

Contents



UNIT-1

Unit-1: Networking and TCP/IP

Communication Protocol

In computing and telecommunications, a **protocol** or **communications protocol** is a formal description of message formats and the rules for exchanging those messages. Protocols may include signaling, authentication and error detection and correction capabilities. In its simplest

form, a protocol can be defined as the rules governing the syntax, semantics, and synchronization of communication. Protocols may be implemented by hardware, software, or a combination of the two. At the lowest level, a protocol defines the behavior of a hardware connection.

Importance of communication protocol

The protocols in human communication are separate rules about appearance, speaking, listening and understanding. All these rules, also called *protocols of conversation*, represent different layers of communication. They work together to help people successfully communicate. The need for protocols also applies to network devices. Computers have no way of learning protocols, so network engineers have written rules for communication that must be strictly followed for successful host-to-host communication. These rules apply to different layers of sophistication such as which physical connections to use, how hosts listen, how to interrupt, how to say good-bye, and in short how to communicate, what language to use and many others. These rules, or protocols, that work together to ensure successful communication are grouped into what is known as a protocol suite.

The widespread use and expansion of **communications protocols** is both a prerequisite for the Internet, and a major contributor to its power and success. The pair of Internet Protocol (or IP) and Transmission Control Protocol (or TCP) are the most important of these, and the term TCP/IP refers to a collection (a "protocol suite") of its most used protocols. Most of the Internet's communication protocols are described in the RFC documents of the Internet Engineering Task Force (or IETF).

Object-oriented programming has extended the use of the term to include the programming protocols available for connections and communication between objects. Protocols fall into many levels of processes and complexity.

Generally, only the simplest protocols are used alone. Most protocols, especially in the context of communications or networking, are layered together into protocol stacks where the various tasks listed above are divided among different protocols in the stack.

Whereas the protocol stack denotes a specific combination of protocols that work together, a reference model is a software architecture that lists each layer and the services each should offer. The classic seven-layer reference model is the OSI model, which is used for conceptualizing protocol stacks and peer entities. This reference model also provides an opportunity to teach more general software engineering concepts like hiding, modularity, and delegation of tasks. This model has endured in spite of the demise of many of its protocols (and protocol stacks) originally sanctioned by the ISO.

Network Architecture:-Network architecture is the design of a communications network. It is a framework for the specification of a network's physical components and their functional organization and configuration, its operational principles and procedures, as well as data formats used in its operation.

In computing, the network architecture is a characteristics of a computer network. The most

prominent architecture today is evident in the framework of the Internet, which is based on the Internet Protocol Suite.

In telecommunication, the specification of a network architecture may also include a detailed description of products and services delivered via a communications network, as well as detailed rate and billing structures under which services are compensated.

In distinct usage in distributed computing, network architecture is also sometimes used as a synonym for the structure and classification of distributed application architecture, as the participating nodes in a distributed application are often referred to as a *network*. For example, the applications architecture of the public switched telephone network (PSTN) has been termed the Advanced Intelligent Network. There are any number of specific classifications but all lie on a continuum between the dumb network (e.g., Internet) and the intelligent computer network (e.g., the telephone network). Other networks contain various elements of these two classical types to make them suitable for various types of applications. Recently the context aware network, which is a synthesis of the two, has gained much interest with its ability to combine the best elements of both.

UUCP:-UUCP is an abbreviation for **Unix-to-Unix Copy**. The term generally refers to a suite of computer programs and protocols allowing remote execution of commands and transfer of files, email and netnews between computers. Specifically, UUCP is one of the programs in the suite; it provides a user interface for requesting file copy operations. The UUCP suite also includes uux (user interface for remote command execution), uucico (communication program), uustat (reports statistics on recent activity), uuxqt (execute commands sent from remote machines), and uuname (reports the uucp name of the local system).

Although UUCP was originally developed on and is most closely associated with Unix, UUCP implementations exist for several other operating systems, including Microsoft's MS-DOS, Digital's VAX/VMS, Commodore's AmigaOS, and Mac OS.

Technology:-UUCP can use several different types of physical connections and link layer protocols, but was most commonly used over dial-up connections. Before the widespread availability of Internet connectivity, computers were only connected by smaller private networks within a company or organization. They were also often equipped with modems so they could be used remotely from character-mode terminals via dial-up lines. UUCP uses the computers' modems to dial out to other computers, establishing temporary, point-to-point links between them. Each system in a UUCP network has a list of neighbor systems, with phone numbers, login names and passwords, etc. When work (file transfer or command execution requests) is queued for a neighbor system, the uucico program typically calls that system to process the work. The uucico program can also poll its neighbors periodically to check for work queued on their side; this permits neighbors without dial-out capability to participate.

Today, UUCP is rarely used over dial-up links, but is occasionally used over TCP/IP.[1][2]

One example of the current use of UUCP is in the retail industry by Epicor CRS Retail Systems for transferring batch files between corporate and store systems via TCP and dial-up on SCO OpenServer, Red Hat Linux, and Microsoft Windows (with Cygwin).[citation needed] The number of systems involved, as of early 2006, ran between 1500 and 2000 sites across 60 enterprises. UUCP's longevity can be attributed to its low/zero cost, extensive logging, native

failover to dialup, and persistent queue management.

UUCP for mail routing

The uucp and uuxqt capabilities could be used to send e-mail between machines, with suitable mail user interface and delivery agent programs. A simple uucp mail address was formed from the adjacent machine name, an exclamation mark or *bang*, followed by the user name on the adjacent machine. For example, the address *barbox!user* would refer to user *user* on adjacent machine *barbox*.

Mail could furthermore be routed through the network, traversing any number of intermediate nodes before arriving at its destination. Initially, this had to be done by specifying the complete path, with a list of intermediate host names separated by bangs. For example, if machine *barbox* is not connected to the local machine, but it is known that *barbox* is connected to machine *foovax* which does communicate with the local machine, the appropriate address to send mail to would be *foovax!barbox!user*.

User *barbox!user* might publish their UUCP email address in a form such as *...!bigsite!foovax!barbox!user*. This directs people to route their mail to machine *bigsite* (presumably a well-known and well-connected machine accessible to everybody) and from there through the machine *foovax* to the account of user *user* on *barbox*. Many users would suggest multiple routes from various large well-known sites, providing even better and perhaps faster connection service from the mail sender.

Bang path

An email address of this form was known as a **bang path**. Bang paths of eight to ten machines (or *hops*) were not uncommon in 1981, and late-night dial-up UUCP links would cause week-long transmission times. Bang paths were often selected by both transmission time and reliability, as messages would often get lost. Some hosts went so far as to try to "rewrite" the path, sending mail via "faster" routes—this practice tended to be frowned upon.

The "pseudo-domain" ending *.uucp* was sometimes used to designate a hostname as being reachable by UUCP networking, although this was never formally in the Internet root as a top-level domain. This would not have made sense anyway, because the DNS system is only appropriate for hosts reachable directly by TCP/IP. Additionally, uucp as a community administers itself and does not mesh well with the administration methods and regulations governing the DNS; *.uucp* works where it needs to; some hosts punt mail out of SMTP queue into uucp queues on gateway machines if a *.uucp* address is recognized on an incoming SMTP connection

UUCPNET and mapping

UUCPNET was the name for the totality of the network of computers connected through UUCP. This network was very informal, maintained in a spirit of mutual cooperation between systems owned by thousands of private companies, universities, and so on. Often, particularly in the private sector, UUCP links were established without official approval from the companies' upper management. The UUCP network was constantly changing as new systems and dial-up links were added, others were removed, etc.

The *UUCP Mapping Project* was a volunteer, largely successful effort to build a map of the connections between machines that were open mail relays and establish a managed namespace. Each system administrator would submit, by e-mail, a list of the systems to which theirs would connect, along with a ranking for each such connection. These submitted map entries were processed by an automatic program that combined them into a single set of files describing all connections in the network. These files were then published monthly in a newsgroup dedicated to this purpose. The UUCP map files could then be used by software such as "pathalias" to compute the best route path from one machine to another for mail, and to supply this route automatically. The UUCP maps also listed contact information for the sites, and so gave sites seeking to join UUCPNET an easy way to find prospective neighbors.

Connections with the Internet

Many uucp hosts, particularly those at universities, were also connected to the Internet in its early years, and e-mail gateways between Internet SMTP-based mail and UUCP mail were developed. A user at a system with UUCP connections could thereby exchange mail with Internet users, and the Internet links could be used to bypass large portions of the slow UUCP network. A "UUCP zone" was defined within the Internet domain namespace to facilitate these interfaces.

With this infrastructure in place, UUCP's strength was that it permitted a site to gain Internet e-mail and Usenet connectivity with only a dial-up modem link to another cooperating computer. This was at a time when true Internet access required a leased data line providing a connection to an Internet *Point of Presence*, both of which were expensive and difficult to arrange. By contrast, a link to the UUCP network could usually be established with a few phone calls to the administrators of prospective neighbor systems. Neighbor systems were often close enough to avoid all but the most basic charges for telephone calls.

Brief of uucp:-

UUCP (UNIX-to-UNIX Copy Protocol) is a set of Unix programs for copying (sending) files between different UNIX systems and for sending commands to be executed on another system. The main UUCP commands (each supported by a UUCP program) are:

- uucp, which requests the copying of a specific file to another specified system
- uux, which sends a UNIX command to another system where it is queued for execution
- uucico, which runs on a UNIX system as the program that carries out the copying and initiates execution of the commands that have been sent. Typically, this program is run at various times of day; meanwhile, the copy (uucp) and command (uux) requests are queued until the uucico program is run.
- uuxqt, which executes the commands sent by uux, usually after being started by the uucico program

The uucico programs are the programs that actually communicate across a network. There are several network protocols (variations on packet size and error-checking) that can be used by uucico programs, depending on the kinds of carrier networks being used.

XNS:-Background

The Xerox Network Systems (XNS) protocols were created by Xerox Corporation in the late 1970s and early 1980s. They were designed to be used across a variety of communication media,

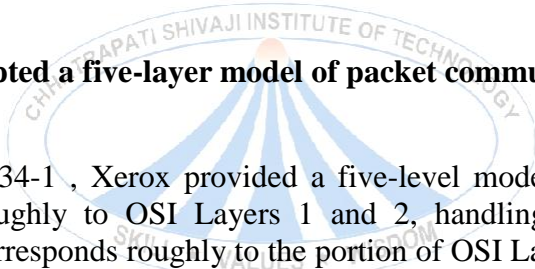
processors, and office applications. Several XNS protocols resemble the *Internet Protocol* (IP) and *Transmission- Control Protocol* (TCP) entities developed by the Defense Advanced Research Projects Agency (DARPA) for the U.S. Department of Defense (DoD).

Because of its availability and early entry into the market, XNS was adopted by most of the early LAN companies, including Novell, Inc., Ungermann-Bass, Inc. (now a part of Tandem Computers), and 3Com Corporation. Each of these companies has since made various changes to the XNS protocols. Novell added the *Service Advertisement Protocol* (SAP) to permit resource advertisement and modified the OSI Layer 3 protocols (which Novell renamed *IPX*, for *Internetwork Packet Exchange*) to run on IEEE 802.3 rather than Ethernet networks. Ungermann-Bass modified RIP to support delay as well as hop count and made other small changes. Over time, the XNS implementations for PC networking have become more popular than XNS as it was designed by Xerox. This chapter presents a summary of the XNS protocol stack in the context of the OSI reference model.

XNS Hierarchy Overview

Although the XNS design objectives are the same as those for the OSI reference model, the XNS concept of a protocol hierarchy is somewhat different from that provided by the OSI reference model, as Figure 34-1 illustrates.

Figure 34-1: Xerox adopted a five-layer model of packet communication.



As illustrated in Figure 34-1, Xerox provided a five-level model of packet communications. Level 0 corresponds roughly to OSI Layers 1 and 2, handling link access and bit-stream manipulation. Level 1 corresponds roughly to the portion of OSI Layer 3 that pertains to network traffic. Level 2 corresponds to the portion of OSI Layer 3 that pertains to internetwork routing, and to OSI Layer 4, which handles interprocess communication. Levels 3 and 4 correspond roughly to the upper two layers of the OSI model, handling data structuring, process-to-process interaction and applications. XNS has no protocol corresponding to OSI Layer 5 (the session layer).

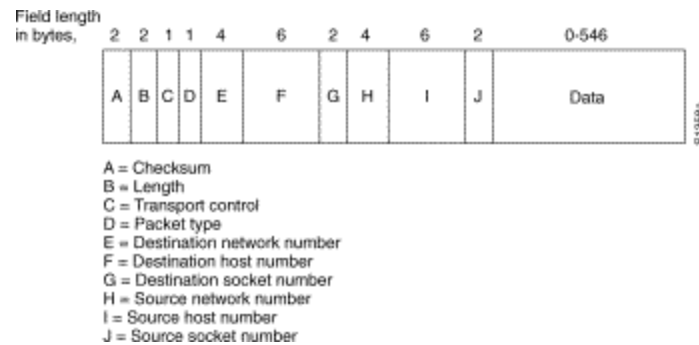
Media Access

Although XNS documentation mentions X.25, Ethernet, and High-Level Data Link Control (HDLC), XNS does not expressly define what it refers to as a Level 0 protocol. As with many other protocol suites, XNS leaves media access an open issue, implicitly allowing any such protocol to host the transport of XNS packets over a physical medium.

Network Layer

The XNS network-layer protocol is called the *Internet Datagram Protocol* (IDP). IDP performs standard Layer 3 functions, including logical addressing and end-to-end datagram delivery across an internetwork. Figure 34-2 illustrates the format of an IDP packet.

Figure Eleven fields comprise an IDP packet.



The following descriptions summarize the IDP packet fields illustrated in Figure

- *Checksum*---A 16-bit field that helps gauge the integrity of the packet after it traverses the internetwork.
- *Length*---A 16-bit field that carries the complete length (including the checksum) of the current datagram.
- *Transport Control*---An 8-bit field that contains the *hop count* and *maximum packet lifetime* (MPL) subfields. The hop- count subfield is initialized to zero by the source and incremented by one as the datagram passes through a router. When the hop- count field reaches 16, the datagram is discarded on the assumption that a routing loop is occurring. The MPL subfield provides the maximum amount of time, in seconds, that a packet can remain on the internetwork.
- *Packet Type*---An 8-bit field that specifies the format of the data field.
- *Destination Network Number*---A 32-bit field that uniquely identifies the destination network in an internetwork.
- *Destination Host Number*---A 48-bit field that uniquely identifies the destination host.
- *Destination Socket Number*---A 16-bit field that uniquely identifies a socket (process) within the destination host.
- *Source Network Number*---A 32-bit field that uniquely identifies the source network in an internetwork.
- *Source Host Number*---A 48-bit field that uniquely identifies the source host.
- *Source Socket Number*---A 16-bit field that uniquely identifies a socket (process) within the source host.

IEEE 802 addresses are equivalent to host numbers, so that hosts that are connected to more than one IEEE 802 network have the same address on each segment. This makes network numbers redundant but nevertheless useful for routing. Certain socket numbers are *well known*, which means that the service performed by the software using them is statically defined. All other socket numbers are reusable.

XNS supports Ethernet Version 2.0 encapsulation for Ethernet, and three types of encapsulation for Token Ring: 3Com, SubNet Access Protocol (SNAP), and Ungermann-Bass.

XNS supports *unicast* (point-to-point), *multicast*, and *broadcast* packets. Multicast and broadcast addresses are further divided into *directed* and *global* types. Directed multicasts deliver packets

to members of the multicast group on the network specified in the destination multicast network address. Directed broadcasts deliver packets to all members of a specified network. Global multicasts deliver packets to all members of the group within the entire internetwork, whereas global broadcasts deliver packets to all internetwork addresses. One bit in the host number indicates a single versus a multicast address. Conversely, all ones in the host field indicate a broadcast address.

To route packets in an internetwork, XNS uses the RIP dynamic routing scheme. Today, RIP is the most commonly used *Interior Gateway Protocol* (IGP) in the Internet community. For more information about RIP, see Chapter 44, "Routing Information Protocol (RIP)."

Transport Layer

OSI transport-layer functions are implemented by several protocols. Each of the following protocols is described in the XNS specification as a Level 2 protocol.

The *Sequenced Packet Protocol* (SPP) provides reliable, connection-based, flow-controlled packet transmission on behalf of client processes. It is similar in function to the Internet Protocol suite's *Transmission-Control Protocol* (TCP) and the OSI protocol suite's *Transport Protocol 4* (TP4). For more information about TCP, see "Internet Protocols." For more information about TP4, see "Open System Interconnection (OSI) Protocols."

Each SPP packet includes a *sequence number*, which is used to order packets and to determine whether any have been duplicated or missed. SPP packets also contain two 16-bit *connection identifiers*. One connection identifier is specified by each end of the connection, and together, the two connection identifiers uniquely identify a logical connection between client processes.

SPP packets cannot be longer than 576 bytes. Client processes can negotiate use of a different packet size during connection establishment, but SPP does not define the nature of this negotiation.

The *Packet Exchange Protocol* (PEP) is a request-response protocol designed to have greater reliability than simple datagram service (as provided by IDP, for example) but less reliability than SPP. PEP is functionally similar to the Internet Protocol suite's *User Datagram Protocol* (UDP). For more information about UDP, see "Internet Protocols." PEP is single-packet based, providing retransmissions but no duplicate-packet detection. As such, it is useful in applications where request-response transactions can be repeated without damaging data, or where reliable transfer is executed at another layer.

The *Error Protocol* (EP) can be used by any client process to notify another client process that a network error has occurred. This protocol is used, for example, in situations where an SPP implementation has identified a duplicate packet.

Upper-Layer Protocols

XNS offers several upper-layer protocols. The *Printing Protocol* provides print services, the *Filing Protocol* provides file-access services, and the *Clearinghouse Protocol* provides name services. Each of these three protocols runs on top of the *Courier Protocol*, which provides conventions for data structuring and process interaction.

XNS also defines Level 4 protocols, which are application protocols, but because they have little to do with actual communication functions, the XNS specification does not include any pertinent

definitions.

The Level 2 *Echo Protocol* is used to test the reachability of XNS network nodes and to support functions such as that provided by the **ping** command found in UNIX and other environments.

IPX/SPX for LAN'S

IPX/SPX stands for **Internet work Packet Exchange/Sequenced Packet Exchange**. IPX and SPX are networking protocols used primarily on networks using the Novell NetWare operating systems.

Protocol Layers

IPX and SPX are derived from Xerox Network Systems' IDP and SPP protocols, respectively. IPX is a network layer protocol (layer 3 of the OSI Model), while SPX is a transport layer protocol (layer 4 of the OSI Model). The SPX layer sits on top of the IPX layer and provides connection-oriented services between two nodes on the network. SPX is used primarily by client/server applications.

IPX and SPX both provide connection services similar to TCP/IP, with the IPX protocol having similarities to IP, and SPX having similarities to TCP. IPX/SPX was primarily designed for local area networks (LANs), and is a very efficient protocol for this purpose (typically its performance exceeds that of TCP/IP on a LAN). TCP/IP has, however, become the *de facto* standard protocol. This is in part due to its superior performance over wide area networks and the Internet (which uses TCP/IP exclusively), and also because TCP/IP is a more mature protocol, designed specifically with this purpose in mind.

Despite the protocols' association with NetWare, they are neither required for NetWare communication (as of NetWare 5.x), nor exclusively used on NetWare networks. NetWare communication requires an NCP implementation, which can use IPX/SPX, TCP/IP, or both, as a transport

Implementation in Windows and Linux

Because of IPX/SPX's prevalence in LANs in the 1990s, Microsoft added support for the protocols into Windows' networking stack, starting with Windows for Workgroups and Windows NT. Microsoft even named their implementation "NWLink", implying that the inclusion of the layer 3/4 transports provided NetWare connectivity. In reality, the protocols were supported as a native transport for Windows' SMB/NetBIOS, and NetWare connectivity required additional installation of an NCP client (Microsoft provided a basic NetWare client with Windows 95 and later, but it was not automatically installed, and initially only supported NetWare bindery mode). NWLink was still provided with Windows (up to and including Windows 2003), but it is neither included with nor supported in Windows Vista. Its use is strongly discouraged because it cannot be used for Windows networking except as a transport for NetBIOS, which is deprecated.

For the most part, Novell's 32-bit Windows client software have eschewed NWLink for an alternative developed by Novell, although some versions permit use of Microsoft's IPX/SPX implementation (with warnings about potential incompatibility).

initially only supported NetWare bindery mode). NWLink was still provided with Windows (up to and including Windows 2003), but it is neither included with nor supported in Windows Vista. Its use is strongly discouraged because it cannot be used for Windows networking except as a transport for NetBIOS, which is deprecated.

Implementations have been written for various flavors of Unix/Linux, both by Novell and other vendors. In particular, Novell's UnixWare supported IPX/SPX natively. However, while UnixWare could act as a client to NetWare servers, and applications could optionally support IPX/SPX as a transport, UnixWare did not provide the ability to share files or printers on a NetWare network without an additional software package. Open Enterprise Server - Linux does not support IPX/SPX.

The open source FreeBSD operating system includes an IPX/SPX stack, to support both a NetWare file system client, nwfs, as well as NetWare server using Mars NWE[1][2] (providing some functionality[3]). OpenBSD dropped support with version 4.2[4][5], and 4.1 needed some work to compile with IPX[6].

TCP and IP Headers

TCP/IP Layers and Protocols:-

TCP and IP together manage the flow of data, both in and out, over a network. Whereas IP indiscriminately pumps packets into the ether, TCP is charged with making sure they get there. TCP is responsible for the following:

- Opening and closing a session
- Packet management
- Flow control
- Error detection and handling

Architecture

TCP/IP is the environment that handles all these operations and coordinates them with remote hosts. TCP/IP was created using the DoD (Department of Defense) model, which is made up of four layers instead of the seven that make up the OSI model.

The primary difference between the OSI and the TCP/IP layer formats is that the Transport Layer does not guarantee delivery at all times. TCP/IP offers the *User Datagram Protocol* (UDP), a more simplified protocol, wherein all the layers in the TCP/IP stack perform specific duties or run applications.

Application Layer

The Application Layer consists of protocols such as SMTP, FTP, NFS, NIS, LPD, Telnet, and Remote Login, all of which fall into areas that are familiar to most Internet users.

Transport Layer

The Transport Layer consists of UDP (User Datagram Protocol) and TCP, where the former delivers packets with almost non-existent checking, and the latter provides delivery guarantees.

Network Layer

The Network Layer is made up of the following protocols: ICMP, IP, IGMP, RIP, OSPF, EGP and BGP4. ICMP (Internet Control Message Protocol) is used for diagnostics and to report on problems with the IP layer and for obtaining information about IP parameters. IGMP (Internet Group Management Protocol) is used for managing multicasting. RIP, OSPF, EGP, and BGP4 are examples of routing protocols.

Link Layer

The Link Layer consists of ARP and RARP, and framing of data, which handle packet transmission.

Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP) is a protocol that provides a reliable stream delivery and connection service to applications. TCP uses sequenced acknowledgment and is able to retransmit packets as needed.

The TCP header appears as follows:

The parts of the header are detailed in the following sections.

Source Port

The source port is the numerical value indicating the source port.

Destination Port

The destination port is the numerical value indicating the destination port.

Sequence Number

The sequence number is the number of the first data octet in any given segment.

Acknowledgement Number (ACK)

When the ACK bit is set, this field contains the next sequence number that the sender of the segment is expecting to receive. This value is always sent.

16-bit						32-bit					
Source Port						Destination Port					
Sequence Number											
Acknowledgement Number (ACK)											
Offset Reserved		U	A	P	R	S	F	Window			
Checksum						Urgent Pointer					
Options and Padding											

Data Offset

Data offset is the numerical value that indicates where the data begins, implying the end of the header by offset.

Reserved

Reserved is not used, but it must be off (0).

Control Bits

The control bits are as follows:

- U(URG) Urgent pointer field significant
- A(ACK) Acknowledgement field significant
- P(PSH) Push function
- R(RST) Reset connection
- S(SYN) Synchronize sequence numbers
- F(FIN) No more data

Window

The window indicates the number of octets the sender is willing to take. This starts with the packet in the ACK field.

Checksum

The checksum field is the 16-bit complement of the sum of all 16-bit words, restricted to the 1s column, in the header and text. If the result is an odd number of header and text octets, then the last octet is padded with zeros to form a 16-bit word that will checksum. Note that the pad is not sent as part of the segment.

Urgent (URG) Pointer

The URG pointer field shows the value of the URG pointer in the form of a positive offset of the sequence number from the octet that follows the URG data or points to the end of urgent data.

Options

Options may be sent at the end of a header, but must always be fully implemented and have a length that is any multiple of 8-bits. The two types of options are

- Type 1: A single octet of option-kind.
- Type 2: An octet of option-kind, an octet of option-length, and the actual option-data octets.

The option has a type, length and value field.

The length field includes the type, option length, as well as the value fields. The value field contains the data that makes up the options being sent. Table 3.1 shows how the octets are formatted in one of three classes. All the options are included in the checksum. An option can begin on any octet boundary as long as any remaining "dead" space is padded to meet the defined packet length.

NOTE

The list of options can be shorter than that designated by the data offset field because the contents of the header beyond the End-of-Option option must be padded with zeros (0).

Table Formatted Octets

Class	Length	Description
0	-	End of options
1	-	No operation
2	4	Maximum segment size

The Class 0 option indicates the end of the option list, ending all options, but not each individual option. Only use this option if the end of the options list does not coincide with the end of the header.

The Class 1 option is used to align an option with a word boundary. Other than that this option does not contain any useful information. Its main purpose is for aligning and formatting.

The Class 2 option indicates the maximum segment size that it will receive. It can only be included in the initial request and in segments where the SYN item bit is set. If this option is not used, there is no size limit.

Internet Protocol (IP)

IP manages how packets are delivered to and from servers and clients.

The IP header appears as follows:

4-bit	8-bit	16-bit	32-bit	
Ver.	Header Length	Type of Service	Total Length	
Identification			Flags	Offset
Time To Live	Protocol		Checksum	
Source Address				
Destination Address				
Options and Padding				

Each field contains information about the IP packet that it carries. The descriptions in the following sections should be helpful.

Version Number

The version number indicates the version of IP that is in use for this packet. IP version 4 (Ipv4) is currently in widespread use.

Header Length

The header length indicates the overall length of the header. The receiving machine then knows when to stop reading the header and start reading data.

Type of Service

Mostly unused, the Type of Service field indicates the importance of the packet in a numerical value. Higher numbers result in prioritized handling.

Total Length

Total length shows the total length of the packet in bytes. The total packet length cannot exceed 65,535 bytes or it will be deemed corrupt by the receiver.

Identification

If there is more than one packet (an invariable inevitability), the identification field has an identifier that identifies its place in line, as it were. Fragmented packets retain their original ID number.

Flags

The first flag, if set, is ignored. If the DF (Do Not Fragment) flag is set, under no circumstances can the packet be fragmented. If the MF (More Fragments) bit is turned on (1), there are packet fragments to come, the last of which is set to off (0).

Offset

If the Flag field returns a 1 (on), the Offset field contains the location of the missing piece(s) indicated by a numerical offset based on the total length of the packet.

Time To Live (TTL)

Typically 15 to 30 seconds, TTL indicates the length of time that a packet is allowed to remain in transit. If a packet is discarded or lost in transit, an indicator is sent back to the sending computer that the loss occurred. The sending machine then has the option of resending that packet.

Protocol

The protocol field holds a numerical value indicating the handling protocol in use for this packet.

Checksum

The checksum value acts as a validation checksum for the header.

Source Address

The source address field indicates the address of the sending machine.

Destination Address

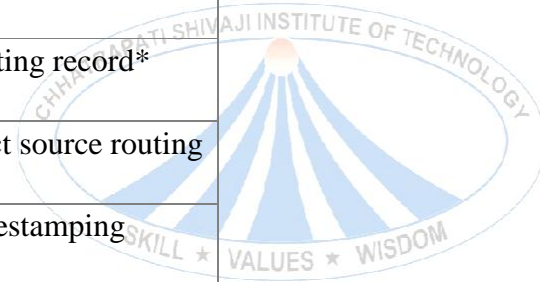
The destination address field indicates the address of the destination machine.

Options and Padding

The Options field is optional. If used, it contains codes that indicate the use of security, strict or loose source routing, routing records, and timestamping. If no options are used, the field is called *padded* and contains a 1. Padding is used to force a byte value that is rounded. Table 3.2 indicates the bit counts for the options available.

Table Bit Counts

Class	Number	Option
0	0	End of option list
0	2	Military security
0	3	Loose source routing
0	7	Routing record*
0	9	Strict source routing
2	4	Timestamping



**This option adds fields.*

TCP/IP provides its services via a "stack" of translation layers, which are called, of all things, TCP/IP. Because TCP and IP are separate protocols, they require a common environment for translation services. As mentioned earlier in this chapter, the TCP/IP stack has four layers as opposed to the OSI's seven layers. In a nutshell, they are

- Application
- Transport
- Network
- Link

Each of these is discussed in the following sections.

Application Layer

The Application Layer combines a few services that the OSI separates into three layers. These services are end user-related: authentication, data handling, and compression. This is where email, Web browsers, telnet clients, and other Internet applications get their connections.

Transport Layer

Again, unlike OSI, this layer is not responsible for guaranteeing the delivery of packets. Its primary responsibility is managing the transfer between the source and the destination. The OSI guarantees that packets are checked and, if they don't match, are dumped and re-requested from the source.

Network Layer

The Network Layer deals strictly with packet-routing management. This layer is well-suited to making determinations on where to send packets based on the information it receives.

Link Layer

The Link Layer manages the connection of the network and provides packet I/O over the network, but not at the application level.

Now that you have a clear idea of what TCP/IP is and what it can do (although hands-on experience is as close as your nearest Internet-connected computer), the following section moves on to the tangible benefits that TCP/IP provides you.

IP v4 address Structure

Internet Protocol version 4 (IPv4) is the fourth revision in the development of the Internet Protocol (IP) and it is the first version of the protocol to be widely deployed. Together with IPv6, it is at the core of standards-based internetworking methods of the Internet. IPv4 is still by far the most widely deployed Internet Layer protocol. As of 2010[update], IPv6 deployment is still in its infancy.

IPv4 is described in IETF publication RFC 791 (September 1981), replacing an earlier definition (RFC 760, January 1980).

IPv4 is a connectionless protocol for use on packet-switched Link Layer networks (e.g., Ethernet). It operates on a best effort delivery model, in that it does not guarantee delivery, nor does it assure proper sequencing, or avoid duplicate delivery. These aspects, including data integrity, are addressed by an upper layer transport protocol (e.g., Transmission Control Protocol).

Addressing

IPv4 uses 32-bit (four-byte) addresses, which limits the address space to 4,294,967,296 (2^{32}) possible unique addresses. However, some are reserved for special purposes such as private networks (~18 million addresses) or multicast addresses (~270 million addresses). This reduces the number of addresses that can potentially be allocated for routing on the public Internet. As addresses are being incrementally delegated to end users, an IPv4 address shortage has been developing, however network addressing architecture redesign via classful network design, Classless Inter-Domain Routing, and network address translation (NAT) has significantly delayed the inevitable exhaustion.

This limitation has stimulated the development of IPv6, which is currently in the early stages of deployment, and is the only long-term solution

Address representations

IPv4 addresses are usually written in dot-decimal notation, which consists of the four octets of the address expressed in decimal and separated by periods. This is the base format used in the conversion in the following table:

Notation	Value	Conversion from dot-decimal
Dot-decimal notation	192.0.2.235	N/A
Dotted Hexadecimal	0xC0.0x00.0x02.0xEB	Each octet is individually converted to hexadecimal form
Dotted Octal	0300.0000.0002.0353	Each octet is individually converted into octal
Hexadecimal	0xC00002EB	Concatenation of the octets from the dotted hexadecimal
Decimal	3221226219	The 32-bit number expressed in decimal
Octal	030000001353	The 32-bit number expressed in octal

Most of these formats should work in all browsers. Additionally, in dotted format, each octet can be of any of the different bases. For example, 192.0x00.0002.235 is a valid (though unconventional) equivalent to the above addresses.

A final form is not really a notation since it is rarely written in an ASCII string notation. That form is a binary form of the hexadecimal notation in binary. This difference is merely the representational difference between the string "0xCF8E83EB" and the 32-bit integer value 0xCF8E83EB. This form is used for assigning the source and destination fields in a software program.

IPv6 address structure

Internet Protocol version 6 (IPv6) is the next-generation Internet Protocol version designated as the successor to Ipv4.

IPv6 has a vastly larger address space than IPv4. This results from the use of a 128-bit address, whereas IPv4 uses only 32 bits. The new address space thus supports 2¹²⁸ (about 3.4×10³⁸) addresses. This expansion provides flexibility in allocating addresses and routing traffic and eliminates the primary need for network address translation (NAT), which gained widespread deployment as an effort to alleviate IPv4 address exhaustion.

IPv6 also implements new features that simplify aspects of address assignment (stateless address autoconfiguration) and network renumbering (prefix and router announcements) when changing Internet connectivity providers. The IPv6 subnet size has been standardized by fixing the size of the host identifier portion of an address to 64 bits to facilitate an automatic mechanism for forming the host identifier from Link Layer media addressing information (MAC address).

Network security is integrated into the design of the IPv6 architecture. Internet Protocol Security (IPsec) was originally developed for IPv6, but found widespread optional deployment first in IPv4 (into which it was back-engineered). The IPv6 specifications mandate IPsec implementation as a fundamental interoperability requirement.

In December 2008, despite marking its 10th anniversary as a Standards Track protocol, IPv6 was only in its infancy in terms of general worldwide deployment. A 2008 study[1] by Google Inc. indicated that penetration was still less than one percent of Internet-enabled hosts in any country. IPv6 has been implemented on all major operating systems in use in commercial, business, and home consumer environments.

Features and differences from Ipv4

In most regards, IPv6 is a conservative extension of IPv4. Most transport- and application-layer protocols need little or no change to operate over IPv6; exceptions are application protocols that embed internet-layer addresses, such as FTP or NTPv3.

IPv6 specifies a new packet format, designed to minimize packet-header processing. Since the headers of IPv4 packets and IPv6 packets are significantly different, the two protocols are not interoperable.

Packet format

The IPv6 packet is composed of three main parts: the fixed header, optional extension headers and the payload.

The fixed header makes up the first 40 octets (320 bits) of an IPv6 data packet. The header contains the source and destination address, traffic classification options, a hop counter, and an indication of the next header. The *Next Header* field points to a chain of zero or more extension headers (chained by *Next Header* fields); the last *Next Header* field points to the upper-layer protocol that is carried in the packet's payload.

Extension headers carry options that are used for special treatment of a packet along the way or at its destination, routing, fragmenting, and for security using the IPsec framework.

The payload can have a size of up to 64 KB in standard mode, or larger with a "jumbo payload" option in a *Hop-By-Hop Options* extension header.

Fragmentation is handled only in the sending host in IPv6: routers never fragment a packet, and hosts are expected to use Path MTU discovery.

Addressing

The increased length of network addresses emphasizes a most important change when moving from IPv4 to IPv6. IPv6 addresses are 128 bits long[12], whereas IPv4 addresses are 32 bits; where the IPv4 address space contains roughly 4.3×10^9 (4.3 billion) addresses, IPv6 has enough room for 3.4×10^{38} (340 trillion trillion trillion) unique addresses.

IPv6 addresses are normally written with hexadecimal digits and colon separators like 2001:db8:85a3::8a2e:370:7334, as opposed to the dot-decimal notation of the 32 bit IPv4 addresses. IPv6 addresses are typically composed of two logical parts: a 64-bit (sub-)network prefix, and a 64-bit host part.

IPv6 addresses are classified into three types: unicast addresses which uniquely identify network

interfaces, anycast addresses which identify a group of interfaces—mostly at different locations—for which traffic flows to the nearest one, and multicast addresses which are used to deliver one packet to many interfaces. Broadcast addresses are not used in IPv6. Each IPv6 address also has a 'scope', which specifies in which part of the network it is valid and unique. Some addresses have node scope or link scope; most addresses have global scope (i.e. they are unique globally).

Some IPv6 addresses are used for special purposes, like the loopback address. Also, some address ranges are considered special, like link-local addresses (for use in the local network only) and solicited-node multicast addresses (used in the Neighbor Discovery Protocol).

Programming Applications: Date time Routines

The **C Standard Library** is the standard library for the C programming language, as specified in the ANSI C standard.^[1] It was developed at the same time as the C POSIX library, which is basically a superset of it^[citation needed]. Since ANSI C was adopted by the International Organization for Standardization,^[2] the C standard library is also called the **ISO C library**.

Informally, the terms **C standard library** or **C library** or **libc** are also used to designate a particular implementation on a given system. In the Unix environment, such an implementation is usually shipped with the operating system and its presence is assumed by many applications. For instance, GNU/Linux comes with the GNU implementation glibc.

The C standard library provides macros, type definitions, and functions for tasks like string handling, mathematical computations, input/output processing, memory allocation and several other operating system services.

Header files

The application programming interface (API) of the C standard library is declared in a number of header files. Each header file contains one or more function declarations, data type definitions, and macros.

After a long period of stability, three new header files (iso646.h, wchar.h, and wctype.h) were added with *Normative Addendum 1* (NA1), an addition to the C Standard ratified in 1995. Six more header files (complex.h, fenv.h, inttypes.h, stdbool.h, stdint.h, and tgmath.h) were added with C99, a revision to the C Standard published in 1999, and five more files (stdalign.h, stdatomic.h, stdnoreturn.h, threads.h, and uchar.h) with C11 in 2011. In total, there are now 29 header files:



Three of the header files (`complex.h`, `stdatomic.h`, `threads.h`) are conditional features that implementations need not support.

The POSIX standard added several nonstandard C headers for Unix-specific functionality. Many have found their way to other architectures. Examples include `unistd.h` and `signal.h`. A number of other groups are using other nonstandard headers - most flavors of Linux have `alloca.h` and HP OpenVMS has the `va_count()` function.

Application Layer in Internet protocol

Application Layer (Layer 7)

At the very top of the OSI Reference Model stack of layers, we find layer 7, the *application layer*. Continuing the trend that we saw in layers 5 and 6, this one too is named very appropriately: the application layer is the one that is used by network applications. These programs are what actually implement the functions performed by users to accomplish various tasks over the network.

It's important to understand that what the OSI model calls an “application” is not exactly the same as what we normally think of as an “application”. In the OSI model, the application layer provides services for user applications to employ. For example, when you use your Web browser, that actual software is an application running on your PC. It doesn't really “reside” at the application layer. Rather, it makes use of the services offered by a protocol that operates at the application layer, which is called the Hypertext Transfer Protocol (HTTP). The distinction between the browser and HTTP is subtle, but important.

The reason for pointing this out is because not all user applications use the application layer of the network in the same way. Sure, your Web browser does, and so does your e-mail client and your Usenet news reader. But if you use a text editor to open a file on another machine on your network, that editor is not using the application layer. In fact, it has no clue that the file you are

using is on the network: it just sees a file addressed with a name that has been mapped to a network somewhere else. The operating system takes care of *redirecting* what the editor does, over the network.

Similarly, not all uses of the application layer are by applications. The operating system itself can (and does) use services directly at the application layer.

That caveat aside, under normal circumstances, whenever you interact with a program on your computer that is designed specifically for use on a network, you are dealing directly with the application layer. For example, sending an e-mail, firing up a Web browser, or using an IRC chat program—all of these involve protocols that reside at the application layer.

There are dozens of different application layer protocols that enable various functions at this layer. Some of the most popular ones include HTTP, FTP, SMTP, DHCP, NFS, Telnet, SNMP, POP3, NNTP and IRC. Lots of alphabet soup, sorry.

As the “top of the stack” layer, the application layer is the only one that does not provide any services to the layer above it in the stack—there isn't one! Instead, it provides services to programs that want to use the network, and to you, the user. So the responsibilities at this layer are simply to implement the functions that are needed by users of the network. And, of course, to issue the appropriate commands to make use of the services provided by the lower layers.

Presentation Layer (Layer 6)

The *presentation layer* is the sixth layer of the OSI Reference Model protocol stack, and second from the top. It is different from the other layers in two key respects. First, it has a much more limited and specific function than the other layers; it's actually somewhat easy to describe, hurray! Second, it is used much less often than the other layers; in many types of connections it is not required.

The name of this layer suggests its main function as well: it deals with the *presentation* of data. More specifically, the presentation layer is charged with taking care of any issues that might arise where data sent from one system needs to be viewed in a different way by the other system. It also takes care of any special processing that must be done to data from the time an application tries to send it until the time it is sent over the network.

Presentation Layer Functions

Here are some of the specific types of data handling issues that the presentation layer handles:

- **Translation:** Networks can connect very different types of computers together: PCs, Macintoshes, UNIX systems, AS/400 servers and mainframes can all exist on the same network. These systems have many distinct characteristics and represent data in different ways; they may use different character sets for example. The presentation layer handles the job of hiding these differences between machines.
- **Compression:** Compression (and decompression) may be done at the presentation layer to improve the throughput of data. (There are some who believe this is not, strictly speaking, a function of the presentation layer.)
- **Encryption:** Some types of encryption (and decryption) are performed at the presentation layer. This ensures the security of the data as it travels down the protocol stack. For example, one of the most popular encryption schemes that is usually associated with the

presentation layer is the Secure Sockets Layer (SSL) protocol. Not all encryption is done at layer 6, however; some encryption is often done at lower layers in the protocol stack, in technologies such as IPSec.

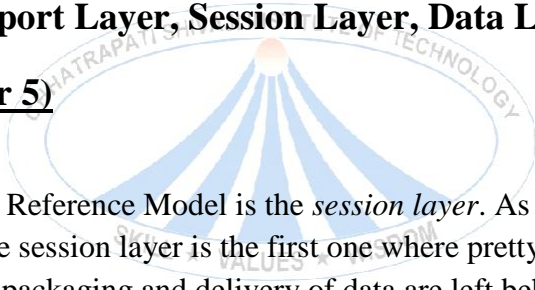
Presentation Layer Role in the OSI Model

The reason that the presentation layer is not always used in network communications is that the jobs mentioned above are simply not always needed. Compression and encryption are usually considered “optional”, and translation features are also only needed in certain circumstances. Another reason why the presentation layer is sometimes not mentioned is that its functions may be performed as part of the application layer.

The fact that the translation job done by the presentation layer isn't always needed means that it is common for it to be “skipped” by actual protocol stack implementations. This means that protocols at layer seven may talk directly with those at layer five. Once again, this is part of the reason why all of the functions of layers five through seven may be included together in the same software package, as described in the overview of layers and layer groupings.

Transport Layer, Session Layer, Data Link Layer

Session Layer (Layer 5)



The fifth layer in the OSI Reference Model is the *session layer*. As we proceed up the OSI layer stack from the bottom, the session layer is the first one where pretty much all practical matters related to the addressing, packaging and delivery of data are left behind—they are functions of layers four and below. It is the lowest of the three upper layers, which collectively are concerned mainly with software application issues and not with the details of network and internet implementation.

The name of this layer tells you much about what it is designed to do: to allow devices to establish and manage *sessions*. In general terms, a session is a persistent logical linking of two software application processes, to allow them to exchange data over a prolonged period of time. In some discussions, these sessions are called *dialogs*; they are roughly analogous to a telephone call made between two people.

Application Program Interfaces (APIs)

The primary job of session layer protocols is to provide the means necessary to set up, manage, and end sessions. In fact, in some ways, session layer software products are more sets of tools than specific protocols. These session-layer tools are normally provided to higher layer protocols through command sets often called *application program interfaces* or *APIs*.

Common APIs include NetBIOS, TCP/IP Sockets and Remote Procedure Calls (RPCs). They allow an application to accomplish certain high-level communications over the network easily, by using a standardized set of services. Most of these session-layer tools are of primary interest to the developers of application software. The programmers use the APIs to write software that is

able to communicate using TCP/IP without having to know the implementation details of how TCP/IP works.

For example, the Sockets interface lies conceptually at layer five and is used by TCP/IP application programmers to create sessions between software programs over the Internet on the UNIX operating system. Windows Sockets similarly lets programmers create Windows software that is Internet-capable and able to interact easily with other software that uses that interface. (Strictly speaking, Sockets is not a protocol, but rather a programming method.)

Transport Layer (Layer4)

The fourth and “middle” layer of the OSI Reference Model protocol stack is the *transport layer*. I consider the transport layer in some ways to be part of both the lower and upper “groups” of layers in the OSI model. It is more often associated with the lower layers, because it concerns itself with the *transport* of data, but its functions are also somewhat high-level, resulting in the layer having a fair bit in common with layers 5 through 7 as well.

Recall that layers 1, 2 and 3 are concerned with the actual packaging, addressing, routing and delivery of data; the physical layer handles the bits; the data link layer deals with local networks and the network layer handles routing between networks. The transport layer, in contrast, is sufficiently conceptual that it no longer concerns itself with these “nuts and bolts” matters. It relies on the lower layers to handle the process of moving data between devices.

The transport layer really acts as a “liaison” of sorts between the abstract world of applications at the higher layers, and the concrete functions of layers one to three. Due to this role, the transport layer’s overall job is to provide the necessary functions to enable communication between software application processes on different computer.

This encompasses a number of different but related duties

Modern computers are multitasking, and at any given time may have many different software applications all trying to send and receive data. The transport layer is charged with providing a means by which these applications can all send and receive data using the same lower-layer protocol implementation. Thus, the transport layer is sometimes said to be responsible for *end-to-end* or *host-to-host* transport (in fact, the equivalent layer in the TCP/IP model is called the “host-to-host transport layer”).

Transport Layer Services and Transmission Quality

Accomplishing this communication between processes requires that the transport layer perform several different, but related jobs. For transmission, the transport layer protocol must keep track of what data comes from each application, then combine this data into a single flow of data to send to the lower layers. The device receiving information must reverse these operations, splitting data and funneling it to the appropriate recipient processes. The transport layer is also responsible for defining the means by which potentially large amounts of application data are divided into smaller blocks for transmission.

Another key function of the transport layer is to provide *connection services* for the protocols and applications that run at the levels above it. These can be categorized as either connection-

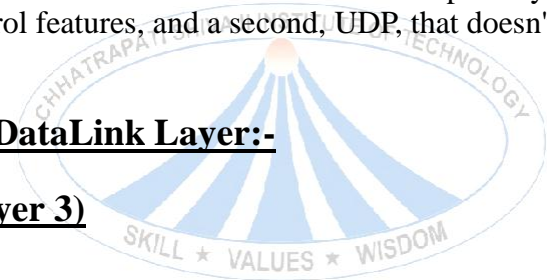
oriented services or connectionless services. Neither is better or worse than the other; they each have their uses. While connection-oriented services can be handled at the network layer as well, they are more often seen in the transport layer in the “real world”. Some protocol suites, such as TCP/IP, provide both a connection-oriented and a connectionless transport layer protocol, to suit the needs of different applications.

The transport layer is also the place in the layer stack where functions are normally included to add features to end-to-end data transport. Where network layer protocols are normally concerned with just “best effort” communications, where delivery is not guaranteed. Transport layer protocols are given intelligence in the form of algorithms that ensure that reliable and efficient communication between devices takes place. This encompasses several related jobs, including lost transmission detection and handling, and managing the rate at which data is sent to ensure that the receiving device is not overwhelmed.

Transmission quality, meaning ensuring that transmissions are received as sent, is so important that some networking references define the transport layer on the basis of reliability and flow-control functions. However, not all transport layer protocols provide these services. Just as a protocol suite may have a connection-oriented and a connectionless transport layer protocol, it may also have one that provides reliability and data management services, and one that does not. Again, this is the case with TCP/IP: there is one main transport layer protocol, TCP that includes reliability and flow control features, and a second, UDP, that doesn't.

Network layer and DataLink Layer:-

Network Layer (Layer 3)



The third-lowest layer of the OSI Reference Model is the *network layer*. If the data link layer is the one that basically defines the boundaries of what is considered a network, the network layer is the one that defines how *internetworks* (interconnected networks) function. The network layer is the lowest one in the OSI model that is concerned with actually getting data from one computer to another even if it is on a remote network; in contrast, the data link layer only deals with devices that are local to each other.

While all of layers 2 through 6 in the OSI Reference Model serve to act as “fences” between the layers below them and the layers above them, the network layer is particularly important in this regard. It is at this layer that the transition really begins from the more abstract functions of the higher layers—which don't concern themselves as much with data delivery—into the specific tasks required to get data to its destination. The transport layer, which is related to the network layer in a number of ways, continues this “abstraction transition” as you go up the OSI protocol stack.

Network Layer Functions

Some of the specific jobs normally performed by the network layer include:

- **Logical Addressing:** Every device that communicates over a network has associated with it a logical address, sometimes called a *layer three* address. For example, on the Internet,

the Internet Protocol (IP) is the network layer protocol and every machine has an IP address. Note that addressing is done at the data link layer as well, but those addresses refer to local physical devices. In contrast, logical addresses are independent of particular hardware and must be unique across an entire internetwork.

- **Routing:** Moving data across a series of interconnected networks is probably the defining function of the network layer. It is the job of the devices and software routines that function at the network layer to handle incoming packets from various sources, determine their final destination, and then figure out where they need to be sent to get them where they are supposed to go. I discuss routing in the OSI model more completely in this topic on the topic on indirect device connection, and show how it works by way of an OSI model analogy.
- **Datagram Encapsulation:** The network layer normally encapsulates messages received from higher layers by placing them into *datagrams* (also called *packets*) with a network layer header.
- **Fragmentation and Reassembly:** The network layer must send messages down to the data link layer for transmission. Some data link layer technologies have limits on the length of any message that can be sent. If the packet that the network layer wants to send is too large, the network layer must split the packet up, send each piece to the data link layer, and then have pieces reassembled once they arrive at the network layer on the destination machine. A good example is how this is done by the Internet Protocol.
- **Error Handling and Diagnostics:** Special protocols are used at the network layer to allow devices that are logically connected, or that are trying to route traffic, to exchange information about the status of hosts on the network or the devices themselves.

Data Link Layer (Layer 2)

The second-lowest layer (layer 2) in the OSI Reference Model stack is the *data link layer*, often abbreviated “DLL” (though that abbreviation has other meanings as well in the computer world). The data link layer, also sometimes just called the *link layer*, is where many wired and wireless local area networking (LAN) technologies primarily function. For example, Ethernet, Token Ring, FDDI and 802.11 (“wireless Ethernet” or “Wi-Fi”) are all sometimes called “data link layer technologies”. The set of devices connected at the data link layer is what is commonly considered a simple “network”, as opposed to an internetwork.

Data Link Layer Sublayers: Logical Link Control (LLC) and Media Access Control (MAC)

The data link layer is often conceptually divided into two sublayers: *logical link control (LLC)* and *media access control (MAC)*. This split is based on the architecture used in the IEEE 802 Project, which is the IEEE working group responsible for creating the standards that define many networking technologies (including all of the ones I mentioned above except FDDI). By separating LLC and MAC functions, interoperability of different network technologies is made easier, as explained in our earlier discussion of networking model concepts.

Data Link Layer Functions

The following are the key tasks performed at the data link layer:

- **Logical Link Control (LLC):** Logical link control refers to the functions required for the establishment and control of logical links between local devices on a network. As mentioned above, this is usually considered a DLL sublayer; it provides services to the network layer above it and hides the rest of the details of the data link layer to allow different technologies to work seamlessly with the higher layers. Most local area networking technologies use the IEEE 802.2 LLC protocol.
- **Media Access Control (MAC):** This refers to the procedures used by devices to control access to the network medium. Since many networks use a shared medium (such as a single network cable, or a series of cables that are electrically connected into a single virtual medium) it is necessary to have rules for managing the medium to avoid conflicts. For example, Ethernet uses the CSMA/CD method of media access control, while Token Ring uses token passing.
- **Data Framing:** The data link layer is responsible for the final encapsulation of higher-level messages into *frames* that are sent over the network at the physical layer.
- **Addressing:** The data link layer is the lowest layer in the OSI model that is concerned with addressing: labeling information with a particular destination location. Each device on a network has a unique number, usually called a *hardware address* or *MAC address*, that is used by the data link layer protocol to ensure that data intended for a specific machine gets to it properly.
- **Error Detection and Handling:** The data link layer handles errors that occur at the lower levels of the network stack. For example, a cyclic redundancy check (CRC) field is often employed to allow the station receiving data to detect if it was received correct.

Unit-2

Creating Sockets

Creating Sockets

What are Sockets and Threads?

A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. The socket associates the server program with a specific hardware port on the machine where it runs so any client program anywhere in the network with a socket associated with that same port can communicate with the server program.

A server program typically provides resources to a network of client programs. Client programs send requests to the server program, and the server program responds to the request.

One way to handle requests from more than one client is to make the server program multi-threaded. A multi-threaded server creates a thread for each communication it accepts from a

client. A thread is a sequence of instructions that run independently of the program and of any other threads.

Using threads, a multi-threaded server program can accept a connection from a client, start a thread for that communication, and continue listening for requests from other clients.

Server-Side Program

The server program establishes a socket connection on Port 4321 in its listenSocket method. It reads data sent to it and sends that same data back to the server in its actionPerformed method.

listenSocket Method

The listenSocket method creates a ServerSocket object with the port number on which the server program is going to listen for client communications. The port number must be an available port, which means the number cannot be reserved or already in use. For example, Unix systems reserve ports 1 through 1023 for administrative functions leaving port numbers greater than 1024 available for use.

Next, the listenSocket method creates a Socket connection for the requesting client. This code executes when a client starts up and requests the connection on the host and port where this server program is running. When the connection is successfully established, the server.accept method returns a new Socket object.

Posix data Type

POSIX Threads, usually referred to as **Pthreads**, is a POSIX standard for threads. The standard, *POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)*, defines an API for creating and manipulating threads.

Implementations of the API are available on many Unix-like POSIX-conformant operating systems such as FreeBSD, NetBSD, OpenBSD, GNU/Linux, Mac OS X and Solaris. DR-DOS and Microsoft Windows implementations also exist: within the SFU/SUA subsystem which provides a native implementation of a number of POSIX APIs, and also within third-party packages such as *pthreads-w32*,^[1] which implements pthreads on top of existing Windows API.

Posix data Type

data ByteCount -- instances of : Eq Ord Num Real Integral Ix Enum Show

A 'ByteCount' is a primitive of type 'unsigned'. At a minimum, an conforming implementation must support values in the range '[0, UINT_MAX]'.

data ClockTick -- instances of : Eq Ord Num Real Integral Ix Enum Show

A 'ClockTick' is a primitive of type 'clock_t', which is used to measure intervals of time in fractions of a second. The resolution is determined by 'getSysVar ClockTick'.

data DeviceID -- instances of : Eq Ord Num Real Integral Ix Enum Show

A `DeviceID` is a primitive of type `dev_t`. It must be an arithmetic type.

data EpochTime -- instances of : Eq Ord Num Real Integral Ix Enum Show

A `EpochTime` is a primitive of type `time_t`, which is used to measure seconds since the Epoch. At a minimum, the implementation must support values in the range `[0, INT_MAX]`.

data FileID -- instances of : Eq Ord Num Real Integral Ix Enum Show

A `FileID` is a primitive of type `ino_t`. It must be an arithmetic type.

data FileMode -- instances of : Eq Ord Num Real Integral Ix Enum Show

A `FileMode` is a primitive of type `mode_t`. It must be an arithmetic type.

data FileOffset -- instances of : Eq Ord Num Real Integral Ix Enum Show

Socket Address

A **socket address** is the combination of an IP address and a port number, much like one end of a telephone connection is the combination of a phone number and a particular extension. Based on this address, internet sockets deliver incoming data packets to the appropriate application process or thread.

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 IETF Request 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 made accessible to the application.

There are also non-Internet sockets, implemented over other transport protocols, such as Systems Network Architecture (SNA).^[2] 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.

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.

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).

For further details on TCP sockets, including other states of TCP sockets, see Transmission Control Protocol.

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-tuple consisting of source and destination IP addresses and port numbers, i.e. of local and remote socket addresses. 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.

Java Socket Programming

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols:

- **TCP:** TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP:** UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This tutorial gives good understanding on the following two subjects:

1. **Socket Programming:** This is most widely used concept in Networking and it has been explained in very detail.
2. **URL Processing:** This would be covered separately. Click here to learn about URL Processing in Java language.

Socket Programming:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The `java.net.Socket` class represents a socket, and the `java.net.ServerSocket` class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

1. The server instantiates a `ServerSocket` object, denoting which port number communication is to occur on.
2. The server invokes the `accept()` method of the `ServerSocket` class. This method waits until a client connects to the server on the given port.
3. After the server is waiting, a client instantiates a `Socket` object, specifying the server name and port number to connect to.
4. The constructor of the `Socket` class attempts to connect the client to the specified server and port number. If communication is established, the client now has a `Socket` object capable of communicating with the server.
5. On the server side, the `accept()` method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an `OutputStream` and an `InputStream`. The client's `OutputStream` is connected to the server's `InputStream`, and the client's `InputStream` is connected to the server's `OutputStream`.

TCP is a twoway communication protocol, so data can be sent across both streams at the same time. There are following usefull classes providing complete set of methods to implement sockets.

ServerSocket Class Methods:

The **`java.net.ServerSocket`** class is used by server applications to obtain a port and listen for client requests

The `ServerSocket` class has four constructors:

SN	Methods with Description
1	<code>public ServerSocket(int port) throws IOException</code> Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	<code>public ServerSocket(int port, int backlog) throws IOException</code> Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.

3	public ServerSocket(int port, int backlog, InetAddress address) throws IOException Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on
4	public ServerSocket() throws IOException Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Lecture No:19

Thread Programming

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

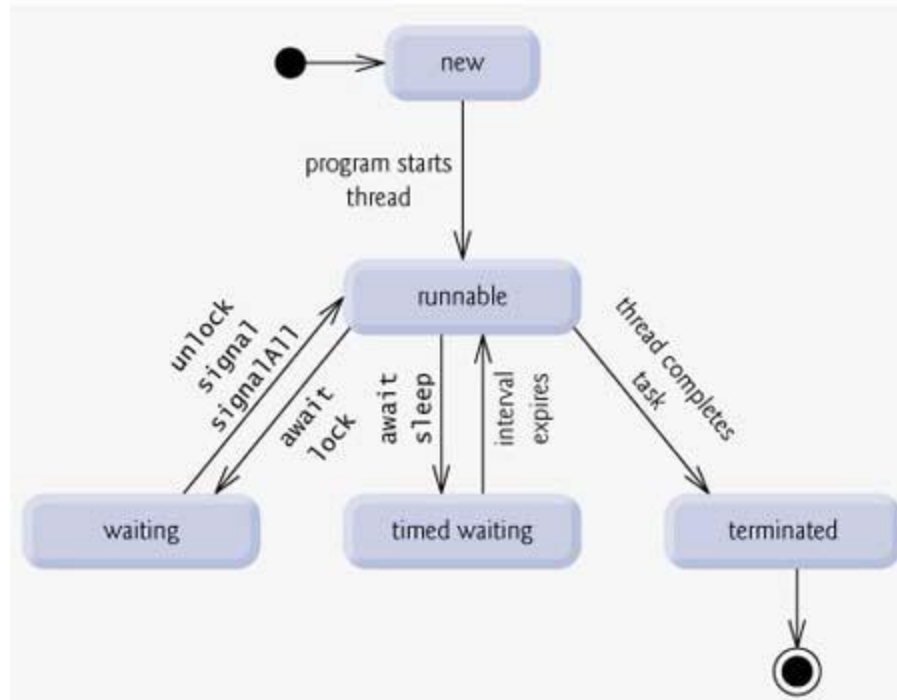
A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.

I need to define another term related to threads: **process**: A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Creating a Thread:

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable, a class need only implement a single method called **run()**, which is declared like this:

```
public void run()
```

You will define the code that constitutes the new thread inside **run()** method. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName);
```

Here *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start()** method, which is declared within Thread. The **start()** method is shown here:

```
void start();
```

Example:

Here is an example that creates a new thread and starts it running:

```
// Create a new thread.  
class NewThread implements Runnable {  
    Thread t;
```

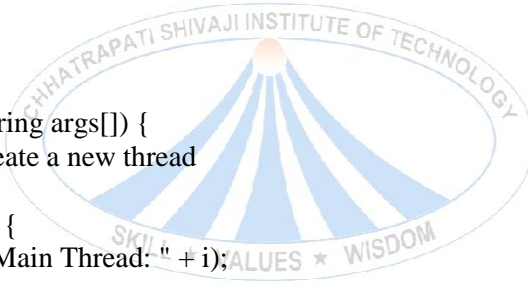
```

NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
}

// This is the entry point for the second thread.
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            // Let the thread sleep for a while.
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
}
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```



This would produce following result:

```

Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2

```

Main Thread: 1
Main thread exiting.

Create Thread by Extending Thread:

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.

The extending class must override the **run()** method, which is the entry point for the new thread. It must also call **start()** to begin execution of the new thread.

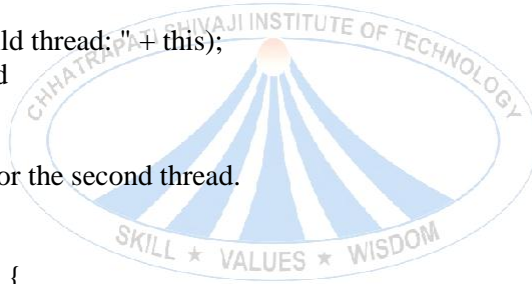
Example:

Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                // Let the thread sleep for a while.
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
    }
}
```



```

    }
    System.out.println("Main thread exiting.");
}
}

```

This would produce following result:

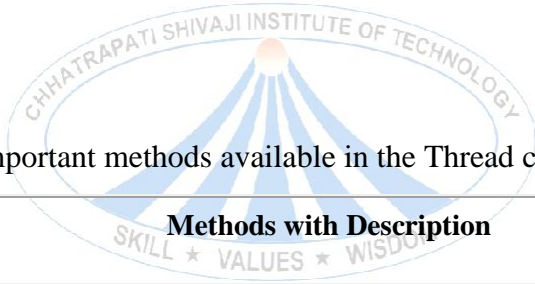
```

Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.

```

Thread Methods:

Following is the list of important methods available in the Thread class.



SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.

6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread

Berkeley Sockets: Overview

Berkeley sockets (or **BSD sockets**) is a computing library with an application programming interface (API) for internet sockets and Unix domain sockets, used for inter-process communication (IPC).

As the API has evolved with little modification from a *de facto* standard into part of the POSIX specification, **POSIX sockets** are basically Berkeley sockets.

Berkeley sockets originated with the 4.2BSD Unix operating system (released in 1983) as an API. Only in 1989, however, could UC Berkeley release versions of its operating system and networking library free from the licensing constraints of AT&T's proprietary Unix. This interface implementation is the original API of the Internet Protocol Suite (TCP/IP).

All modern operating systems now have some implementation of the Berkeley socket interface, as it became the standard interface for connecting to the Internet. Even the Winsock implementation for MS Windows, developed by unaffiliated developers, closely follows the Berkeley standard.

BSD vs POSIX

As the Berkeley socket API evolved over time, and ultimately into the POSIX socket API,^[1] certain functions were deprecated or even removed and replaced by others. The POSIX API is also designed to be reentrant. These features now set the classic BSD API from the POSIX API apart.

Action

BSD

POSIX

Conversion from text address to packed address	inet_aton	inet_pton
Conversion from packed address to text address	inet_ntoa	inet_ntop
Forward lookup for host name/service	gethostbyname, gethostbyaddr, getservbyname, getservbyport	getaddrinfo
Reverse lookup for host name/service	gethostbyaddr, getservbyport	getnameinfo

C and other programming languages

The BSD sockets API is written in the programming language C. Most other programming languages provide similar interfaces, typically written as a wrapper library based on the C API.^[2]

Alternatives

The STREAMS-based Transport Layer Interface (TLI) API offers an alternative to the socket API. However, recent systems that provide the TLI API also provide the Berkeley socket API.

Header files

The Berkeley socket interface is defined in several header files. The names and content of these files differ slightly between implementations. In general, they include:

<sys/socket.h>

Core BSD socket functions and data structures.

<netinet/in.h>

AF_INET and AF_INET6 address families and their corresponding protocol families PF_INET and PF_INET6. Widely used on the Internet, these include IP addresses and TCP and UDP port numbers.

<sys/un.h>

PF_UNIX/PF_LOCAL address family. Used for local communication between programs running on the same computer. Not used on networks.

<arpa/inet.h>

Functions for manipulating numeric IP addresses.

<netdb.h>

Functions for translating protocol names and host names into numeric addresses. Searches local data as well as DNS.

Socket Address Structures

There are various structures which are used in Unix Socket Programming to hold information about the address and port and other information. Most socket functions require a pointer to a socket address structure as an argument. Structures defined in this tutorial are related to Internet Protocol Family.

The first structure is struct *sockaddr* that holds socket information:

```
struct sockaddr{
    unsigned short  sa_family;
    char            sa_data[14];
};
```

This is a generic socket address structure which will be passed in most of the socket function calls. Here is the description of the member fields:

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	This represents an address family. In most of the Internet based applications we use AF_INET.
sa_data	Protocol Specific Address	The content of the 14 bytes of protocol specific address are interpreted according to the type of address. For the Internet family we will use port number IP address which is represented by <i>sockaddr_in</i> structure defined below.

Second structure that helps you to reference to the socket's elements is as follows:

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

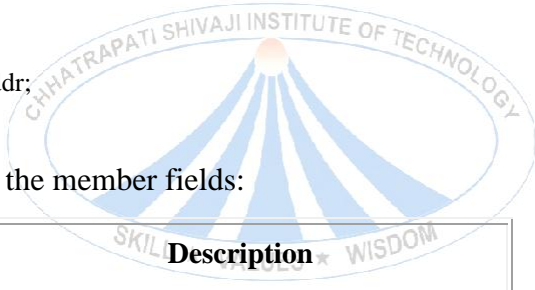
Here is the description of the member fields:

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	This represents an address family. In most of the Internet based applications we use AF_INET.
sin_port	Service Port	A 16 bit port number in Network Byte Order.
sin_addr	IP Address	A 32 bit IP address in Network Byte Order.
sin_zero	Not Used	You just set this value to NULL as this is not being used.

The next structure is used only in the above structure as a structure field and holds 32 bit netid/hostid.

```
struct in_addr {
    unsigned long s_addr;
};
```

Here is the description of the member fields:



Attribute	Values	Description
s_addr	service port	A 32 bit IP address in Network Byte Order.

Tips on Socket Structures:

Socket address structures are an integral part of every network program. We allocate them, fill them in, and pass pointers to them to various socket functions. Sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents.

We always pass these structures by reference (ie. we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument.

When a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function. We call these value-result arguments.

Always set the structure variables to NULL (i.e. '\0') by using memset() or bzero() functions otherwise it may get unexpected junk values in your structure.

Byte Manipulation & Address Conversion Functions

Elementary Socket System Calls – Socket, Connect

Elementary system calls

This section describes, in the context of the client-server model, the socket system calls needed to create and connect a pair of sockets and transmit data.

A socket created with the **socket** call is identified only by its address family. It has no name.

Without a name, processes cannot reference it and no communication can take place.

Communication processes are bound by an association. In the Internet domain, as mentioned earlier, an association consists of a local address and port number, and a foreign address and port number. The **bind**(SSC) system call allows a process to specify the local half of this association by binding an address to the local end of a socket.

The call takes the form:

bind(*s*, *name*, *namelen*);

Arguments to **bind** include the socket, a pointer to the address (*name*), and the length of the address (*namelen*). With this information, a process is able to rendezvous with an unrelated process. The operation is asymmetric, with one process taking the role of server and the other of client.

Server (Internet domain)

Server sockets listen passively for incoming connection requests rather than actively initiating them. To create a server socket, call **listen**(SSC) after the socket is bound.

listen(*s*, *backlog*);

The **listen** call identifies the socket that receives the connections and marks it as accepting connections. The maximum queue length (*backlog*) that **listen** can specify is five.

A call to **listen** does not complete the connection; it only indicates a willingness on the part of the server process to listen for incoming connection requests. Applications must call **accept**(SSC) to establish an actual connection:

accept(*s, addr, addrlen*)

The ***addr*** and ***addrlen*** arguments return the address of the connected client process, and **accept** automatically creates a new socket descriptor. Assuming the server is concurrent, the process immediately forks and the child process handles the connection while the parent continues to listen for more connection requests.

Client (Internet domain)

Client sockets actively initiate calls, rather than listening passively for incoming connection requests. Active sockets use the **connect(SSC)** function to establish connections. To create a socket to rendezvous with a server, the **socket** call is used as before. With streams sockets, it is not necessary to bind an address to the client (see table below). This is because with a connection-oriented protocol the server's address must be known, but the client's address is unimportant because no other process will try to access it. With connectionless sockets, however, each datagram must contain the destination address of the client and an address bound to it with the **bind** system call.

A client process issues the **connect** call in an attempt to establish a connection with a listening server process. A connection is a mechanism that avoids the need to transmit the identity of the sending socket each time data is sent.

The **connect** call provides for the exchange of the identities of each endpoint just once, prior to the transmission of data. Successful completion of **connect** results in a connection between two sockets.

connect(*s, name, namelen*)

connect does not return until a connection is established. *s* is the socket descriptor of the client process. ***name*** and ***namelen*** are pointers to the address family and size of the other socket.

Establishing a connection

Client	Server	
socket()	socket()	
	bind()	
	listen()	
connect()	accept()	

Data transfer (Internet domain)

Once a pair of sockets is connected, data can flow from one to the other. The **read(S)** and **write(C)** functions may be used to do this, just as they are for normal files.

read(*fildes*, *buf*, *nbyte*)

write(*fildes*, *buf*, *nbyte*)

fildes refers to the socket descriptor returned by the **socket** call in the client, and by **accept** in the server. The **write** system call attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the socket associated with *fildes*.

The **send(SSC)** and **recv(SSC)** calls are almost identical to **read** and **write**, except that they provide a *flags* argument.

send(*s*, *msg*, *len*, *flags*)

recv(*s*, *msg*, *len*, *flags*)

Applications using TCP use flags to signal the presence of urgent data. The *flags* argument may be specified as non-zero if one or more of the following constants is required:

MSG_OOB out-of-band data

MSG_PEEK look at data without reading

MSG_DONTROUTE send data without routing

Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independent from normal data, along with a signal. For a more complete discussion of out-of-band data, see the *SCO TCP/IP Programmer's Guide*.

The **MSG_PEEK** flag lets the caller look at the data without discarding it as read. When specified in a **recv** call, any data present is returned to the user but treated as though still unread. The next **read** or **recv** call applied to the socket will return the data previously previewed.

The **MSG_DONTROUTE** option specifies that normal routing mechanisms are bypassed and the message is directed to the network interface specified in the network portion of the destination address.

The **close(S)** system call is used to discard the end of a socket connection when it is no longer needed. If one end of a socket is closed and the other tries to write to it, the **write** will return an error. *s* is the socket descriptor being closed in:

close(s)

If the socket is a stream socket, the system will continue to attempt to transfer data. The **shutdown(SSC)** call causes all or part of a full-duplex connection to be terminated:

shutdown(s, how)

If argument **how** is 0, further receives are disabled. When **how** is 1, further sends are disallowed, while 2 disallows additional sends and receives.

Byte order

Because of differences between various computer architectures and network protocols, the byte-swapping routines can be used to simplify manipulation of names and addresses:

htonl converts 32-bit host to network long integer

htons converts 16-bit host to network short integer

ntohl converts 32-bit network to host long integer

ntohs converts 16-bit network to host short integer

For example, to convert an integer before sending it through a socket:

```
i=htonl(i);  
write_data(s, &i, sizeof(i));
```

After reading data, it can be converted back:

```
read_data(s, &i, sizeof(i));  
i=ntohl(i);
```

See the **bstring(S)** manual page in the *SCO TCP/IP Programmer's Reference* for further information on byte ordering routines.

bind, listen, accept, fork, exec**Socket API functions**

This list is a summary of functions or methods provided by the Berkeley sockets 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 `sendto()` and `recvfrom()`, 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. IPv4 only.
- `select()` is used to pend, waiting for one or more of a provided list of sockets to be ready to read, ready to write, or that have errors.
- `poll()` is used to check on the state of a socket in a set of sockets. The set can be tested to see if any socket can be written to, read from or if an error occurred.
- `getsockopt()` is used to retrieve the current value of a particular socket option for the specified socket.
- `setsockopt()` is used to set a particular socket option for the specified socket.

Further details are given below.

socket()

`socket()` creates an endpoint for communication and returns a file descriptor for the socket. `socket()` takes three arguments:

- *domain*, which specifies the protocol family of the created socket. For example:
 - `PF_INET` for network protocol IPv4 or
 - `PF_INET6` for IPv6.
 - `PF_UNIX` for local socket (using a file).
- *type*, one of:
 - `SOCK_STREAM` (reliable stream-oriented service or Stream Sockets)
 - `SOCK_DGRAM` (datagram service or Datagram Sockets)
 - `SOCK_SEQPACKET` (reliable sequenced packet service), or
 - `SOCK_RAW` (raw protocols atop the network layer).
- *protocol* specifying the actual transport protocol to use. The most common are `IPPROTO_TCP`, `IPPROTO_SCTP`, `IPPROTO_UDP`, `IPPROTO_DCCP`. These protocols are specified in `<netinet/in.h>`. The value 0 may be used to select a default protocol from the selected domain and type.

The function returns -1 if an error occurred. Otherwise, it returns an integer representing the newly-assigned descriptor.

Prototype

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

bind()

`bind()` assigns a socket to an address. When a socket is created using `socket()`, it is only given a protocol family, but not assigned an address. This association with an address must be performed

with the `bind()` system call before the socket can accept connections to other hosts. `bind()` takes three arguments:

- `sockfd`, a descriptor representing the socket to perform the bind on.
- `my_addr`, a pointer to a `sockaddr` structure representing the address to bind to.
- `addrlen`, a `socklen_t` field specifying the size of the `sockaddr` structure.

`Bind()` returns 0 on success and -1 if an error occurs.

Prototype

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

listen()

After a socket has been associated with an address, `listen()` prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes, i.e., for socket types (`SOCK_STREAM`, `SOCK_SEQPACKET`). `listen()` requires two arguments:

- `sockfd`, a valid socket descriptor.
- `backlog`, an integer representing the number of pending connections that can be queued up at any one time. The operating system usually places a cap on this value.

Once a connection is accepted, it is dequeued. On success, 0 is returned. If an error occurs, -1 is returned.

Prototype

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

accept()

When an application is listening for stream-oriented connections from other hosts, it is notified of such events (cf. `select()` function) and must initialize the connection using the `accept()` function. The `accept()` function creates a new socket for each connection and removes the connection from the listen queue. It takes the following arguments:

- `sockfd`, the descriptor of the listening socket that has the connection queued.
- `cliaddr`, a pointer to a `sockaddr` structure to receive the client's address information.
- `addrlen`, a pointer to a `socklen_t` location that specifies the size of the client address structure passed to `accept()`. When `accept()` returns, this location indicates how many bytes of the structure were actually used.

The `accept()` function returns the new socket descriptor for the accepted connection, or -1 if an error occurs. All further communication with the remote host now occurs via this new socket.

Datagram sockets do not require processing by `accept()` since the receiver may immediately respond to the request using the listening socket.

Prototype

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

connect()

The `connect()` system call *connects* a socket, identified by its file descriptor, to a remote host specified by that host's address in the argument list.

Certain types of sockets are *connectionless*, most commonly user datagram protocol sockets. For these sockets, `connect` takes on a special meaning: the default target for sending and receiving data gets set to the given address, allowing the use of functions such as `send()` and `recv()` on connectionless sockets.

`connect()` returns an integer representing the error code: 0 represents success, while -1 represents an error.

Prototype

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

gethostbyname() and gethostbyaddr()

The `gethostbyname()` and `gethostbyaddr()` functions are used to resolve host names and addresses in the domain name system or the local host's other resolver mechanisms (e.g., `/etc/hosts` lookup). They return a pointer to an object of type *struct hostent*, which describes an Internet Protocol host. The functions take the following arguments:

- *name* specifies the name of the host. For example: `www.wikipedia.org`
- *addr* specifies a pointer to a *struct in_addr* containing the address of the host.
- *len* specifies the length, in bytes, of *addr*.
- *type* specifies the address family type (e.g., `AF_INET`) of the host address.

The functions return a NULL pointer in case of error, in which case the external integer *h_errno* may be checked to see whether this is a temporary failure or an invalid or unknown host. Otherwise a valid *struct hostent ** is returned.

These functions are not strictly a component of the BSD socket API, but are often used in conjunction with the API functions. Furthermore, these functions are now considered legacy interfaces for querying the domain name system. New functions that are completely protocol-agnostic (supporting IPv6) have been defined. These new function are `getaddrinfo()` and `getnameinfo()`, and are based on a new *addrinfo* data structure.

TCP ports (ephemeral, reserved), Berkeley Sockets: I/O asynchronous

Ephemeral port

An **ephemeral port** is a short-lived transport protocol port for Internet Protocol (IP) communications allocated automatically from a predefined range by the TCP/IP software. It is used by the Transmission Control Protocol (TCP), User Datagram Protocol (UDP), or the Stream Control Transmission Protocol (SCTP) as the port assignment for the client end of a client–server communication to a well known port on a server.

On servers, ephemeral ports may also be used as the port assignment on the server end of a communication. This is done to continue communications with a client that initially connected to one of the server's well-known service listening ports, to make the well-known port available to service requests from other clients. File Transfer Protocol (FTP) and Remote Procedure Call (RPC) applications are two protocols that can behave in this manner. Note that the term 'server' here includes workstations running services that receive connections initiated from other clients (such as Remote Desktop Protocol or RDP).

The allocations are temporary and only valid for the duration of the communication session. After completion of the communication session, the ports become available for reuse.^[note 1] Since the ports are used on a per request basis they are also called dynamic ports.

The Internet Assigned Numbers Authority (IANA) suggests the range 49152 to 65535 for dynamic or private ports.

Many Linux kernels use the port range 32768 to 61000.^[note 2] FreeBSD has used the IANA port range since release 4.6. Previous versions, including the Berkeley Software Distribution (BSD), use ports 1024 through 5000 as ephemeral ports.

Microsoft Windows operating systems through XP use the range 1025 to 5000 as ephemeral ports by default.^[3] Windows Vista, Windows 7, and Server 2008 use the IANA range by default.^[4] Windows Server 2003 uses the range 1025 to 5000 by default, until Microsoft security update MS08-037 from 2008 is installed, after which it uses the IANA range by default.^[5] Few Microsoft articles other than KB Article 956188 reference the new port range used by Windows Server 2003, leading to confusion.

In addition to the default range, all versions of Windows since Windows 2000 also allow the option to use a non-default port range with a maximum of 1025 to 65535.^{[6][7]} Some Microsoft

articles misleadingly list only this non-default range, leading to a popular misconception that 1025 to 65535 is the default or required port range used.

select & poll functions, signal & fcntl functions

Differences between **poll()** and **select()** and to the end some mentions about the more modern event-driven alternatives such as **epoll()**, **kqueue** and more. I recommend a library such as **libev** or **libevent**.

Those libs make it possible to write event-based programs in a portable manner, as the underlying technologies like **epoll** (Linux), **kqueue** (FreeBSD, NetBSD, OpenBSD, Darwin) and **/dev/poll** (Solaris, HPUX), **pollset** (AIX), **Event Completion** (Solaris 10) are different between platforms and aren't standardized. The Windows solution seems to be **I/O Completion Ports**

History

select() was introduced in 4.2BSD Unix, released in August 1983.

poll() was introduced in SVR3 Unix, released 1986. In Linux, the **poll()** system call was introduced in 2.1.23 (January 1997) while the **poll()** library call was introduced in **libc 5.4.28** (May 1997)

Functionality

select() and **poll()** provide basically the same functionality. They only differ in the details:

- **select()** overwrites the **fd_set** variables whose pointers are passed in as arguments 2-4, telling it what to wait for. This makes a typical loop having to either have a backup copy of the variables, or even worse, do the loop to populate the bitmasks every time **select()** is to be called. **poll()** doesn't destroy the input data, so the same input array can be used over and over.
- **poll()** handles many file handles, like more than 1024 by default and without any particular work-arounds. Since **select()** uses bitmasks for file descriptor info with fixed size bitmasks it is much less convenient. On some operating systems like Solaris, you can compile for support with > 1024 file descriptors by changing the **FD_SETSIZE** define.
- **poll** offers somewhat more flavours of events to wait for, and to receive, although for most common networked cases they don't add a lot of value
- Different timeout values. **poll** takes milliseconds, **select** takes a struct **timeval** pointer that offers microsecond resolution. In practise however, there probably isn't any difference that will matter.

Going with an event-based function instead should provide the same functionality as well. They do however often force you to use a different approach since they're often callback-based that get triggered by events, instead of the loop style approach **select** and **poll** encourage.

Speed

poll and **select** are basically the same speed-wise: slow.

- They both handle file descriptors in a linear way. The more descriptors you ask them to check, the slower they get. As soon as you go beyond perhaps a hundred file descriptors or so - of course depending on your CPU and hardware - you will start noticing that the mere waiting for file descriptor activity and the following checking which file descriptor that it was, takes a significant time and becomes a bottle neck.
- The select() API with a "max fds" as first argument of course forces a scan over the bitmasks to find the exact file descriptors to check for, which the poll() API avoids. A small win for poll().
- select() only uses (at maximum) three bits of data per file descriptor, while poll() typically uses 64 bits per file descriptor. In each syscall invoke poll() thus needs to copy a lot more over to kernel space. A small win for select().

Going with an event-based function is the only sane option if you go beyond a small number of file descriptors. The libev guys did a benchmarking of libevent against libev and their results say clearly that libev is the faster one.

Compared to poll and select, any event-based system will give a performance boost already with a few hundred file descriptors and then the benefit just grows the more connections you add.

Portability

select - has existed for a great while and exists almost everywhere. A problem with many file descriptors is that you cannot know if you will overflow the the bitmask as you can't check the file descriptor against FD_SETSIZE in a portable manner.

Many unix versions allow FD_SETSIZE to be re-defined at compile time, but Linux does not

One quirk is that the include header required for the *fd_set* type varies between systems.

Some - but not all - systems modify the timeout struct so that on return from select, the program can know how long time actually passed. If you repeat select() calls, you need to init the timeout struct each loop!

poll - Not existing on older unices nor in Windows before Vista. Broken implementation in Mac OS X at least up to 10.4 and earlier. Up to 10.3 you could check for a poll() that works with all arguments zeroed. The 10.4 bug is more obscure and I don't know of any way to detect it without checking for OS.

Lots of early implementations did poll as a wrapper around select().

poll's set of bits to return is fairly specific in the standards, but vary a lot between implementations