH PF_ASH
#define AF_ECONET PF_ECONET
#define AF_ATMSVC PF_ATMSVC
#define AF_SNA PF_SNA
#define AF_IRDA PF_IRDA
#define AF_PPPOX PF_PPPOX
#define AF_WANPIPE PF_WANPIPE
#define AF_BLUETOOTH PF_BLUETOOTH
#define AF_MAX PF_MAX

```

A socket can be used to communicate using a variety of protocols, from the standard Internet protocol used when we browse the Web. These families are defined in **bits/socket.h**, which is automatically included from **sys/socket.h**.

There are several types of sockets: stream sockets and datagram sockets are the most commonly used. The types of sockets are also defined in **/usr/include/bits/socket.h**

```

/* Types of sockets. */
enum __socket_type
{
    SOCK_STREAM = 1, /* Sequenced, reliable, c
                     byte streams. */
#define SOCK_STREAM SOCK_STREAM
    SOCK_DGRAM = 2, /* Connectionless, unrel
                     of fixed maximum lengt
#define SOCK_DGRAM SOCK_DGRAM
    SOCK_RAW = 3, /* Raw protocol interface
#define SOCK_RAW SOCK_RAW
    SOCK_RDM = 4, /* Reliably-delivered mes
#define SOCK_RDM SOCK_RDM
    SOCK_SEQPACKET = 5, /* Sequenced, reliable, c
                        datagrams of fixed max
#define SOCK_SEQPACKET SOCK_SEQPACKET
    SOCK_PACKET = 10 /* Linux specific way of
                     at the dev level. For
                     other similar things o
#define SOCK_PACKET SOCK_PACKET
};

```

The 3rd argument for the **socket()** function is the protocol, which should always be 0. The specification allows for multiple protocols within a protocol family, so this argument is used to select on of the protocols from the family.

/usr/include/bits/socket.h

```

/* Protocol families. */
#define PF_UNSPEC 0 /* Unspecified. */
#define PF_LOCAL 1 /* Local to host (pipes a
#define PF_UNIX PF_LOCAL /* Old BSD name for PF_L
#define PF_FILE PF_LOCAL /* Another non-standard
#define PF_INET 2 /* IP protocol family. */
#define PF_AX25 3 /* Amateur Radio AX.25.
#define PF_IPX 4 /* Novell Internet Protoc
#define PF_APPLETALK 5 /* Appletalk DDP. */
#define PF_NETROM 6 /* Amateur radio NetROM.
#define PF_BRIDGE 7 /* Multiprotocol bridge.
#define PF_ATMPVC 8 /* ATM PVCs. */
#define PF_X25 9 /* Reserved for X.25 proj
#define PF_INET6 10 /* IP version 6. */
#define PF_ROSE 11 /* Amateur Radio X.25 PLP
#define PF_DECnet 12 /* Reserved for DECnet pr
#define PF_NETBEUI 13 /* Reserved for 802.2LLC
#define PF_SECURITY 14 /* Security callback pseu
#define PF_KEY 15 /* PF_KEY key management
#define PF_NETLINK 16
#define PF_ROUTE PF_NETLINK /* Alias to emulate 4.
#define PF_PACKET 17 /* Packet family. */
#define PF_ASH 18 /* Ash. */
#define PF_ECONET 19 /* Acorn Econet. */
#define PF_ATMSVC 20 /* ATM SVCs. */
#define PF_SNA 22 /* Linux SNA Project */
#define PF_IRDA 23 /* IRDA sockets. */
#define PF_PPPOX 24 /* PPPoX sockets. */
#define PF_WANPIPE 25 /* Wanpipe API sockets.

```

However, in practice, most protocol families only have one protocol, which means this should usually be set for 0; the first and only protocol in the enumeration of the family.

```

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

```

1. **sin_family** = specifies the address family, usually the constant **AF_INET**
2. **sin_addr** = holds the IP address returned by **inet_addr()** to be used in the socket connection.
3. **sin_port** = specifies the port number and must be used with **htons()** function that converts the **host byte order** to **network byte order** so that it can be transmitted and routed properly when opening the socket connection. The reason for this is that computers and network protocols order their bytes in a non-compatible fashion.

The lines above set up the **serv_addr** structure for use in the **bind** call.

```

/* Structure describing a generic socket address. */
struct sockaddr
{
    __SOCKADDR_COMMON (sa_); /* Common data: address f
    char sa_data[14]; /* Address data. */
};

```

The address family is **AF_INET**, since we are using **IPv4** and the **sockaddr_in** structure. The short integer value for port must be converted into network byte order, so the **htons()** (Host-to-Network Short) function is used.

The **bind()** call passes the socket file descriptor, the address structure, and the length of the address structure. This call will bind the socket to the current IP address on port 20001.

```
if (bind(sockfd, (struct sockaddr *) &serv_addr,
    sizeof(serv_addr)) < 0) error("ERROR on bindin
```

The **listen()** call tells the socket to listen for incoming connections, and a subsequent **accept()** call actually accepts an incoming connection. The **listen()** function places all incoming connections into a backlog queue until an **accept()** call accepts the connections. The last argument to the **listen()** call sets the maximum size for the backlog queue.

```
listen(sockfd,5);
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
if (newsockfd < 0) error("ERROR on accept");
```

The final argument of the **accept()** is a pointer to the size of the address structure. This is because the **accept()** function will write the connecting client's address information into the address structure and the size of that structure is **clilen**. The **accept()** function returns a new socket file descriptor for the accepted connection:

```
newsockfd = accept(sockfd,
    (struct sockaddr *) &cli_addr,&clilen);
```

This way, the original socket file descriptor can continue to be used for accepting new connections, while the new socket file descriptor is used for communicating with the connected client.

The **send()** function sends the 13 bytes of the string **Hello, world\n** to the new socket that describes the new connection.

```
send(newsockfd, "Hello, world!\n", 13,0);
```

To compile, the server.c:

```
gcc -o server server.c
```

and to run

```
./server port#
```

client.c

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
        fprintf(stderr, "usage %s hostname port\n", argv[0]);
        exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
        (char *)&serv_addr.sin_addr.s_addr,
        server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd, (struct sockaddr *) &serv_addr,
        error("ERROR connecting");
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd, buffer, strlen(buffer));
    if (n < 0)
        error("ERROR writing to socket");
    bzero(buffer,256);
    n = read(sockfd, buffer, 255);
    if (n < 0)
        error("ERROR reading from socket");
    printf("%s\n", buffer);
    close(sockfd);
    return 0;
}

```

To compile

```
gcc -o client client.c
```

and to run

```
./client hostname port#
```

First, we run **server.c** as in

```
$ ./server 20001
```

Then, on **client side**

```
$ ./client myhostname 20001
Please enter the message:
```

Then, **server side** has the following message when connected successfully:

```
$ ./server 20001
server: got connection from 127.0.0.1 port 47173
```

Then, on **client side**

```
$ ./client myhostname 20001
Please enter the message: Hello from client
```

Then, **server side** has the following message:

```
$ ./server 20001
server: got connection from 127.0.0.1 port 47173
Here is the message: Hello from Client
```

Client side gets message (Hello, world!) from the server:

```
$ ./client myhostname 20001
Please enter the message: Hello from Client
Hello, world!
```

To run this codes, we don't need two machines. One is enough!

Sharing between processes - Sockets

Sharing between processes - Sockets using PThreads

(http://www.bogotobogo.com/cplusplus/multithreading_ipc.php)

Network Byte Order

The port number and IP address used in the **AF_INET** socket address structure are expected to follow the network byte ordering (big-endian). This is the opposite of x86's little-endian byte ordering, so these values must be converted. There are specialized functions for the conversions, and they are defined in **netinet.h** and **arpa/inet.h**.

"Basically, we want to convert the numbers to Network Byte Order before they go out on the wire, and convert them to Host Byte Order as they come in off the wire."

1. **htonl (long value) Host-to-Network Long**

Converts a 32-bit integer from the host's byte order to network byte order.

2. **htons (short value) Host-to-Network Short**

Converts a 16-bit integer from the host's byte order to network byte order.

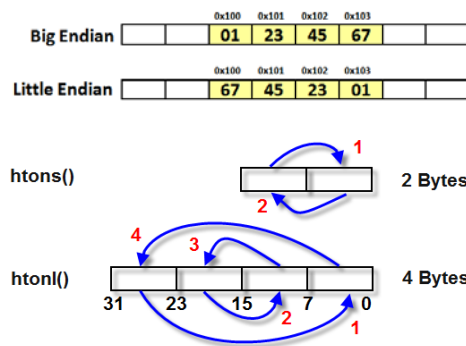
3. **ntohl (long value) Network-to-Host Long**

Converts a 32-bit integer from network byte order to the host's byte order.

4. **ntohs (short value) Network-to-Host Short**

Converts a 16-bit integer from network byte order to the host's byte order.

5. For C code for the conversions, please visit



Little Endian/Big Endian & TCP Sockets

(http://www.bogotobogo.com/Embedded/Little_endian_big_endian_htons_htonl.php).

Internet Address Conversion

1. **inet_aton (char *ascii_addr, struct in_addr *network_addr) ASCII-to-Network**

Converts an ASCII string containing IP address in dotted-number format into an **in_addr**, which only contains a 32-bit integer representing the IP address in network byte order.

2. **inet_ntoa (struct in_addr *network_addr) Network to ASCII**

It is passed a pointer to an **in_addr** structure containing an IP address, and the function returns a character pointer to an ASCII string containing the IP address in dotted-number format. This string is held in a statically allocated memory buffer in the function, so it can be accessed until the next call to **inet_ntoa()**, and the string will be overwritten.

TCP Connection establishment: Three-way-handshake

To establish a connection, TCP uses a **three-way handshake**. Before a client attempts to connect with a server, the server must first bind to and listen at a port to open it up for connections: this is called a **passive open**. Once the passive open is established, a client may initiate an **active open**.

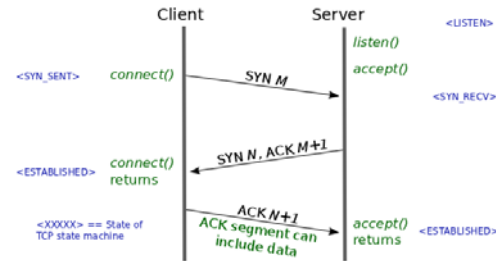
In other words, before communication begins, TCP establishes a new connection using a three-way handshake. This is because

1. Both sender and receiver must be ready before data transport starts, and they need to agree on set of

- parameters such as the Maximum Segment Size (MSS).
2. TCP end points maintain state about communications in both directions, and the handshake allows the state to be created and initialized.
 3. TCP establishes a stream of bytes in both directions, and the three-way handshake allows both streams to be established and acknowledged.

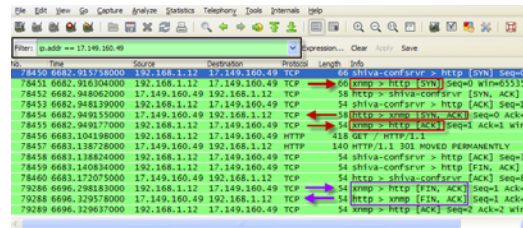
To establish a connection, the **three-way handshake** occurs:

1. **SYN**: The active open is performed by the client sending a SYN to the server.
2. **SYN-ACK**: In response, the server replies with a SYN-ACK.
3. **ACK**: Finally, the client sends an ACK back to the server.



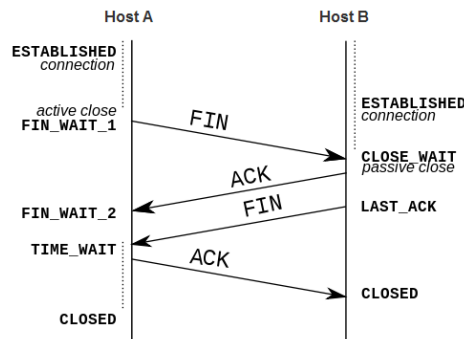
The picture above: from TCP Fast Open: expediting web services (<http://lwn.net/Articles/508865/>)

Here is the sample using Wireshark when I requested a page from apple.com, and then closed it:



It is also possible to terminate the connection by a 3-way handshake, more strictly it's a 2 (FIN/ACK) x 2 (FIN/ACK) handshake:

1. host A sends a **FIN**
2. host B replies with a **ACK (with data) + FIN**
3. host A replies with an **ACK**



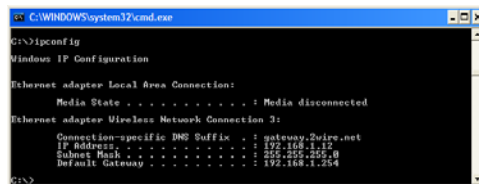
picture from wiki (http://en.wikipedia.org/wiki/Transmission_Control_Protocol)

Basic Network command

1. ipconfig (internet protocol configuration)

A console application that displays all current TCP/IP network configuration values.

In other words, it is commonly used to identify the addresses information of a computer on a network. It can show the physical address as well as the IP address. It can modify Dynamic Host Configuration Protocol DHCP and Domain Name System DNS settings.



```

C:\WINDOWS\system32\cmd.exe
C:\>ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

    Media State . . . . . : Media disconnected

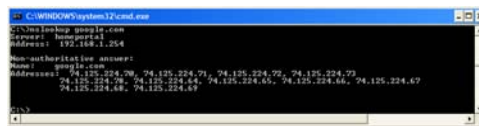
Ethernet adapter Wireless Network Connection 3:

    Connection-specific DNS Suffix  . : gateway.2wire.net
    IP Address. . . . . : 192.168.1.12
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.1.254

C:\>
  
```

2. nslookup

A network administration command-line tool available for many computer operating systems for querying the Domain Name System (DNS) to obtain domain name or IP address mapping or for any other specific DNS record.



```

C:\WINDOWS\system32\cmd.exe
C:\>nslookup google.com
Server: hqjwzj1.com
Address: 192.168.1.254

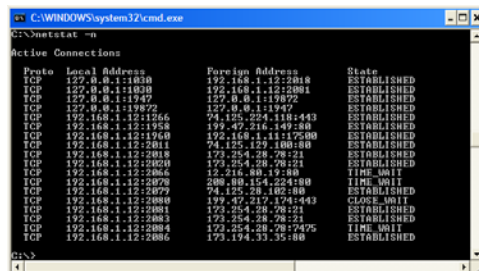
Non-authoritative answer:
Name: google.com
Address: 74.125.224.70, 74.125.224.71, 74.125.224.72, 74.125.224.73
74.125.224.78, 74.125.224.64, 74.125.224.65, 74.125.224.66, 74.125.224.67
74.125.224.68, 74.125.224.69

C:\>
  
```

3. netstat(network statistics)

A command-line tool that displays network connections (both incoming and outgoing), routing tables, and a number of network interface (network interface controller or software-defined network interface) and network protocol statistics.

Simply put, it provides useful information about the current TCP/IP settings of a connection.



```

C:\WINDOWS\system32\cmd.exe
C:\>netstat -n

Active Connections

Proto Local Address Foreign Address State
TCP 127.0.0.1:1030 192.168.1.12:8088 ESTABLISHED
TCP 127.0.0.1:1030 192.168.1.12:8081 ESTABLISHED
TCP 127.0.0.1:1747 127.0.0.1:19872 ESTABLISHED
TCP 127.0.0.1:19872 127.0.0.1:1747 ESTABLISHED
TCP 192.168.1.12:1266 74.125.224.110:443 ESTABLISHED
TCP 192.168.1.12:1958 192.42.216.149:80 ESTABLISHED
TCP 192.168.1.12:1960 192.168.1.11:17500 ESTABLISHED
TCP 192.168.1.12:2011 74.125.129.100:80 ESTABLISHED
TCP 192.168.1.12:2018 173.254.28.70:21 ESTABLISHED
TCP 192.168.1.12:2020 173.254.28.70:21 ESTABLISHED
TCP 192.168.1.12:2066 12.216.80.49:80 TIME_WAIT
TCP 192.168.1.12:2070 208.80.154.224:80 TIME_WAIT
TCP 192.168.1.12:2079 74.125.28.102:80 ESTABLISHED
TCP 192.168.1.12:2080 192.52.212.174:443 CLOSE_WAIT
TCP 192.168.1.12:2081 173.254.28.70:21 ESTABLISHED
TCP 192.168.1.12:2083 173.254.28.70:21 ESTABLISHED
TCP 192.168.1.12:2084 173.254.28.70:17425 TIME_WAIT
TCP 192.168.1.12:2086 173.194.33.35:80 ESTABLISHED

C:\>
  
```

4. traceroute

Traceroute is a computer network diagnostic tool for displaying the route (path) and measuring transit delays of packets across an Internet Protocol (IP) network. **Tracert** is a Windows utility program that can be used to trace the route taken by data from the router to the destination network. It also shows the number of hops taken during the entire

transmission route.

```
C:\WINDOWS\system32\cmd.exe
C:\>tracert google.com

Tracing route to google.com [74.125.224.142]
over a maximum of 30 hops:
  0  7 ms  2 ms  2 ms  hnsportal1 [192.168.1.254]
  1  *      *      *      Request timed out.
  2  *      *      *      Request timed out.
  3  *      *      *      Request timed out.
  4  *      *      *      Request timed out.
  5  *      *      *      Request timed out.
  6  27 ms  27 ms  27 ms  12.82.77.149
  7  28 ms  28 ms  28 ms  12.122.114.21
  8  *      *      *      Request timed out.
  9  *      *      *      Request timed out.
 10  42 ms  42 ms  41 ms  209.85.249.3
 11  42 ms  42 ms  42 ms  64.233.174.119
 12  29 ms  29 ms  29 ms  nsg64007-14.14.1e188.net [74.125.224.142]

Trace complete.
```

Here's how traceroute works.

Traceroute **probes successive hops** in order to find the network path between a host that's doing the probing and a destination host which might be some remote web server. And, we want to find the path, the network path between our computer and the remote host server.

What traceroute does is it sends a packet towards that remote host, only a single hop onto the network. And then, causes the network to send a message back, or a reply back. Then, it sends a packet two hops into the network. It elicits a response from there, and so on. And eventually, the packet will reach the remote host, which will then send a response back.

This gives us information about what routers are between our computer and the host, the number of them, and the sequence in which they're organized.

xinetd/inetd

The **internet daemon (Disk And Execution MONitor)**

[xinetd/inetd] listens for connections on many ports at once.

When a client connects to a service, the daemon program runs the appropriate server. So, this boils down to the need for servers to be running all the time.

xinetd can be configured by modifying its configuration file.

They are typically in **/etc/xinetd.conf** and files in **/etc/xinetd.d** directory.

Here is the list all of the services that are enabled extracted by using **grep -v "^#" /etc/inetd.conf**

```
defaults
{
    log_type           = SYSLOG daemon info
    log_on_failure     = HOST
    log_on_success     = PID HOST DURATION EXIT

    cps               = 50 10
    instances         = 50
    per_source        = 10

    v6only            = no

    groups            = yes
    umask              = 002
}
includedir /etc/xinetd.d
```

As an example of **xinetd** configuration file, here is the **daytime** service:

```
# This is the configuration for the tcp/stream daytime service

service daytime
{
    # This is for quick on or off of the service
    disable      = yes

    # The next attributes are mandatory for all services
    id           = daytime-stream
    type         = INTERNAL
    wait         = no
    socket_type  = stream
    protocol     = socket type is usually enough
    .....
}
```

The **daytime** service that the **getdate** program connects to is actually handled by **xinetd** itself, and it can be made available using both **SOCK_STREAM (tcp)** and **SOCK_DGRAM (udp)** sockets.

Another example for **file transfer** service:

```
# default: off
# description: The kerberized FTP server accepts FTP connections
#             that can be authenticated with Kerberos 5.
service ftp
{
    flags          = REUSE
    socket_type    = stream
    wait           = no
    user           = root
    server         = /usr/kerberos/sbin/ftpd
    server_args    = -l -a
    log_on_failure += USERID
    disable        = yes
}
```

The **ftp** file transfer service is available only via **SOCK_STREAM** sockets and is provided by external program, in this case **gssftp**. The daemon will start these external program when a client connects to the **ftp** port.

To activate service configuration, we can edit the **xinetd** configuration and send a hang-up signal to the daemon process. If a daemon process has a configuration file which is modified after the process has been started, there should be a way to tell that process to re-read its configuration file, without stopping the process. Many daemons provide this mechanism using the **SIGHUP** signal handler. When we want to tell the daemon to re-read the file we simply send it the **SIGHUP** signal.

```
killall -HUP xinetd
```

Network Sniffing

On a **switched network**, each packet moves between a computer and a port on a switch, or between two switches. It's the job of the switch to transmit a packet only when the line is clear, and only to the necessary ports. This way, it's like each computer is on its own private network. We never have packet collisions and we leave as much of our network clear as possible. This means we get the most throughput possible out of our network.

On an **unswitched network**, all packets go to all ports and visible to all computers. If two computers want to send a packet at the same time, we have a collision. Both computers must resend, and that particular bit of bandwidth used for the broken transmission is completely wasted. This means that actual network throughput is much lower than the theoretical potential.

On an **unswitched network**, where Ethernet packets pass through every device on the network, expecting each system device to only look at the packets sent to its destination address. But it is quite trivial to set a device to **promiscuous mode**, which causes it to look at all packets, regardless of the destination address. Most packet-capturing codes, such as **tcpdump**, drop the device they are listening to into promiscuous mode by default. This promiscuous more can be set using **ifconfig**:

```
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:1D:09:67:11:69
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:0
          RX packets:4186029513 errors:0 dropped:0 overru
          TX packets:3343760394 errors:0 dropped:0 overru
          collisions:0 txqueuelen:1000
          RX bytes:1289597250 (1.2 GiB)  TX bytes:3321844
          Interrupt:169  Memory:f8000000-f8012800
```

```
$ sudo ifconfig eth0 promisc
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:1D:09:67:11:69
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:150
          RX packets:4186031551 errors:0 dropped:0 overru
          TX packets:3343761162 errors:0 dropped:0 overru
          collisions:0 txqueuelen:1000
          RX bytes:1290474745 (1.2 GiB)  TX bytes:3322962
          Interrupt:169  Memory:f8000000-f8012800
```

To go back to the state before:

```
$ sudo ifconfig eth1 -promisc
```

tcpdump allows us to save the packets that are captured, so that we can use it for future analysis. The saved file can be viewed by the same tcpdump command.

```
$ sudo tcpdump -i -x
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
23:25:49.194398 IP 171.13.148.146.ssh > ads1-99-67-214-221.dsl.pltn3.sbcglobal.net.hacl-qs: P
3132292720:11229386(116) ack 9534613 win 65535
0x0000: 4510 009c ff07 4000 4006 9965 d32b 9492  E....0.0..e.e..
0x0010: 6343 d6dd 0016 0406 bab2 fe70 368f 138b  C.....p6...
0x0020: 5018 ffff a26d 0000 ea26 d817 8ead 2a95  P...M...d...
0x0030: cf83 15ec 8861 e8ed c75a b9ce 207f 47f8  ....a....G...
0x0040: 87ba 0a9f 2463 03ed d732 7f6c 72c9 64ef  ..-Kc...-f-d-
0x0050: c635
23:25:49.194416 IP 171.13.148.146.ssh > ads1-99-67-214-221.dsl.pltn3.sbcglobal.net.hacl-qs: P
116:232(116) ack 1 win 65535
0x0000: 4510 009c ff08 4000 4006 9964 d32b 9492  E....0.0..d.e..
0x0010: 6343 d6dd 0016 0406 bab2 feea 368f 138b  C.....6...
0x0020: 5018 ffff a26d 0000 f65a fe1c e6d0 e7ff  P...M...f...
0x0030: beef 6261 1655 1702 c841 fa83 5180 570c  -baU...s.W...
0x0040: a159 5674 c19a 7116 82ed 047c 4a55 90c8  .YVt.q....|3U..
0x0050: 007f
23:25:49.194712 IP 171.13.148.146.51163 > ns.lgdacon.net.domain: 53206+ PTR? 221.214.67.99.in-
addr.rpa. (4)
0x0000: 4500 0048 0f99 4000 4011 b9cf d32b 9492  E..W..0.0.....
0x0010: a87c 6502 c7d8 0035 0034 7182 c7d8 0100  .le...S.Q....
0x0020: 0001 0000 0000 0000 0332 3231 0332 3134  ....221.214
0x0030: 0236 3702 3839 0769 6e29 6164 6472 0461  .67.99.in-addr.a
0x0040: 7270 6100 000c 0001  rpa....
```

We can also use open source software like **wireshark** to read the **tcpdump pcap (Packet CAPture)** files (Unix-like systems implement pcap in the **libpcap** library).

Capturing packets sometimes called **sniffing** when it's not necessarily meant for public viewing. Sniffing packets in promiscuous more on an unswitched network can turn up lots of information.

However, our network is only unswitched within individual access points, and there is some provision in wifi to allow multiple access points in the same space.

Raw_Sock Sniffing

In this section, we are accessing the network at lower level than transport/session layer. By using **raw sockets**, all the details are exposed and must be handled explicitly. Note that we've been using **stream sockets**, where the data is neatly wrapped in a TCP/IP connection. Accessing session layer, the OS takes care of all of the lower-level details of transmission, correction, and routing.

By specifying **SOCK_RAW**, we can use **raw sockets**. The protocol matters because there are multiple options. The protocol can be **IPPROTO_TCP** or **IPPROTO_UDP**. Here we are sniffing packet of TCP.

```
// sn.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void packet_dump(const unsigned char *buf, const unsigned
{
    unsigned char c;
    int i,j;
    for(i = 0; i < len; i++) {
        printf("%02x ", buf[i]);
        if((i % 16) == 15 || (i == len-1)) {
            for(j = 0; j < 15 - (i % 16); j++)
                printf(" ");
            for(j = (i - (i % 16)); j <= i; j++)
                c = buf[j];
            if((c > 31) && (c < 127))
                printf("%c", c);
            else
                printf(".");
        }
        printf("\n");
    }
}

int main()
{
    int received_length, sock_fd;
    int i;
    u_char buf[5000];

    if((sock_fd = socket(PF_INET, SOCK_RAW, IPPROTO_T
        printf("error: socket");
        exit(1);
    }

    for(i = 0; i < 5; i++) {
        received_length = recv(sock_fd, buf, 4000
        printf("%d byte packet\n", received_length);
        packet_dump(buf, received_length);
    }
}
```

We opened a raw TCP socket and listens for five packets, print out the raw data of each one with the **packet_dump()**. The **recv()** function receives a message from a socket. The **recv()** call can be used on a connection mode socket or a bound, connectionless socket. If no messages are available at the socket, the **recv()** call waits for a message to arrive unless the socket is nonblocking. If a socket is nonblocking, -1 is returned and the external variable **errno** is set to **EWOULDBLOCK**.

Here the **buf** is declared as a **u_char** type, and it's for our convenience since it's been repeatedly used in network programming. The **u** in **u_char** stands for **unsigned**. Actually, the **char** data-type isn't always used to store characters. Since **char** is the only data type whose size is always **1 byte** on any platform, it is used often to store 1 byte data. 1 byte can hold 255 values but the regular **char** datatype is a signed type and hence stores values from -127 to 127 i.e. After 127, the number is represented in 2's complement notation and hence the numbers are represented as negative. To use only the values 0 to 255, the unsigned type is used. In this case, everything is considered as a positive number and 2's complement is not taken.

To run the compiled code, we need to have root privilege since the use of raw sockets requires it.

```
$ gcc -o sn sn.c
$ sudo ./sn
40 byte packet
45 00 00 28 87 ee 40 00 74 06 dd 0c 63 43 d6 dd | E..(..@
d3 2b 94 88 07 ef 00 16 c5 bc d9 8c ab c6 ca 28 | .+.....
50 10 fa 63 f6 5d 00 00 | P..c.].
40 byte packet
45 00 00 28 87 f1 40 00 74 06 dd 09 63 43 d6 dd | E..(..@
d3 2b 94 88 07 ef 00 16 c5 bc d9 8c ab c6 cb 2c | .+.....
50 10 ff ff ef bd 00 00 | P.....
40 byte packet
45 00 00 28 87 f3 40 00 74 06 dd 07 63 43 d6 dd | E..(..@
d3 2b 94 88 07 ef 00 16 c5 bc d9 8c ab c6 cc 30 | .+.....
50 10 fe fb ef bd 00 00 | P.....
40 byte packet
45 00 00 28 87 f4 40 00 74 06 dd 06 63 43 d6 dd | E..(..@
d3 2b 94 88 07 ef 00 16 c5 bc d9 8c ab c6 cd 34 | .+.....
50 10 fd f7 ef bd 00 00 | P.....
40 byte packet
45 00 00 28 87 f5 40 00 74 06 dd 05 63 43 d6 dd | E..(..@
d3 2b 94 88 07 ef 00 16 c5 bc d9 8c ab c6 ce 38 | .+.....
50 10 fc f3 ef bd 00 00 | P.....
```

We captured packets, but it is not reliable and will miss some of the packets in case when there is a lot of traffic. And we only captured TCP packets. To capture other packets such as UDP, we need to open additional raw sockets. The biggest issue of capturing raw socket packets is the dependency on OS.

For more tcpdump, check here
(<http://www.bogotobogo.com/Linux/tcpdump.php>).

Sniffing using libpcap

libpcap was originally developed by the **tcpdump** developers in the Network Research Group at Lawrence Berkeley Laboratory. The low-level packet capture, capture file reading, and capture file writing code of tcpdump was extracted and made into a library, with which tcpdump was linked. It is now developed by the same tcpdump.org group (<http://www.tcpdump.org/>) that develops tcpdump.

The **libpcap** provides a portable framework for low-level network monitoring. It can provide network statistics collection, security monitoring and network debugging. Since almost every system vendor provides a different interface for packet capture,

the libpcap authors created this system-independent API to ease in porting and to alleviate the need for several system-dependent packet capture modules in each application.

We can use the **libpcap** to smooth out the inconsistencies of raw sockets. While the library is still using raw sockets, it knows how to correctly work on with raw sockets on multiple platforms/architectures.

```
// pcap-sn.c

#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>;

void packet_dump(const unsigned char *buf, const unsigned
{
    unsigned char c;
    int i,j;
    for(i = 0; i < len; i++) {
        printf("%02x ", buf[i]);
        if((i % 16) == 15 || (i == len-1)) {
            for(j = 0; j < 15 - (i % 16); j++)
                printf("| ");
            for(j = (i - (i % 16)); j <= i; j++)
                c = buf[j];
                if((c > 31) && (c < 127))
                    printf("%c", c);
                else
                    printf(".");
            }
            printf("\n");
        }
    }
}

int main()
{
    struct pcap_pkthdr header;
    const u_char *packet;
    char err_buf[PCAP_ERRBUF_SIZE];
    char *device;
    pcap_t *pcap_handle;
    int i;

    device = pcap_lookupdev(err_buf);
    if(!device) {
        printf("Error in %s: %s\n", "pcap_lookupdev",
            exit(1);
    }

    printf("Sniffing on device %s\n", device);

    pcap_handle = pcap_open_live(device, 4096, 1, 0,
    if(!pcap_handle) {
        printf("Error in %s: %s\n", "pcap_open_live",
            exit(1);
    }

    for(i = 0; i < 5; i++) {
        packet = pcap_next(pcap_handle, &header);
        printf("%d byte packet\n", header.len);
        packet_dump(packet, header.len);
    }
    pcap_close(pcap_handle);
}
```

The **pcap** functions use a error buffer to return error and status messages, so we used this function to display this buffer:

```
if(!pcap_handle) {
    printf("Error in %s: %s\n", "pcap_open_live",
        exit(1);
}
```

The **err_buf** is the error buffer, its size coming from a define in **pcap.h** set to **256**. The header variable is a **pcap_pkthdr** structure containing extra capture information about the packet,

the time it captured and its length. The **pcap_handle** pointer is similar to a file descriptor but it is used to reference a packet-capturing object:

```
struct pcap_pkthdr header;
const u_char *packet;
char err_buf[PCAP_ERRBUF_SIZE];
char *device;
pcap_t *pcap_handle;
```

The **pcap_lookupdev()** looks for a suitable device to sniff on. This device is returned as a string pointer referencing static function memory. In our case, it is **/dev/eth0**. It will return NULL if it can't find a suitable interface:

```
device = pcap_lookupdev(err_buf);
if(!device) {
    printf("Error in %s: %s\n", "pcap_lookupdev",
        exit(1);
}

printf("Sniffing on device %s\n", device);
```

The **pcap_open_live()** opens a packet-capturing device, and then returns a handle to it. The arguments are the device to sniff, the maximum packet size, a promiscuous flag, a timeout value, and a pointer to the error buffer. Because we want to capture in promiscuous mode, the flag is set to 1:

```
pcap_handle = pcap_open_live(device, 4096, 1, 0,
    if(!pcap_handle) {
        printf("Error in %s: %s\n", "pcap_open_live",
            exit(1);
    }
```

The **pcap_next()** is used in the packet capture loop to get the next packet. This function is passed the **pcap_handle** and a pointer to a **pcap_pkthdr** structure so it can fill it with details of the capture. The function returns a pointer to the packet and then prints the packet. The **pcap_close()** closes the capture interface:

```
for(i = 0; i < 5; i++) {
    packet = pcap_next(pcap_handle, &header);
    printf("%d byte packet\n", header.len);
    packet_dump(packet, header.len);
}
pcap_close(pcap_handle);
```

Output should look like this:

```

$ gcc -o pcap_sn pcap_sn.c -l pcap
$ sudo ./pcap_sn
Sniffing on device eth0
1134 byte packet
01 00 5e 04 50 01 00 22 19 55 11 27 08 00 45 00 | ..^..P..
04 60 71 6b 00 00 10 11 8e 65 d3 2b 94 8b ef 04 | .`qk...
50 01 07 c4 2c a0 04 4c 50 bf 80 61 2d 22 a2 f3 | P....,
e8 a4 00 00 2f 6d 00 00 01 b6 16 58 24 60 5f db | ....m.
7f 1b 6d fc 6d b7 f1 b6 df c6 db 7f 1b 6d fc 6d | ...m.m..
b7 f1 b6 df c6 db 7f 1b 6d fc 6d b7 ed 83 79 90 | .....
.....
62 3e f2 b0 5e c6 07 fe 07 02 99 5e 60 71 75 4f | b>..^..
aa c8 7d f4 40 c2 92 c5 86 b6 b7 5e d9 84 e1 0d | ..).@..
30 f9 b4 cd 17 dd c6 f7 3c 38 9e 53 71 4a      | 0.....
1134 byte packet
01 00 5e 04 50 01 00 22 19 55 11 27 08 00 45 00 | ..^..P..
04 60 71 6c 00 00 10 11 8e 64 d3 2b 94 8b ef 04 | .`ql...
50 01 07 c4 2c a0 04 4c 6f 18 80 61 2d 23 a2 f3 | P....,
e8 a4 00 00 2f 6d 25 a2 84 74 f9 51 05 26 e4 66 | ....m%
51 01 88 de e7 6f 07 0a 71 75 b7 bc 51 fe ac 4a | Q....o.
b0 a8 7a 0c df d2 b3 f6 29 71 6f 9a e5 9f aa e0 | ..Z....
.....
0b a4 b5 a1 d1 77 f5 ba 59 ac 15 28 2d df 5e 4f | .....w.
97 b3 be a1 de 79 e2 1b 8a 84 85 33 bf b3 54 c0 | .....y.
ff f9 5a 03 0c 07 97 f9 73 19 8d 77 ed fb f7 24 | ..Z....
80 cb ec cc fc 2d 0b 41 92 0c 90 0e a4 06      | .....-..

```

Packet Sniffing using wget and tcpdump

In this section, we will check the response to the http url hitting-
<http://ad.doubleclick.net/ad/DY212/?order=sny0003;sz=1x1;ord=1322627560>,
to see we are getting http 200 response.

We're going to do packet dump tcpdump (we may use wireshark)
when our server fires the tracking pixel? I am using linux
commend:

```
sudo tcpdump port 80 -XX -s 1024
```

and use another linux window to run:

```
wget http://ad.doubleclick.net/ads/813/?order=smsn0009;sz=
```

Then we get the output something like this:

```

23:50:12.616992 IP hx-in-f149.1e100.net.http > 201.13.198
P 475:859(384) ack 268 win 123
0x0000: 001d 0967 1169 2c6b f562 1105 0800 4500 ...g.i
0x0010: 01b4 f2f9 0000 2d06 9f7a 4a7d 4795 d32b .....
0x0020: 9492 0050 d524 dca0 9756 c704 1fe5 8018 ...P.$
0x0030: 007b a053 0000 0101 080a 7d60 c38b a55e .{.S..
0x0040: 1066 4854 5450 2f31 2e30 2032 3030 204f ..fHTTP
0x0050: 4b0d 0a43 6f6e 7465 6e74 2d54 7970 653a ..K..Co
0x0060: 2069 6d61 6765 2f67 6966 0d0a 4c61 7374 ..image
0x0070: 2d4d 6f64 6966 6965 643a 2053 756e 2c20 -Modif
0x0080: 3031 2046 6562 2032 3030 3920 3038 3a30 01.Feb
0x0090: 303a 3030 2047 4d54 0d0a 4461 7465 3a20 0:00.G
0x00a0: 5765 642c 2033 3020 4e6f 7620 3230 3131 Wed,.3
0x00b0: 2032 313a 3433 3a31 3420 474d 540d 0a45 ..21:43
0x00c0: 7870 6972 6573 3a20 5468 752c 2030 3120 xpires
0x00d0: 4465 6320 3230 3131 2032 313a 3433 3a31 Dec.20
0x00e0: 3420 474d 540d 0a58 2d43 6f6e 7465 6e74 4.GMT.
0x00f0: 2d54 7970 652d 4f70 7469 6f6e 733a 206e -Type=
0x0100: 6f73 6e69 6666 0d0a 5365 7276 6572 3a20 osniff
0x0110: 7366 6665 0d0a 436f 6e74 656e 742d 4c65 sffe..
0x0120: 6e67 7468 3a20 3433 0d0a 582d 5853 532d ngth:..

```

The 200 OK means doubleClick is returning the response properly. In other words, the pixel is fired properly, then we can assure it's been registered a hit.

Also see php cURL (../php/php13B_curl.php)

Blocking socket vs non-blocking socket

By default, TCP **socket** is set as **blocking (sleeping)** mode.

What does it mean blocking? Well, we're doing communication. We do something, and wait for the response. So, until we get the response, we cannot do anything. Just wait... and wait. What's the response? It could be either normal message or error.

For example, client did call **receive()** to remote server to read from a stream, control isn't returned to our program until at least one byte of data is read from the remote site. We can call this waiting process as **blocking**. If we use **recv()** in **non-blocking** mode by setting a descriptor as such, it will return any data that the system has in its read buffer for that socket. But, it won't wait for that data.

Among the socket APIs we used, **accept()**, **connect()**, **recv()**, and **recvfrom()** are blocking functions.

When we first create the socket descriptor with **socket()**, the kernel sets it to blocking. If we do not want a socket to be blocking, we have to make a call to **fcntl()**:

```
#include <unistd.h>
#include <fcntl.h>

sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

By setting a socket to non-blocking, we can poll the socket for information. If we try to read from a non-blocking socket and there's no data there, it's not allowed to block-it will return -1 and errno will be set to EWOULDBLOCK. The **non-blocking** mode is set by changing one of the socket's flags. The flags are a series of bits, each one representing a different capability of the socket.

Actually, it's a three-step process:

1. use F_GETFL to get the current flags
2. set or clear the O_NONBLOCK flag
3. then use F_SETFL to set the flags.

For more info, Blocking vs. non-blocking sockets (<http://www.scottklement.com/rpg/socketut/nonblocking.html>).

However, in general, the polling is not a good idea because it makes our program in a busy-wait looking for data on the socket, and it will consume the precious CPU time.

We have another way of checking to see if there's data waiting to be read: **select()**!

What are the pros/cons of select(), non-blocking I/O and SIGIO?

Using **non-blocking I/O** means that we have to poll sockets to see if there is data to be read from them. Polling should usually be avoided since it uses more CPU time than other techniques.

Using SIGIO allows our application to do what it does and have the operating system tell it (with a signal) that there is data waiting for it on a socket. The only drawback to this solution is that it can be confusing, and if we're dealing with multiple sockets we will have to do a **select()** anyway to find out which one(s) is ready to be read.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

The function monitors sets of file descriptors; in particular **readfds**, **writefds**, and **exceptfds**. If we want to see if we can read from standard input and some socket descriptor, **sockfd**, just add the file descriptors **0** and **sockfd** to the set **readfds**. The parameter **numfds** should be set to the values of the highest file descriptor plus one. In this example, it should be set to **sockfd+1**, since it is assuredly higher than standard input (0).

When **select()** returns, **readfds** will be modified to reflect which of the file descriptors you selected which is ready for reading.

Using **select()** is great if our application has to accept data from more than one socket at a time since it will block until any one of a number of sockets is ready with data. In other words, **select()** gives us the power to monitor several sockets at the same time. It'll tell us which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions, if we really want to know that.

One other advantage to **select()** is that we can set a time-out value after which control will be returned to us whether any of the sockets have data for us or not.

The **select()**, though very portable, is one of the slowest methods for monitoring sockets. One possible alternative is **libevent** that encapsulates all the system-dependent stuff involved with getting socket notifications.

Active FTP vs Passive FTP

1. Active FTP

1. FTP server's port 21 from anywhere (Client initiates connection)
2. FTP server's port 21 to ports > 1023 (Server responds to client's control port)
3. FTP server's port 20 to ports > 1023 (Server initiates data connection to client's data port)
4. FTP server's port 20 from ports > 1023 (Client sends ACKs to server's data port)

(*)The main problem with active mode FTP actually falls on the client side. The FTP client doesn't make the actual connection to the data port of the server--it simply tells the server what port it is listening on and the server connects back to the specified port on the client. From the client side firewall this appears to be an outside system initiating a connection to an internal client--something that is usually blocked.

2. Passive FTP

1. FTP server's port 21 from anywhere (Client initiates connection)
2. FTP server's port 21 to ports > 1023 (Server responds to client's control port)
3. FTP server's ports > 1023 from anywhere (Client initiates data connection to random port specified by server)
4. FTP server's ports > 1023 to remote ports > 1023 (Server sends ACKs (and data) to client's data port)

Client-Server : VPN

Visit VPN

(<http://www.bogotobogo.com/VideoStreaming/VPN.php>).

Switch vs Router

1. **Hub** can't identify the source or intended destination of the information it receives, so it sends the information to all of the computers connected to it, including the one that sent it. A hub can send or receive information, but it can't do both at the same time. This makes hubs slower than switches. Hubs are the least complex and the least expensive of these devices.
2. **Switches** work the same way as hubs, but they can identify the intended destination of the information that they receive, so they send that information to only the computers that are supposed to receive it. Switches can send and receive information at the same time, so they can send information faster than hubs can.
3. **Routers** enable computers to communicate and they can pass information between two networks
4. **Switches** usually work at **Layer 2 (Data or Datalink)** of the OSI Reference Model, using **MAC addresses**
5. **Routers** work at **Layer 3 (Network)** with Layer 3 addresses (IP).

6. The algorithm that switches use to decide how to forward packets is different from the algorithms used by routers to forward packets.
7. One of these differences in the algorithms between switches and routers is how **broadcasts** are handled.
8. On any network, the concept of a broadcast packet is vital to the operability of a network. Whenever a device needs to send out information but doesn't know who it should send it to, it sends out a broadcast. Broadcasts are used any time a device needs to make an announcement to the rest of the network or is unsure of who the recipient of the information should be.
9. A **hub** or a **switch** will pass along any broadcast packets they receive to all the other segments in the broadcast domain.
10. But a **router will not**. Without the specific address of another device, it will not let the data packet through. This is a good thing for keeping networks separate from each other, but not so good when we want to talk between different parts of the same network. This is where switches come in.
11. **Switches** don't scale to large networks: table for all destinations may blow up and it may broadcast new destinations to the whole world.
12. While there are several technologies such as Ethernet, 4G, and wireless, **switches** don't work across more than one link layer technology.
13. **Switches** do not provide much for traffic control.

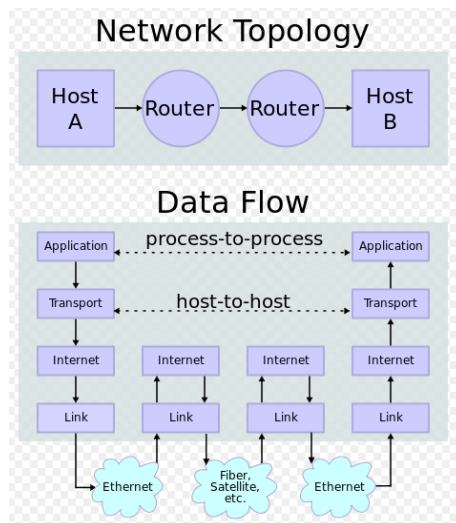
Socket programming with Qt

TCP sockets using Qt

(http://www.bogotobogo.com/cplusplus/sockets_server_client_QT.php).

Networking

4-Layer Model



Picture from wiki

(http://en.wikipedia.org/wiki/File:IP_stack_connections.svg)

Protocols and **layering** is the primary structuring method used to divide up network functionality. Each protocol instance talks virtually to its peer using the protocol. Also, each instance of a protocol uses only the services of the lower layer.

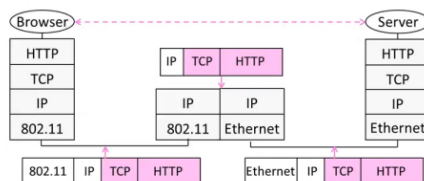
This is about modularization of a complex system. As we already know the protocol refers to a sequence of communication and computation to control the system. So a modularize the protocol is what people call the layer **protocol stack**.

This is not for efficiency but for **evolvability**. In other words, this allows specialization of business sectors, and a common interface among them but also for technology reasons. We have so many unforeseen and unforeseeable needs in the future for our technology that we would rather keep a stack where we can pull out one part of the whole system without having to redesign the entire system.

Encapsulation is the mechanism used for protocol layering. So, the lower layer wraps higher layer content, adding its own control information (header/trailer), compression/encryption, segmentation/disassemble, etc. to make a new message for delivery.

Advantages of the network layers abstraction (encapsulation):

1. Break a complex task of communication into smaller pieces.
2. Lower layers can change implementation without affecting upper layers as long as the interface between layers remains the same. For example, the difference in the underlying connection systems (between wire and wireless) does not affect the upper layer communications as shown in the picture below:



3. Lower layers hide the implementation details from higher layers.

Unit of Data for each layer of Reference Model

| Layer | Unit of Data |
|-------------|--------------|
| Application | Message |
| Transport | Segment |
| Network | Packet |
| Link | Frame |
| Physical | Bit |

Network Layer (IP)

Summary of IP Network Layer

1. The Internet protocol (IP) is an example of a network layer, and is required for all communications in the Internet.
2. There are currently two main versions of the IP protocol used in the Internet: IP Version 4, and IP Version 6.
3. The Internet protocol is responsible for delivering self-contained datagrams from a source host to the specified destination.
4. It makes no promise to deliver packets in order, or at all.
5. It has a feature to prevent packets looping forever (TTL).
6. It will fragment packets if they are too long.
7. It uses a checksum to reduce chances of delivering to wrong address.

| Property | Behavior |
|----------------|---|
| Data | individually routed packets. Hop-by-hop routing. |
| Unreliable | Packet might be dropped |
| Best effort | if necessary |
| Connectionless | No per-flow state Packets may not be in order |

(*note) An Internet router is allowed to drop packets when it has insufficient resources(best effort service). There can also be cases when resources are available (e.g., link capacity) but the router drops the packet anyways. The following are examples of scenarios where a router drops a packet even when it has sufficient resources:

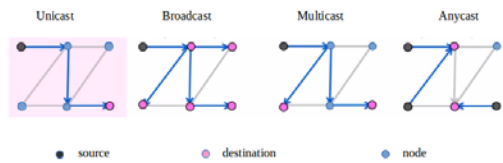
1. A router configured as a firewall, that dictates which packets should be denied.
2. An ISP that limits bandwidth consumed by customers, even though there is available capacity.

Transport Layer (TCP)

- 1. TCP is responsible for providing reliable, in-sequence end-to-end delivery of data between applications. In other words, TCP delivers a stream of bytes from one end to the other, reliably and in-sequence, on behalf of an application.
- 2. When a TCP packet arrives at the destination, the data portion is delivered to the service (or application) identified by the destination port number.
- 3. TCP will retransmit missing data even if the application can not use it - for example, in Internet telephony a late arriving retransmission may arrive too late to be useful.
- 4. TCP saves an application from having to implement its own mechanisms to retransmit missing data, or resequence arriving data.

| Property | Behavior |
|---------------------|--|
| Connection oriented | Three-way handshake (http://www.bogotobogo.com/cplusplus/sockets_server_client.php#three_way_handshake) for connection setup. |
| Reliable | Acknowledgments indicate delivery. Checksums detect corrupted data. Sequence numbers detect missing data. Flow-control prevents overrunning receiver. |
| In-sequence | Data delivered to application in sequence transmitted. |
| Congestion Control | It controls network congestion |

Packet Delivery Models

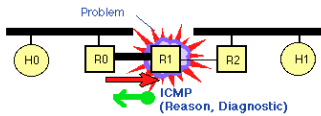


Internet Control Message Protocol (ICMP)

Making Network layer work.

- 1. Internet Protocol (IP)
 - 1. creat IP datagrams
 - 2. deliver datagrams from end to end hop-by-hop
- 2. Routing Tables - algorithms to populate router forwarding tables
- 3. ICMP
 - 1. Examples: ping, tracerouter
 - 2. communicates network layer information between end hosts and routers
 - 3. reports error conditions
 - 4. helps to diagnose problems

| Property | Behavior |
|-------------------|--|
| Reporting Message | Self-contained message reporting error |
| Unreliable | Simple datagram service - no retries |



Picture from Internet Control Message Protocol (ICMP)
(<http://www.erg.abdn.ac.uk/~gorry/eg3567/inet-pages/icmp.html>)

ping

- 1. ping can be used to measure end-to-end delay.
- 2. ping can be used to test if a machine is alive.
- 3. ping can be maliciously used as a way to attack a machine by flooding it with ping requests.
- 4. ping sends out **ICMP ECHO_REQUEST** message to the destination.

traceroute

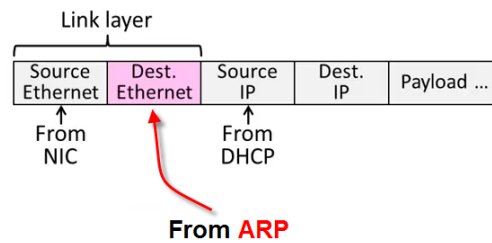
It contains a client interface to ICMP. Like the **ping**, it may be used by a user to verify an end-to-end Internet Path is operational, but also provides information on each of the Intermediate Systems (i.e. IP routers) to be found along the IP Path from the sender to the receiver. **traceroute** uses ICMP echo messages. These are addressed to the target IP address. The sender manipulates the **TTL (hop count)** value at the IP layer to force each hop in turn to return an error message.

Address Resolution Protocol (ARP)

We can retrieve MAC address (Ethernet address) via the **Address Resolution Protocol (ARP)**.

- 1. A network device (e.g. laptop) sends an ARP request to the switch ("I want the MAC address of the device with IP address 192.168.102.3").
- 2. The switch broadcasts the ARP request to all devices.
- 3. The device with the appropriate IP address makes an ARP response back to the switch.
- 4. The switch relays the ARP response back to the network device.

This is in a sense reverse of the DHCP where obtaining IP by giving device info. ARP provides IP info to get device info (MAC address).



Error Detection

Sending bits via network is not perfect, and some bits may be received in error whether due to loss or due to a noise in the signal. How do we detect the error in bits?

Here we will discuss three ways of detecting it:

1. Parity
2. Checksums
3. CRC (Cyclic Redundancy Check)s

Note that those are limited to error detection but not the correction as done in Hamming code (http://en.wikipedia.org/wiki/Hamming_code) etc.

1. Parity

This is the simplest.

We take n data bits, add 1 check bit that is modulo 2 for the sum of the D bits.

For example, let's take 7 bit data: 1001100.

The sum of the bit is 3, then if we do modulo, $3 \% 2 = 1$

So, the parity bit becomes **1**.

The bits we're sending is now 10011001

Note that we used one of the two variants: **even** parity bit.

We could have used the **odd** parity bit as shown in the table below.

| 7 bit data | # of 1 bits | Even parity | Odd parity |
|------------|-------------|-------------|------------|
| 0000000 | 0 | 00000000 | 00000001 |
| 10100010 | 3 | 10100011 | 10100010 |
| 1101001 | 4 | 11010010 | 11010011 |
| 1111111 | 7 | 11111111 | 11111110 |

Here is scenario for the successful transmission for even parity assuming we are sending a simple 7-bit value 1001100 with the parity bit (8th bit) following on the right, and with ^ denoting an XOR gate:

1. A wants to transmit: 1001100
2. A computes parity bit value: $1 \wedge 0 \wedge 0 \wedge 1 \wedge 1 \wedge 0 \wedge 0 = 1$
3. A adds parity bit and sends: 10011001
4. B receives: 10011001
5. B computes parity: $1 \wedge 0 \wedge 0 \wedge 1 \wedge 1 \wedge 0 \wedge 0 \wedge 1 = 0$

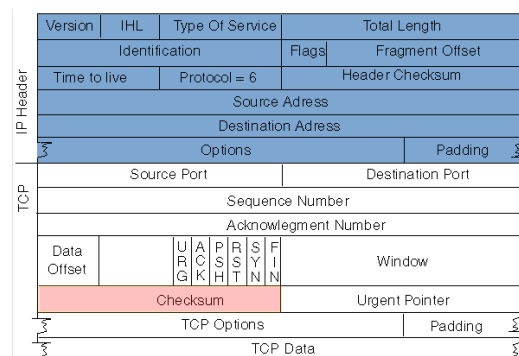
6. B reports correct transmission after observing expected even result.

Summary:

1. If an odd number of bits (including the parity bit) are transmitted incorrectly, the parity bit will be incorrect, thus indicating that a parity error occurred in the transmission.
2. The parity bit is only suitable for detecting errors; it cannot correct any errors, as there is no way to determine which particular bit is corrupted. The data must be discarded entirely, and re-transmitted from scratch.
3. On a noisy transmission medium, successful transmission can therefore take a long time, or even never occur.
However, parity has the advantage that it uses only a single bit and requires only a number of XOR gates to generate.
4. Parity bit checking is used occasionally for transmitting ASCII characters, which have 7 bits, leaving the 8th bit as a parity bit.

2. Checksums

Checksums are widely used in TCP/IP/UDP for error detection and provides stronger protection than parity.



Picture from Cisco: TCP Performance

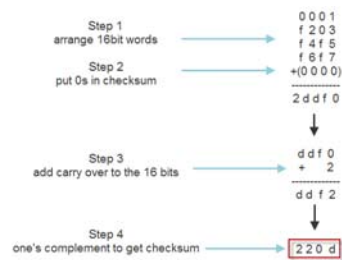
(http://www.cisco.com/web/about/ac123/ac147/ac174/ac196/about_cisco_ipj_archive_article09186a00800c8417.html)

Here is the description of **checksum** in RFC793:

The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words...

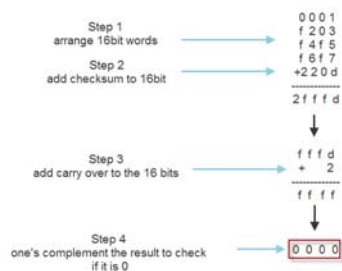
Sending can be divided into 4 steps:

1. Arrange data in 16-bit words
2. Put zero in checksum position
3. Add any carryover back to get 16 bits
4. Complement to get sum



Receiving can also be divided into 4 steps:

1. Arrange data in 16-bit words
2. Add checksum to the 16-bit words
3. Add any carryover back to get 16 bits
4. Complement the result and check if it is 0



3. CRC (Cyclic Redundancy Check)

CRCs are so called because the check (data verification) value is a redundancy (it expands the message without adding information) and the algorithm is based on cyclic codes. - wiki
(http://en.wikipedia.org/wiki/Cyclic_redundancy_check)

Given **n** data bits, generate **k** check bits such that the **n+k** bits are evenly divisible by a divisor **D**.

For example, $n=301$, $k=1$, and $D=3$:

the bits to send would be 4 bits: 301?. But we can start with 3010. $3010 \% 3 = 1$, so to make it divisible by $D=3$, it should be 3012.

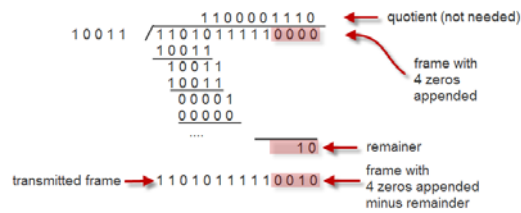
Sending procedure should be like this:

1. Extend the **n** data bits with **k** zeros.
2. Divide by the divisor **D**.
3. Keep remainder, and throw away quotient.
4. Adjust **k** check bits by remainder.

This picture below is for the case when

1. Data bits: 110101111
2. Check bits, **k** = 4
3. Divisor, **D** = 10011

Receiving procedure is the same, and need to check if the remainder is zero.



Network Applications

Dominant model for network applications is **TCP Byte Stream** model where one side writes and the other side reads. It's the building block of most applications today though there are other models such as datagrams, real-time streams.

1. web server http
2. skype client
 - Rendezvous service that allows users not behind a NAT to call users behind a NAT.
 - An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol by Salman A. Baset and Henning Schulzrinne(pdf) (http://www.bogotobogo.com/cplusplus/files/socket/Skype_peer_to_peer.pdf)
3. Bit Torrents
 - Tit For Tat algorithm - gives download preference to peers that give data to you.
 - Visit P2P (<http://www.bogotobogo.com/cplusplus/files/socket>)
 - <https://wiki.theory.org/BitTorrentSpecification>
 - <https://wiki.theory.org/BitTorrentSpecification>

References

Beej's Guide to Network Programming Using Internet Sockets (http://www.bogotobogo.com/cplusplus/files/socket/bgnet_USLetter.pdf) or get it from http://beej.us/guide/bgnet/output/print/bgnet_USLetter.pdf (http://beej.us/guide/bgnet/output/print/bgnet_USLetter.pdf)

BogoToBogo
contactus@bogotobogo.com (mailto:#)

FOLLOW BOGOTOBOGO

f (<https://www.facebook.com/KHongSanFrancisco>)
t (<https://twitter.com/KHongTwit>) **g+**
(<https://plus.google.com/u/0/+KHongSanFrancisco/posts>)

ABOUT US (/ABOUT_US.PHP)

contactus@bogotobogo.com (mailto:#)

Golden Gate Ave, San Francisco, CA 94115

Golden Gate Ave, San Francisco, CA 94115

Copyright © 2016, bogotobogo
Design: Web Master (<http://www.bogotobogo.com>)