P-1479

TCP/IP HEADER COMPRESSION TOOL

By

R.SENTHILRAJA

Reg. No 71202621041

of

Kumaraguru College of Technology
Coimbatore - 641 006.d

A PROJECT REPORT
Submitted to the

FACULTY OF INFORMATION AND COMMUNICATION ENGINEERING

In partial fulfillment of the requirements

for the award of the degree

of

MASTER OF COMPUTER APPLICATIONS

June, 2005

BONAFIDE CERTIFICATE

Certified that this project report titled

TCP/IP HEADER COMPRESSION TOOL

Is Bonafide work of

Mr. R.SENTHILRAJA (Reg. No: 71202621041)

who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

[illegible]
Project Guide

[illegible]
Head of the Department

We examined the Candidate with University Register No. 71202621041 in the project Viva-Voce examination held on 24.06.05

[illegible]
Internal Examiner

[illegible] 24/6/05
External Examiner

ABSTRACT

The TCP/IP Header Compression Tool allows the user to improve TCP/IP performance by reducing TCP/IP Header size from 40 bytes to 3 bytes.

Normally the TCP/IP Header size is 40 bytes (20 bytes of TCP and 20 bytes of IP), all these header fields serve some useful purpose and it's not possible to simply omit some in the name of efficiency.

However, TCP establishes connections and typically, tens or hundreds of packets are exchanged on each connection. How much of the per-packet information is likely to stay constant over the life of a connection? HALF – So, if the sender and receiver keep track of active connections and the receiver keeps a copy of the header from the last packet it saw from each connection, the sender gets a factor-of-two compression by sending only a small (≤ 8 bit) *connection identifier* together with the 20 bytes that change and letting the receiver fill in the 20 fixed bytes from the saved header.

One can scavenge a few more bytes by noting that any reasonable link-level framing protocol will tell the receiver the length of a received message so *total length* (bytes 2 and 3) is redundant. But then the *header checksum* (bytes 10 and 11), which protects individual hops from processing a corrupted IP header, is essentially the only part of the IP header being sent. It seems rather silly to protect the transmission of information that isn't being transmitted. So, the receiver can check the header checksum when the header is actually sent (i.e., in an uncompressed datagram) but, for compressed datagrams, regenerate it locally at the same time the rest of the IP header is being regenerated.

This leaves 16 bytes of header information to send. All of these bytes are likely to change over the life of the conversation but they do not all change at the same time. For example, during an FTP data transfer only the *packet ID*, *sequence number* and *checksum* change in the sender → receiver direction and only the *packet ID*, *ack*, *checksum* and, possibly, *window*, change in the receiver → sender direction. With a copy of the last packet sent for each connection, the sender can figure out what fields change in the current packet then send a bit mask indicating what changed followed by the changing fields.

If the sender only sends fields that differ, the above scheme gets the average header size down to around ten bytes. However, it's worthwhile looking at how the fields change: The packet ID typically comes from a counter that is incremented by one for each packet sent. I.e., the difference between the current and previous packet IDs should be a small, positive integer, usually < 256 (one byte) and frequently = 1. For packets from the sender side of a data transfer, the sequence number in the current packet will be the sequence number in the previous packet plus the amount of data in the previous packet (assuming the packets are arriving in order). Since IP packets can be at most 64K, the sequence number change must be $< 2^{16}$ (two bytes). So, if the *differences* in the changing fields are sent rather than the fields themselves, another three or four bytes per packet can be saved. That gets us to the five-byte header target.

Recognizing a couple of special cases will get us three byte headers for the two most common cases: *Interactive typing traffic and Bulk data transfer.*

This intellectual exercise suggests it is possible to get three byte headers, it seems reasonable to flesh out the missing details and actually implement something.

ACKNOWLEDGEMENT

I take this opportunity to thank our beloved Principal **Dr.K.K.Padmanabhan B.Sc, (Engg), M.Tech, Ph.D,** who has been a source of inspiration at all the times.

We are indebted to our Head of the Department of Computer Science & Engineering **Prof.Dr.S.Thangasamy Ph.D,** for his constant encouragement and support for the students.

I extend my sincere thanks to our Project Coordinator **Mr.A.Muthukumar M.C.A, M.Phil,** who with his careful supervision has ensured me in attaining perfection of work.

I wish to express my profound gratitude to my Project Guide **Mr.P.Gopalakrishnan M.C.A,** for his uplifting ideas, inspiring guidance and valuable suggestions, which have been very helpful in refining upon the project.

I would like to thank **Mrs.C.Radha M.Com,** Administrator, Senas.Net and my friend and guide **Mr.V.Senthilmurugan B.E,** Software Engineer, Senas.Net, for their unstrained cooperation, help and valuable suggestions throughout the project work.

I would also like to thank all the teaching staff members, Non teaching staff members, friends and well wishers for making this project a realistic.

Finally, last, but most Importantly, I would like to thank my beloved parents whose moral, mental and financial support allowed this project to come into existence.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVATIONS

ARP	- Address Resolution Protocol
BSD	- Berkeley Sockets Distribution
CRC	- Cyclic Redundancy Check
DNS	- Domain Name System
FDDI	- Fiber Distributed Data Interface
FTP	- File Transfer Protocol
HDLC	- High level Data Link Control
ICMP	- Internet Control Message Protocol
IEEE	- Institute of Electrical and Electronics Engineers
IGMP	- Internet Group Message Protocol
IP	- Internet Protocol
LCP	- Link Control Protocol
LLC	- Logical Link Control
MAC	- Media Access Control
MTU	- Maximum Transfer Unit
NCP	- Network Control Protocol
TCP	- Transmission Control Protocol
OSI	- Open Systems Interconnection
PPP	- Point-to-Point Protocol
RARP	- Reverse Address Resolution Protocol
RFC	- Request For Comment
SAP	- Service Access Points
SLIP	- Serial Line IP
SMTP	- Simple Mail Transfer Protocol
TOS	- Type Of Service
TTL	- Time To Live
UDP	- User Datagram Protocol

CHAPTER 1

INTRODUCTION

As increasingly powerful computers find their way into people's homes, there is growing interest in extending Internet connectivity to those computers. Unfortunately, this extension exposes some complex problems in link-level framing, address assignment, routing, authentication and performance. As of this writing there is active work in all these areas.

This project describes a method that has been used to improve TCP/IP performance by reducing the header size. This is essential for low speed Serial links and useful for high-speed lines.

However, this protocol compresses more effectively (the average compressed header is 3 bytes) and is both efficient and simple to implement (the Unix implementation is 250 lines of C and requires, on the average, 90μs (~170 instructions) for a 20MHz MC68020 to compress or decompress a packet).

This compression is specific to TCP/IP datagrams. Compressing UDP/IP datagrams was investigated but found that they were too infrequent to be worth the bother and either there was insufficient datagram-to-datagram coherence for good compression (e.g., name server queries) or the higher level protocol headers overwhelmed the cost of the UDP/IP header (e.g., Sun's RPC/NFS).

Separately compressing the IP and the TCP portions of the datagram was also investigated but rejected since it increased the average compressed header size by 50% and doubled the compression and decompression code size.

1.1 THE PROBLEM

Internet services one might wish to access over a serial IP link from home range from interactive “terminal” type connections (e.g., telnet, rlogin, xterm) to bulk data transfer (e.g., FTP, SMTP). Header compression is motivated by the need for good interactive response. I.e., the *line efficiency* of a protocol is the ratio of the data to header + data in a datagram. If efficient bulk data transfer is the only objective, it is always possible to make the datagram large enough to approach an efficiency of 100%.

Human-factors studies have found that interactive response is perceived as “bad” when low-level feedback (character echo) takes longer than 100 to 200 ms.

Protocol headers interact with this threshold three ways:

(1) If the line is too slow, it may be impossible to fit both the headers and data into a 200 ms window: One typed character results in a 41 byte TCP/IP packet being sent and a 41 byte echo being received. The line speed must be at least 4000 bps to handle these 82 bytes in 200 ms.

(2) Even with a line fast enough to handle packetized typing echo (4800 bps or above), there may be an undesirable interaction between bulk data and interactive traffic: For reasonable line efficiency the bulk data packet size needs to be 10 to 20 times the header size. I.e., the line *maximum transmission unit* or *MTU* should be 500 to 1000 bytes for 40 byte TCP/IP headers.

Even with type-of-service queuing to give priority to interactive traffic, a telnet packet has to wait for any in-progress bulk data packet to finish. Assuming data transfer in only one direction, that wait averages half the MTU or 500 ms for a 1024 byte MTU at 9600 bps.

(3) Any communication medium has a maximum signaling rate, the Shannon limit. Based on an AT&T, the Shannon limit for a typical dialup phone line is around 22,000 bps. Since a full duplex, 9600 bps modem already runs at 80% of the limit, modem manufacturers are starting to offer asymmetric allocation schemes to increase effective bandwidth: Since a line rarely has equivalent amounts of data flowing both directions simultaneously, it is possible to give one end of the line more than 11,000 bps by either time-division multiplexing a half-duplex line (e.g., the Telebit Trailblazer) or offering a low-speed "reverse channel" (e.g., the USR Courier HST). In either case, the modem dynamically tries to guess which end of the conversation needs high bandwidth by assuming one end of the conversation is a human (i.e., demand is limited to <300 bps by typing speed). The factor-of-forty bandwidth multiplication due to protocol headers will fool this allocation heuristic and cause these modems to "thrash".

From the above, it's clear that one design goal of the compression should be to limit the bandwidth demand of typing and ack traffic to at most 300 bps. A typical maximum typing speed is around five characters per second which leaves a budget 30 - 5 = 25 characters for headers or five bytes of header per character typed.

Five byte headers solve problems (1) and (3) directly and, indirectly, problem (2): A packet size of 100–200 bytes will easily amortize the cost of a five byte header and offer a user 95–98% of the line bandwidth for data. These short packets mean little interference between interactive and bulk data traffic.

Another design goal is that the compression protocol be based solely on information guaranteed to be known to both ends of a single serial link.

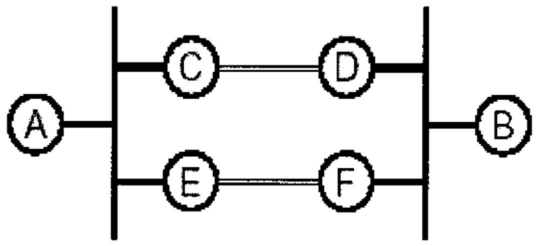

Figure 1.1: A topology that gives incomplete information at gateways

Consider the topology shown in *Figure 1.1*, where communicating hosts A and B are on separate local area nets (the heavy black lines) and the nets are connected by two serial links (the open lines between gateways C–D and E--F).

One compression possibility would be to convert each TCP/IP conversation into a semantically equivalent conversation in a protocol with smaller headers, e.g., to an X.25 call. But, because of routing transients or multipathing, it's entirely possible that some of the A–B traffic will follow the A-C-D-B path and some will follow the A-E-F-B path. Similarly, it's possible that A-->B traffic will flow A-C-D-B and B-->A traffic will flow B-F-E-A. None of the gateways can count on seeing all the packets in a particular TCP conversation and a compression algorithm that works for such a topology cannot be tied to the TCP connection syntax.

A physical link treated as two, independent, simplex links (one each direction) imposes the minimum requirements on topology, routing and pipelining. The ends of each simplex link only have to agree on the most recent packet(s) sent on that link.

Thus, although any compression scheme involves shared state, this state is spatially and temporally local and adheres to Dave Clark's principle of *fate sharing*: The two ends can only disagree on the state if the link connecting them is inoperable, in which case the disagreement doesn't matter.

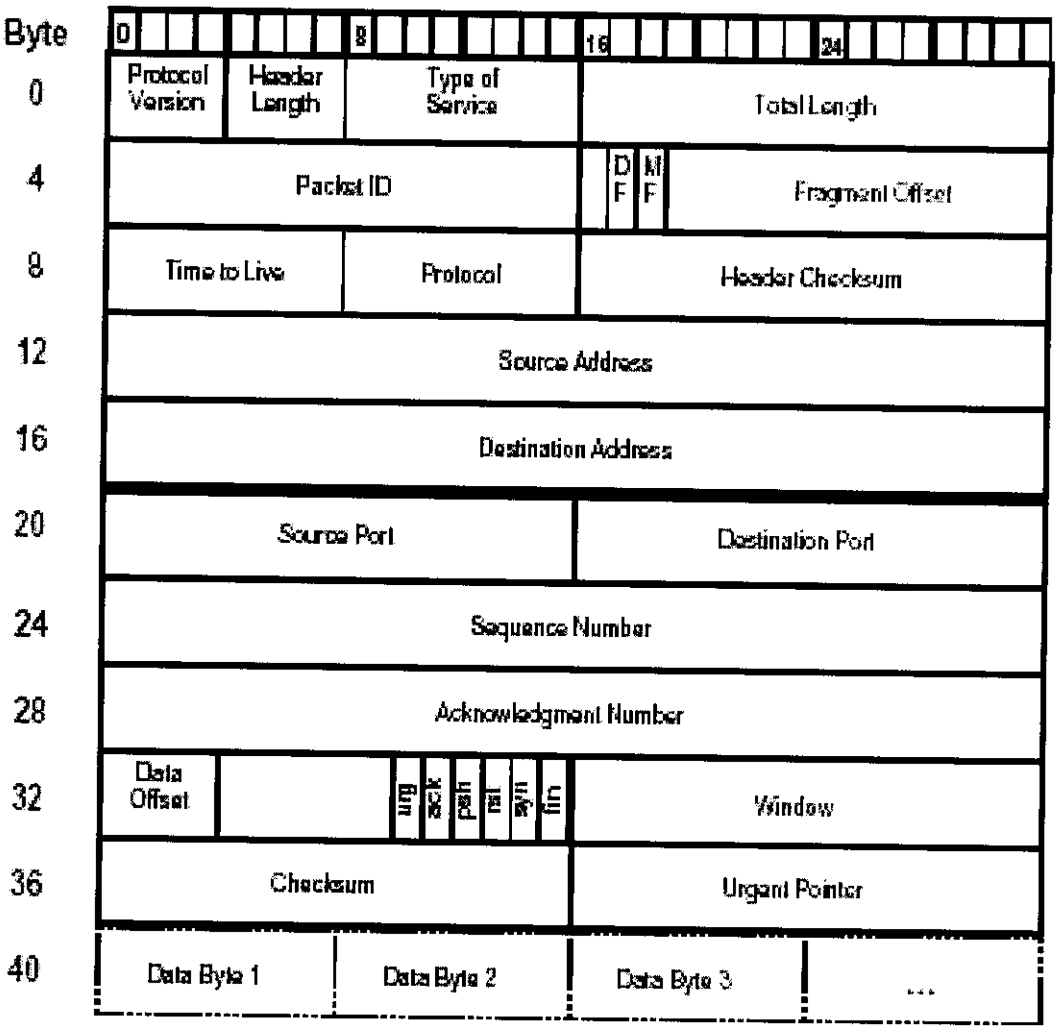

Figure 1.2: The header of a TCP/IP datagram

1.2 WORK DONE

The TCP/IP Header Compression Tool allows the user to improve TCP/IP performance by reducing TCP/IP Header size from 40 bytes to 3 bytes.

Figure 1.2 shows a typical (and minimum length) TCP/IP datagram header. The TCP/IP Header size is 40 bytes: 20 bytes of TCP and 20 bytes of IP. Unfortunately, since the TCP and IP protocols were not designed by a committee, all these header fields serve some useful purpose and it's not possible to simply omit some in the name of efficiency.

However, TCP establishes connections and typically, tens or hundreds of packets are exchanged on each connection. How much of the per-packet information is likely to stay constant over the life of a connection? HALF – the shaded fields in *Figure 1.3*. So, if the sender and receiver keep track of active connections and the receiver keeps a copy of the header from the last packet it saw from each connection, the sender gets a factor-of-two compression by sending only a small (<=8 bit) *connection identifier* together with the 20 bytes that change and letting the receiver fill in the 20 fixed bytes from the saved header.

One can scavenge a few more bytes by noting that any reasonable link-level framing protocol will tell the receiver the length of a received message so *total length* (bytes 2 and 3) is redundant. But then the *header checksum* (bytes 10 and 11), which protects individual hops from processing a corrupted IP header, is essentially the only part of the IP header being sent. It seems rather silly to protect the transmission of information that isn't being transmitted.

So, the receiver can check the header checksum when the header is actually sent (i.e., in an uncompressed datagram) but, for compressed datagrams, regenerate it locally at the same time the rest of the IP header is being regenerated.

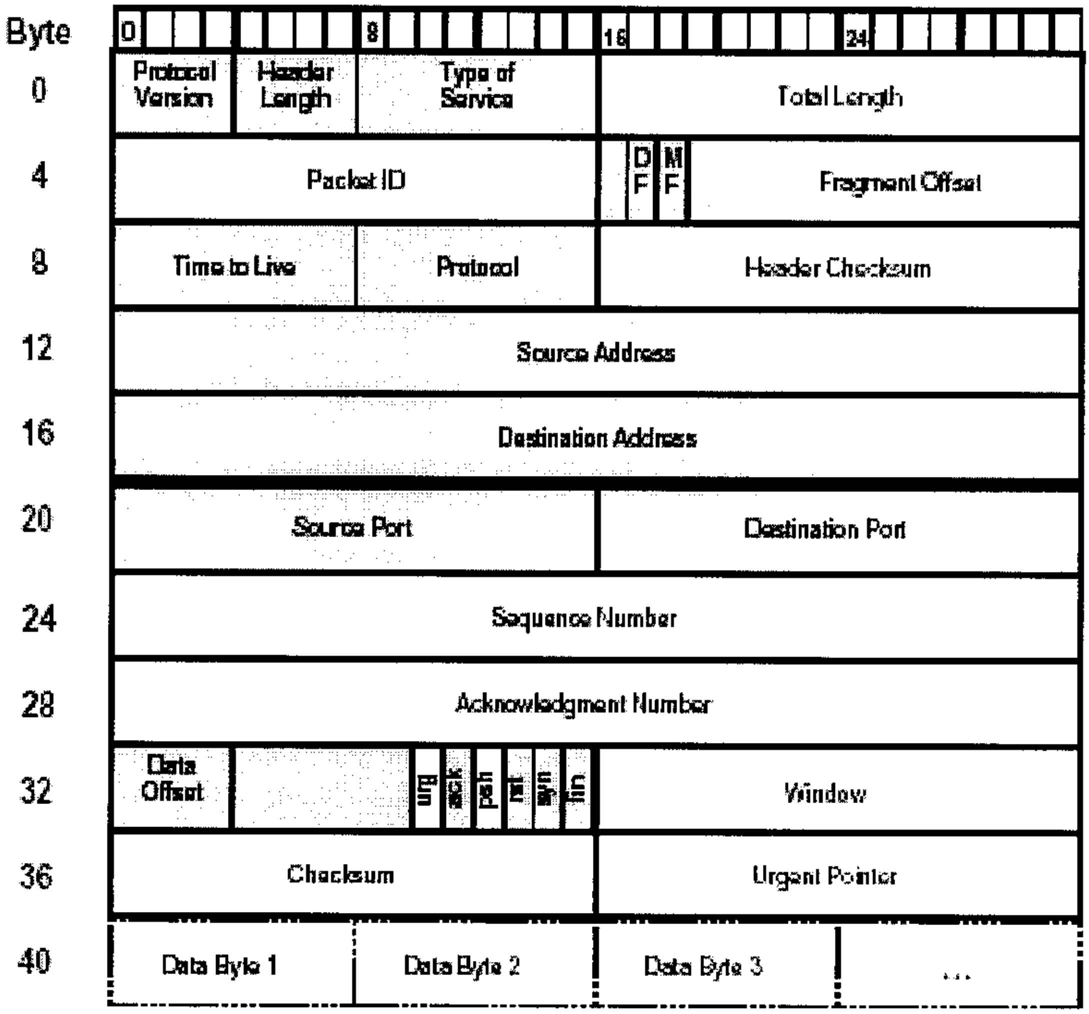

Figure 1.3: Fields those changes during a TCP connection

This leaves 16 bytes of header information to send. All of these bytes are likely to change over the life of the conversation but they do not all change at the same time. For example, during an FTP data transfer only the *packet ID*, *sequence number* and *checksum* change in the sender → receiver direction and only the *packet ID*, *ack*, *checksum* and, possibly, *window*, change in the receiver → sender direction. With a copy of the last packet sent for each connection, the sender can figure out what fields change in the current packet then send a bit mask indicating what changed followed by the changing fields.

If the sender only sends fields that differ, the above scheme gets the average header size down to around ten bytes. However, it's worthwhile looking at how the fields change: The packet ID typically comes from a counter that is incremented by one for each packet sent. I.e., the difference between the current and previous packet IDs should be a small, positive integer, usually < 256 (one byte) and frequently =1. For packets from the sender side of a data transfer, the sequence number in the current packet will be the sequence number in the previous packet plus the amount of data in the previous packet (assuming the packets are arriving in order). Since IP packets can be at most 64K, the sequence number change must be $< 2^{16}$ (two bytes). So, if the *differences* in the changing fields are sent rather than the fields themselves, another three or four bytes per packet can be saved. That gets us to the five-byte header target.

Recognizing a couple of special cases will get us three byte headers for the two most common cases - Interactive typing traffic and Bulk data transfer.

This intellectual exercise suggests it is possible to get three byte headers, it seems reasonable to flesh out the missing details and actually implement something.

CHAPTER 2

PROTOCOL ARCHITECTURE

2.1 THE OSI REFERENCE MODEL

The *Open System Interconnection (OSI) reference model* describes how information from a software application in one computer moves through a network medium to a software application in another computer. The OSI reference model is a conceptual model composed of seven layers, each specifying particular network functions. The model was developed by the International Organization for Standardization (ISO) in 1984, and it is now considered the primary architectural model for intercomputer communications.

The OSI model divides the tasks involved with moving information between networked computers into seven smaller, more manageable task groups. A task or group of tasks is then assigned to each of the seven OSI layers. Each layer is reasonably self-contained so that the tasks assigned to each layer can be implemented independently. This enables the solutions offered by one layer to be updated without adversely affecting the other layers.

The following list details the seven layers of the Open System Interconnection (OSI) reference model. *Figure 2.1* illustrates the OSI Model.

- Layer 7—Application
- Layer 6—Presentation
- Layer 5—Session
- Layer 4—Transport
- Layer 3—Network
- Layer 2—Data link
- Layer 1—Physical

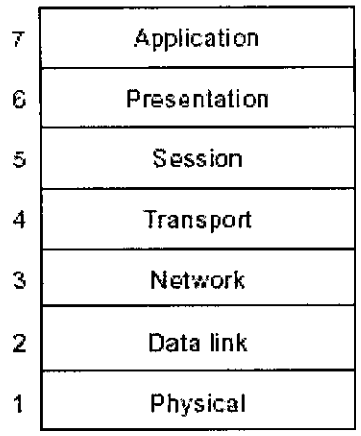

Figure 2.1: The OSI Reference Model

2.1.1 Characteristics of the OSI Layers

The seven layers of the OSI reference model can be divided into two categories: *upper layers* and *lower layers.*

The *upper layers* of the OSI model deal with application issues and generally are implemented only in software. The highest layer, the application layer, is closest to the end user. Both users and application layer processes interact with software applications that contain a communications component. The term upper layer is sometimes used to refer to any layer above another layer in the OSI model.

The *lower layer*s of the OSI model handle data transport issues. The physical layer and the data link layer are implemented in hardware and software. The lowest layer, the physical layer, is closest to the physical network medium (the network cabling, for example) and is responsible for actually placing information on the medium. *Figure 2.2* illustrates the division between the upper and lower OSI layers.

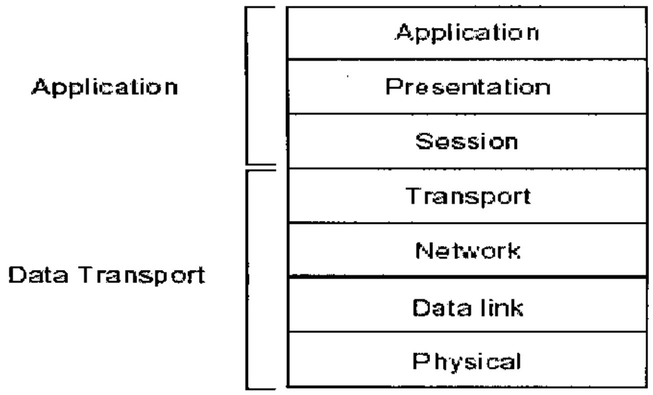

Figure 2.2: Two Sets of Layers Make up the OSI Layers

2.1.2 Interaction between OSI Model Layers

A given layer in the OSI model generally communicates with three other OSI layers: the layer directly above it, the layer directly below it, and its peer layer in other networked computer systems. The data link layer in System A, for example, communicates with the network layer of System A, the physical layer of System A, and the data link layer in System B. *Figure.2.3* illustrates this example.

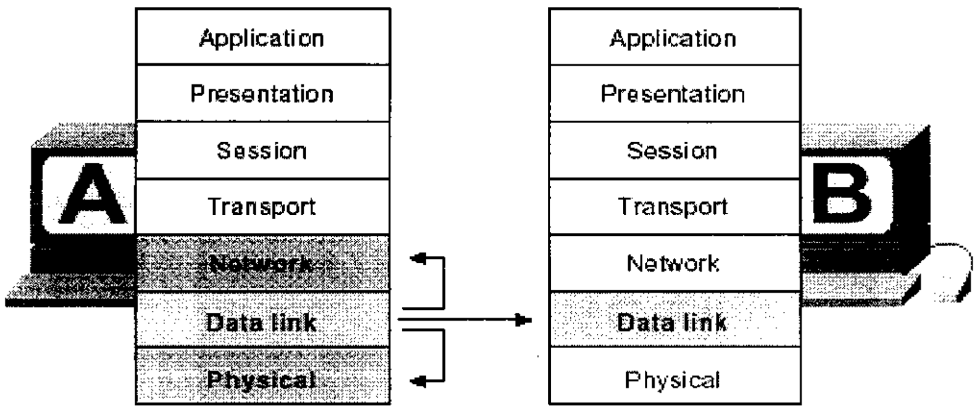

Figure 2.3: OSI Model Layers Communicate with Other Layers

2.1.3 OSI Layer Services

One OSI layer communicates with another layer to make use of the services provided by the second layer. The services provided by adjacent layers help a given OSI layer communicate with its peer layer in other computer systems. Three basic elements are involved in layer services: *the service user, the service provider, and the service access point (SAP).*

In this context, the *service user* is the OSI layer that request services from an adjacent OSI layer. The *service provider* is the OSI layer that provides services to service users. OSI layers can provide services to multiple service users. The SAP is a conceptual location at which one OSI layer can request the
services of another OSI layer. *Figure 2.4* illustrates how these three elements interact at the network and data link layers.

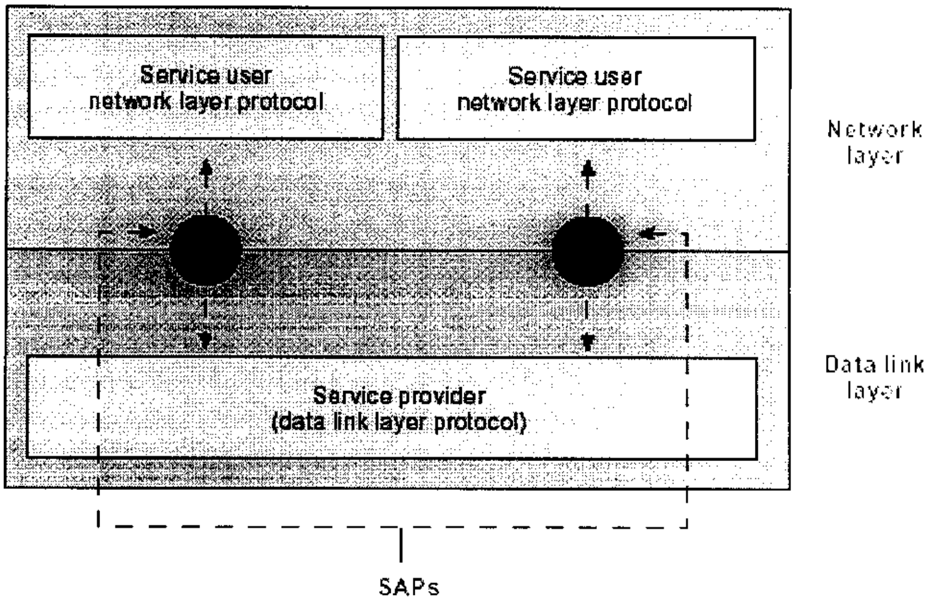

Figure 2.4: Service Users, Providers, and SAPs Interact at the Network and Data Link Layers

2.1.4 OSI Model Layers and Information Exchange

The seven OSI layers use various forms of control information to communicate with their peer layers in other computer systems. This *control information* consists of specific requests and instructions that are exchanged between peer OSI layers. Control information typically takes one of two forms: *headers and trailers. Headers* are prepended to data that has been passed down from upper layers. *Trailers* are appended to data that has been passed down from upper layers.

An OSI layer is not required to attach a header or a trailer to data from upper layers. Headers, trailers, and data are relative concepts, depending on the layer that analyzes the information unit.

At the network layer, for example, an information unit consists of a Layer 3 header and data. At the data link layer, however, all the information passed down by the network layer (the Layer 3 header and the data) is treated as data.
In other words, the data portion of an information unit at a given OSI layer potentially can contain headers, trailers, and data from all the higher layers. This is known as *encapsulation.*

Figure 2.5 shows how the header and data from one layer are encapsulated into the header of the next lowest layer.

2.1.5 Information Exchange Process

The information exchange process occurs between peer OSI layers. Each layer in the source system adds control information to data, and each layer in the destination system analyzes and removes the control information from that data.