California State University, San Bernardino

CSUSB ScholarWorks

Theses Digitization Project John M. Pfau Library

1997

# Design and implemetation of internet mail servers with embedded data compression

Alka Nand

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd-project

Part of the Computer and Systems Architecture Commons

Recommended Citation

Nand, Alka, "Design and implemetation of internet mail servers with embedded data compression" (1997). *Theses Digitization Project*. 1482.
https://scholarworks.lib.csusb.edu/etd-project/1482

This Thesis is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

# DESIGN AND IMPLEMENTATION OF INTERNET MAIL SERVERS WITH EMBEDDED DATA COMPRESSION

---

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Alka Nand

March 1997

DESIGN AND IMPLEMENTATION OF INTERNET MAIL SERVERS WITH

EMBEDDED DATA COMPRESSION

---

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Alka Nand

March 1997

Approved by:

Tong L. Yu, Chair, Computer Science

3/7/97

Date

George M. Georgiou

Kerstin Voigt

# ABSTRACT

The Internet is used to transmit massive amounts of information every second. The vast volume of network traffic may cause congestion resulting in delays. The motivation for this thesis rose from the need for Internet servers that perform data compression within the server. On many networks, electronic mail (e-mail) is the most extensively used application. In this thesis, an Internet mail server with data compression is presented. Different compression mechanisms, such as Huffman coding, arithmetic coding, and dictionary techniques, are evaluated. The LZ77 compression scheme provides good speed and compression ratios. The e-mail system was designed using Object Oriented methodology. The POP3 server retrieves mail for individual users; SMTP clients and servers send and receive mail across the Internet. The LZ77 compression scheme is incorporated within the SMTP clients and servers. The SMTP protocol was extended to allow for the mail client and server to negotiate compression transparently. Experimental results based on the implemented e-mail system show that the system is able to transmit mail data across the Internet at enhanced transmission speeds. Embedding the task of data compression within the mail server achieves the goal of increasing effective bandwidth and reducing network traffic.

## ACKNOWLEDGMENTS

I sincerely thank California State University, San Bernardino, and the Computer Science department for supporting me in finishing my thesis.

I would like to express my special appreciation to Dr. Tong Yu, my advisor, who guided me from the very beginning of this research, and who was always available whenever I needed. I would also like to thank my committee members Dr. Georgiou and Dr. Voigt, and my graduate coordinator Dr. Concepcion for their valuable suggestions and comments.

I also extend my gratitude to WaterNet for sponsoring a PPP Internet access account from an early stage of the thesis.

My special gratitude to my family, particularly my son Pulkit who showed great understanding all along.

TABLE OF CONTENTS

ABSTRACT ........ iii
ACKNOWLEDGMENTS ........ iv
LIST OF TABLES ........ vii
LIST OF ILLUSTRATIONS ........ viii
CHAPTER 1. INTRODUCTION ........ 1
1.1 Motivation ........ 2
1.2 Organization of Chapters ........ 4
CHAPTER 2. INTERNET SERVERS ........ 6
2.1 Network Communications ........ 6
2.1.1 The Network Layers and Protocols ........ 7
2.1.2 The Client-Server Model ........ 15
2.2 E-Mail Servers ........ 18
2.2.1 Simple Mail Transfer Protocol (SMTP) ........ 20
2.2.2 POP3 ........ 24
CHAPTER 3. DATA COMPRESSION ........ 29
3.1 Compression Techniques ........ 30
3.1.1 Huffman Coding ........ 31
3.1.2 Arithmetic Coding ........ 32
3.1.3 Dictionary Techniques ........ 33
CHAPTER 4. E-MAIL SYSTEM WITH DATA COMPRESSION ........ 37
4.1 Preliminary Investigations ........ 38
4.2 Mail Server Architecture ........ 39
4.2.1 The Mailbox Database ........ 40
4.2.2 Outgoing Queue ........ 41
4.2.3 SMTP Server ........ 41
4.2.4 SMTP Client ........ 42
4.2.5 POP3 Server ........ 43
4.3 Data Compression Handling in The Mail Server ........ 44

4.4 SMTP Protocol Extension ........ 45
4.5 Implementation Details ........ 50
CHAPTER 5. PERFORMANCE EVALUATION ........ 59
5.1 Comparing Transmission Speeds ........ 59
5.2 Tasks Accomplished ........ 62
CHAPTER 6. FUTURE ENHANCEMENTS AND CONCLUSION ........ 64
6.1 Enhancements to Designed Server ........ 64
6.1.1 Allowing Multiple Compression Schemes ........ 64
6.1.2 Automatic Selection ........ 65
6.2 Extending The Design to Other Servers ........ 66
6.3 Conclusion ........ 66
APPENDIX A: MAJOR CLASSES ........ 68
ACRONYMS ........ 76
REFERENCES ........ 78

LIST OF TABLES

Table 4-1: File Size and Compression Ratios for Different Kinds of Files .......... 39

Table 5-1: Comparison of Transmission Time .................................................... 61

LIST OF ILLUSTRATIONS


Figure 2.1: Network Layers in ISO/OSI Network Model ........ 8

Figure 2.2: The Layers of the TCP/IP Protocol Suite ........ 11

Figure 2.3: Basic Elements of a Network E-Mail System ........ 18

Figure 2.4: Basic Elements of an Internet E-Mail System ........ 19

Figure 2.5: Sample Mail Transaction ........ 21

Figure 2.6: POP Client/Server Configuration ........ 25

Figure 2.7: Sample POP3 Transaction ........ 27

Figure 4.1: Internet Mail Server ........ 40

Figure 4.2: Outgoing Mail Use Case Diagram ........ 47

Figure 4.3: Incoming Mail Use Case Diagram ........ 49

Figure 4.4: Sample Transaction with SMTP Compression Service Extension ........ 50

Figure 4.5: Main Class Diagram ........ 51

Figure 4.6: MailboxDB Class Diagram ........ 52

Figure 4.7: Mailbox Database Design ........ 53

Figure 4.8: TCP Class Diagram ........ 54

# CHAPTER 1. INTRODUCTION

The Internet is becoming an ever-increasing source of information. Internet servers provide specific services that are beneficial to all network users or at least a group of network users. Typically, a user request involving access to an Internet Server is transmitted by the client application across the network. The client process requests for a connection to a server and once the connection is established, requests the service from the server. The Internet has gained widespread popularity and is used to transmit massive amounts of information every second. As the amount of information that is needed, desired and available, increases, the need for compressing this information efficiently increases as well. As society becomes more advanced and complex, we need to be able to communicate ever more rapidly. The vast volume of network traffic may cause congestion, resulting in delays and other problems. The more bytes sent across the Internet, the more the traffic, the higher the costs and the more the delays. Network traffic can often be reduced by compressing the data before sending it. Data compression allows transmission of data at speeds many times faster than otherwise possible.

The goal of data compression is to develop techniques that can represent the given information in the most efficient way. Data compression is related intimately with data representation. Latest data compression techniques exploit the different

kinds of structures that may be present in different kinds of data like textual data, speech data, image data etc.

## 1.1 MOTIVATION

The motivation for this thesis rose from the need for Internet servers that perform data compression within the servers. Data compression allows speedy transmission of data. Currently, existing servers do not incorporate data compression techniques. It is left to the user or user level application programs to perform any compression of data before transmitting it across large distances on the Internet. This implies that large amounts of data that could potentially be compressed, is being transmitted as such, incurring loss of both time and money.

On most networks, electronic mail (e-mail) is the most extensively used application. As a matter of fact, about one-half of all Internet connections established by Internet users are for transmission and receipt of e-mail messages [1]. In view of the popularity of electronic mail, this thesis will concentrate on Internet Mail Servers. Initially Internet mail was intended specifically for the exchange of text messages. As such the message format specified for mail transfer limited the contents of electronic mail messages to relatively short lines of seven-bit ASCII. With increase in use of electronic mail for transport of non-text messages, such as multimedia messages that might include audio or images, this format and its limitations proved increasingly restrictive for the user community. Users were forced

to convert any non-textual data that they might wish to send into seven-bit bytes representable as printable ASCII characters before invoking a local mail program to send the mail. Some examples of such encoding currently employed in the Internet are pure hexadecimal, uuencode, the 3-in-4 base 64 scheme, the Andrew Toolkit Representation and many others [1]. These problems have since been solved with the help of several mechanisms that combine to overcome most of these limitations. The introduction of Multipurpose Internet Mail Extensions (MIME) made it possible to include, in a standardized way, arbitrary types of data objects [2]. However there is still no provision for mail data to be compressed before transmission. The main thrust of this thesis is, therefore, to design and implement an Internet mail server that automatically and efficiently compresses mail data in manner that is transparent to the user. In order to incorporate the mail compression algorithm within the mail server it became necessary to develop an indigenous mail delivery system. This Internet e-mail system effectively speeds up transmission and helps in reducing network traffic. The compression capabilities of the mail server allow transmission of data many times faster than otherwise possible.

The system has been developed in the Windows environment. It is a 32-bit application that can run on 32-bit Windows operating systems like Windows 95 or Windows NT. It can also be executed on Windows 3.1x 16-bit Windows operating systems using Win32s. Win32s is a subsystem created by Microsoft for the Windows operating systems 3.1x and Windows for Workgroups (WFW) 3.1x 16-bit Windows

operating systems. The Win32s libraries allow Win3.1x users to run Win32 (32-bit) applications (that run on Windows 95 and Windows NT) on their 16-bit operating systems. Sockets have been used for communication on the TCP/IP networks. The Windows Socket Interface is based on the socket paradigm and has been derived from the Berkeley socket interface that was designed for UNIX systems.

## 1.2 ORGANIZATION OF CHAPTERS

Chapter 2 introduces Internet Servers in general and Mail servers in particular. Preliminaries of network communication are discussed with a brief overview of the different ISO/OSI layers and the associated protocols. Since network server architectures are based on the Client-Server model, it seems appropriate to review concepts of the Client-Server methodology. The socket paradigm for TCP communications and the basics of the Windows Socket Interface are described. Lastly, the design of E-Mail systems and the essential components are also explained in this chapter.

Data compression is an integral part of this thesis and Chapter 3 introduces the various different data compression mechanisms. Compression techniques may be *lossless* in which no information is lost, or *lossy* in which higher compression is achieved at the cost of loss of information. For the purpose of compression within the mail server lossless techniques have been studied. Emphasis has been given to some

of the more popular mechanisms like Huffman Coding, Arithmetic Coding and Dictionary techniques.

Chapter 4 describes the whole process of implementing the Internet E-Mail server with embedded compression. To begin with an investigation of the effects of incorporating data compression before transmission was carried out. A brief evaluation of the different compression techniques was done to determine the one most appropriate for use in the E-Mail server. The architecture of the new E-Mail system is described in great detail, elaborating on the mailbox database, the POP3 server and the SMTP client and server modules. This chapter also elucidates how data compression was implemented within the server. The SMTP protocol had to be extended to support compression. The extension to the protocol provides a means whereby an SMTP client and server that support compression may recognize each other. The SMTP protocol extensions and its implications have been discussed at length in this chapter.

In Chapter 5 several performance evaluation results are presented and the speedups achieved are illustrated. Future enhancements like more sophisticated data analysis for more optimal selection compression algorithm, are considered in Chapter 6. Suggestions for extending the design to other Internet servers and the final conclusions, are also included in this final chapter of the thesis.

# CHAPTER 2. INTERNET SERVERS

The Internet is a collection of communication networks that are connected together by gateways. Gateways are devices that connect two subnetworks and allow communication between them even though they may or may not be similar. The Internet is the largest and most widely known internetwork in the world. It connects well over 20,00 computer networks in around 130 countries [1]. The key internetworking concepts, necessary for a discussion of the E-Mail system designed as part of this thesis, are presented in this chapter.

## 2.1 Network Communications

A computer network may be defined as two or more interconnected computers, capable of communicating with each other. The communication network is the facility that provides a data transfer service among computers attached to the network. Conceptually , a network may be divided into 2 fundamental components: *network applications* and a *network communication subsystem* [1]. The network communication system is the delivery system used to transmit network applications data across the network.

For two *entities* in different systems to successfully communicate they must "speak the same language". Entities refers to the user application programs, file transfer software packages, electronic mail facilities or any other agents that are

capable of sending or receiving data. Communication between the two entities must conform to some mutually acceptable set of conventions. The set of conventions and rules that govern the exchange of data is known as *protocol.*

The Reference Model of Open Systems Interconnection, also referred to as the ISO/OSI model, uses layers to organize a network into well-defined, functional modules. All communication functions are partitioned into a vertical set of layers [3]. Each layer is responsible for a related subset of the functions required to communicate with another system. It provides a specific functionality to the next higher layer, shielding it from lower-level implementation details and in turn relies on the adjacent lower layer to perform more primitive functions. The task of communication is thus decomposed into a number of manageable subtasks.

### 2.1.1 The Network Layers and Protocols

Figure 2.1 shows the network layers in the ISO/OSI network model. The ISO/OSI model and the different layers and protocols are briefly reviewed.

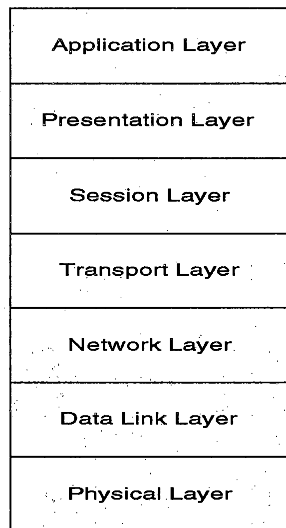


Figure 2.1: Network Layers in ISO/OSI Network Model

The physical layer actually transmits the unstructured bit stream through the network's communication channels. This layer includes the hardware needed to accomplish the transmission and deals with the mechanical, electrical, functional and procedural characteristics to access the physical medium.

The data link layer primarily prevents data corruption within the physical layer. It provides for reliable transfer of information by sending blocks of data frames with the necessary synchronization, error control and flow control.

The Network layer may be termed as the delivery system within the network that is responsible for establishing, maintaining and terminating connections. This layer provides the upper layers independence from the data transmission and switching technologies used to connect systems.

The Transport layer provides reliable, transparent transfer of data between communication end points. While the network layer delivers data packets across the network, the transport layer transports data within the host computer making sure the data reaches the correct application.

The Session layer is the users interface to the network and establishes, manages, and terminates connections (sessions) between cooperating applications. As such it provides the control structure for communication between applications and handles details such as account names, password, and user authorization.

The Presentation layer provides independence to the application processes from differences in data representation (syntax). It handles all details related to the network's interface to printers, video displays, and file formats.

The Application layer provides access to the OSI environment for users and contains details about network-wide applications like E-Mail and distributed databases.

For communication between two systems, the same set of layered functions must exist on both the systems. Communication is achieved when the corresponding (peer) entities in the same layer in two different systems talk to each other via a protocol. Each of the ISO/OSI layers is associated with a corresponding protocol that it uses to communicate. Within the ISO/OSI model, the layer name is used to identify the layer's protocol. For example, the transport layer protocols are referred as the transport protocols. Conceptually when two host computers talk to each other, the

corresponding layers within each host also carry on a conversation. Communication between peer processes is *virtual communication*, with no direct interchange except at the physical layer. In other words, above the physical layer, each protocol sends data down to the next lower layer to enable the data to get across to the target machine.

The OSI approach is specially useful since it allows communication between heterogeneous computers as long as they implement the same set of communication functions that are organized into the same set of layers, and peer layers share a common protocol.

The TCP/IP protocol suite is based on the ISO/OSI model. Both deal with communications between heterogeneous computers and both are based on the concept of protocol. However an historical difference between the two is the importance laid on internetworking by TCP/IP. Internetworking refers to the communication between two systems that are not attached to the same network. This involves passage of data across at least two networks. Furthermore, these networks may be quite different from each other. Another difference between the ISO/OSI model and the TCP/IP model is that the latter places equal importance on connectionless and connection-oriented services whereas the former is based solely on connection-oriented service. A connection-oriented service must establish connection with another system before any communication can occur, as opposed to

a connectionless service in which data is transferred from one entity to another without prior construction of a connection.

The TCP/IP protocol architecture is organized into layers. TCP is an acronym for Transport Control Protocol while IP stands for Internet Protocol. The Internet consists of thousands of networks that use the TCP/IP protocol suite. The TCP/IP protocol suite is a collection of complementary and cooperative protocols that work together to communicate information across the Internet. TCP/IP is generally considered to be a 4-layer system as indicated in Figure 2.2 [18].

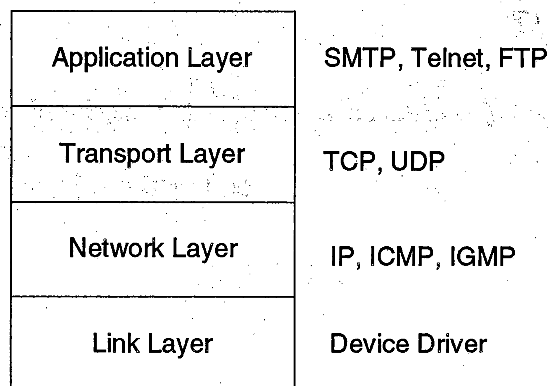


Figure 2.2: The Layers of the TCP/IP Protocol Suite

The link layer or network access layer corresponds to the data link layer of the ISO/OSI model and normally includes the device driver in the operating system and the corresponding network interface in the computer. Together they handle all hardware details of physically interfacing with the transmission media in a manner transparent to all other layers above it. The network layer (also called the Internet layer) handles movement of data allowing it to traverse multiple networks between

hosts. It is responsible for data routing. IP (Internet Protocol), ICMP (Internet Control Message Protocol) and IGMP (Internet Group Management Protocol) are the network layer protocols that are usually implemented within hosts and gateways. The transport layer delivers data between two processes on different hosts. Two vastly different transport protocols provide this functionality: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP is a *reliable* protocol that guarantees delivery of data through the use of checksums, acknowledgment messages etc. Conversely, UDP is an *unreliable* protocol that provides a much simpler service, thus cutting costs in terms of complexity and network bandwidth. Any desired reliability is the responsibility of the application layer. The Application layer contains various protocols for sharing of resources (e.g. between computers) and remote access ( e.g. terminal-to-computer). The most common TCP/IP applications that are implemented on almost every application are SMTP, the Simple Mail Transfer Protocol, FTP, the File Transfer Protocol, and TELNET for remote login. A critical difference between the application layer and the lower layers is that it concerns itself with the details of the application without being bothered about the movement of data across the network. In contrast the lower layers know nothing about the application but handle all the communication details.

As is clear from the above explanation, the TCP/IP protocol suite refers to a collection of protocols that include the Transport Control Protocol and the Internet

Protocol, but is not limited to these two alone. The commonly used TCP/IP protocols and their a brief description of their functionality is given below.

- TCP

The Transmission Control Protocol is a transport layer protocol that provides reliable movement of data between applications.

- UDP

The User Datagram Protocol is another transport layer protocol that sends and receives *datagrams* for applications. Datagrams are units of information that travel from sender to receiver. UDP is unreliable and does not guarantee that a datagram will ever get to its final destination.

- IP

The Internet Protocol is the main protocol of the network layer. It is used by both TCP and UDP for the movement of data between host computers.

- ICMP

The Internet Control Message Protocol is used by the IP layer to exchange error messages and other critical information with the IP layer in another host.

- IGMP

The Internet Group Management Protocol is another network layer protocol that is used with multicasting: sending a UDP datagram to multiple hosts.

In order to transfer data from one computer to another computer on the network, there must be some way to uniquely identify the destination computer. For this, each computer must be associated with a unique identifier or address. Computers on the Internet contain one or more network interface cards, which connect the computers to the Internet. Each network interface card that is attached to the Internet must possess a unique Internet address. An Internet address is known as an IP address. However, a single host computer on the Internet may have several network interface cards, in which case it would have several valid IP addresses. An IP address is 32 bits or 4 bytes wide and is represented in dotted decimal notation. For example, 134.24.32.66 represents an IP address. In addition to IP addresses, TCP/IP associates a *port* with a protocol. The transport layer routes packets to and from application programs and hence requires a way to identify each application. Each network application has a unique port number that is assigned to it every time it creates a session. From the perspective of the Internet, a port is the address of the application or process. Transport layer protocols store source and destination port numbers. As discussed previously, the Internet includes application protocols for the more commonly used applications like FTP, TELNET, and E-Mail etc. These applications use well-known port assignments that are commonly used for that

particular application. For example, the well-known port assignment for the Simple Mail Transfer Protocol or SMTP is 25 and that for TELNET is 23.

### 2.1.2 The Client-Server Model

The ISO/OSI and the TCP model allows network designers to partition design issues. The applications layer within these models resolves design issues related to specific applications. Most software for network applications is based on another model--the Client-Server Model. Network communication requires a network connection between two entities that talk to each other. A network connection consists of both ends of the communication process, as well as the path between them. The Client-server model divides the network application into two sides: the client side and the server side. Like the ISO/OSI reference model and the TCP/IP model the Client-Server model separates network software design issues into specific, well defined modules namely the client issues and the server issues. The mail server designed as part of this thesis, is based on the client-server model.

In a typical client-server scenario the server application performs all its initializations and then goes to sleep, spending most of its time waiting for a request from a client application. Server applications provide certain specific services that may be beneficial to all network users or at least a group of network users. For example, a company's e-mail server would provide e-mail services that may be accessed from any computer within the company's network. Every time a network user requests to send a mail message to another user, an e-mail client application like

Eudora would transmit a request across the network for a connection to an e-mail server application. The e-mail client would then request the server to send the mail. The e-mail server receives and processes the request and performs all the necessary tasks to ensure proper delivery of the mail message. Likewise, whenever a user requests a file transfer from one computer to another, a client application like FTP sets up a TCP connection to the FTP server application residing on the target machine. The user request is transmitted to the FTP server which then receives the file and does all associated processing required to achieve the transfer.

The socket paradigm facilitates creation of sophisticated server operations and the development of robust client programs. The socket interface is an API for TCP/IP networks. The socket interface allows creation of communication endpoints called *sockets* and transferring of data between them. A socket represents one end of a communication link and has access to all the information associated with the link. However, before a socket can be accessed across the network, it must be bound to an address. Binding makes the socket accessible to other sockets on the network by establishing its address.

In a typical client-server application, the client process requests a connection and the server process accepts it. The server process creates a socket and then configures it using the local IP address and protocol port to associate a local address with the socket. The socket is then bound to the host's IP address and the application's protocol port. The server then listens for the transport layer to deliver

client requests at the specified protocol port. The client process creates a socket but does not need to bind it to its own local IP address. In most cases, on a TCP/IP network, the socket implementation selects the protocol port for the client program and notifies the client when data arrives at the port. However, the IP address of the remote host and the protocol port of the remote server application needs to be specified to configure the created socket for communication with the remote host. Whenever the client process requests a connection to the server process, the server accepts the client's request and establishes a connection. Thenceforth, a direct full-duplex connection exists between the client and the server processes. The two processes can send and receive data through their respective connected sockets for the duration of the connection.

The Windows Socket Interface is an API for TCP/IP networks in the Windows environment and is called Winsock. Internet applications can be written using the library functions provided by the Winsock, WINSOCK.DLL. It has been derived from the Berkeley-socket interface for UNIX systems. Winsock takes advantage of the Windows message-driven environment and is implemented as a dynamic link library (DLL) as opposed to the Berkeley socket-interface that is built into the UNIX operating system. The library of support functions exist as an executable module that the Windows operating system can load at execution time.

## 2.2 E-MAIL SERVERS

Electronic Mail (e-mail) is most definitely one of the most popular Internet applications. Figure 2.3 shows the basic components that a network e-mail system comprises of.

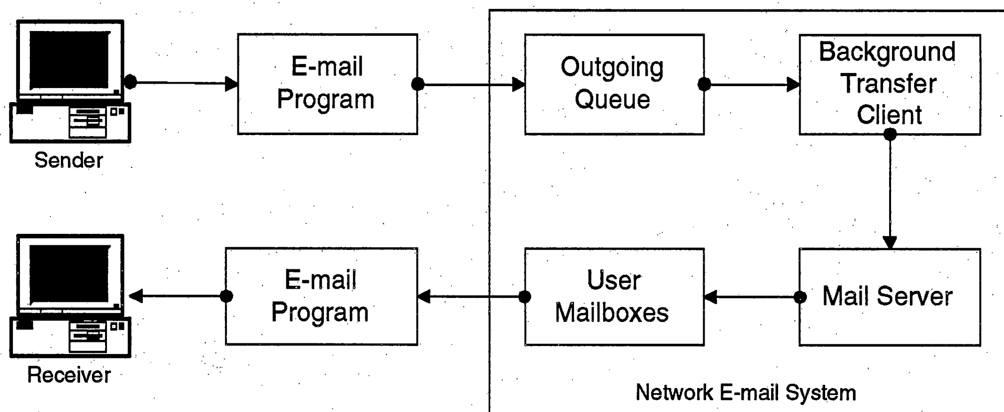


Figure 2.3: Basic Elements of a Network E-Mail System

A user-interface on the sender and receiver machines provides e-mail access to network users. The network e-mail system consists of the outgoing queue that maintains a queue of all messages to be transmitted, a client process and a server process and a collection of individual mail boxes for each users incoming mail. The user-interface or the e-mail program may or may not be an integral part of the network e-mail system, i.e. the user-interface may very well be a separate client program that uses a client-server model to interact with the e-mail system. The

mailbox may be a user address of a single user, consisting of the username and hostname (e.g. jane@orion.csusb.edu) or it may be a database that maintains e-mail data. This database physically stores the incoming messages for individual users.

Figure 2.4 illustrates an overview of e-mail exchange using TCP/IP. The actual components that the Internet e-mail system uses are shown in Figure 2.4.

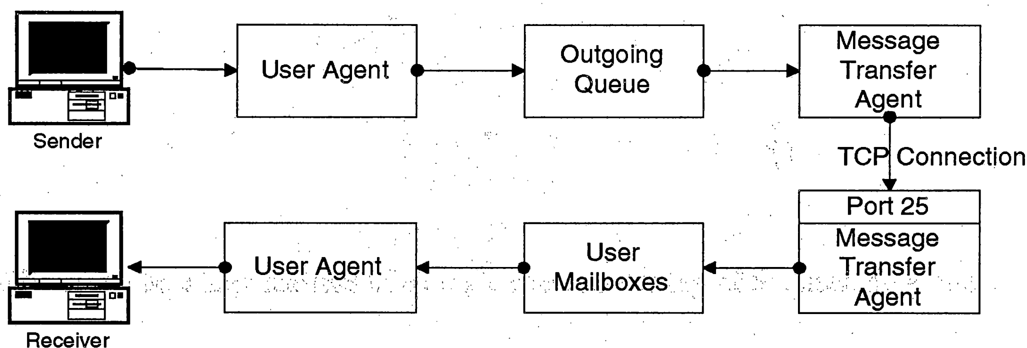


Figure 2.4: Basic Elements of an Internet E-Mail System

The *user agent* is the e-mail program of Figure 2.3 that users deal with, e.g. Elm and Pine on UNIX systems and Eudora on Windows-based systems. Likewise the *message transfer agent* (MTA) replaces the client and server processes of Figure 2.3 and performs the exchange of mail using TCP. While the task of the user agent is to provide the Internet user with a friendly front-end to the Internet's e-mail system, the message transfer agent is mainly concerned with e-mail related services, such as sending or receiving mail for a host computer. The MTA program shields the host

from a wide variety of user agents or other MTAs. To the host computer, the message transfer agent represents the e-mail system. It plays a crucial part in all e-mail transmissions and their role in the Internet's e-mail system cannot be undermined. Once the user agent sends the e-mail message to the outgoing queue, it is the responsibility of the message transfer agent to retrieve the message and transmit it to another MTA. This process of passing the message from one MTA to another continues until the message finally reaches the destination host. Message transfer agents are client and server programs that establish TCP connections to communicate with other MTAs typically using the Simple Mail Transfer Protocol (SMTP).

### 2.2.1 Simple Mail Transfer Protocol (SMTP)

Simple Mail Transfer Protocol (SMTP) is the backbone of the Internet E-Mail system and provides for two way communication between the client (local) and server (remote) MTAs. The objective of SMTP is to transfer mail reliably and efficiently. RFC 821 [4] specifies the SMTP protocol. RFC 822 [5] specifies the format of the electronic mail message that is transmitted using SMTP between two MTAs. Communication between two MTAs uses Network Virtual Terminal (NVT) ASCII. NVT uses standard, 7-bit ASCII encoding for all data, including letters, digits, and punctuation marks and hides computer differences related to line-feeds, form-feeds, carriage-returns, end-of-line markers etc. RFC 854 [6] describes the NVT format in detail. SMTP commands are sent by the client to the server. The

server in turn replies back with numeric reply codes and optional human-readable strings.

In response to user mail request, the sender-SMTP (client) establishes a two-way transmission channel to a receiver-SMTP (server) on TCP port 25. The client then awaits a greeting message (reply code 220) from the server SMTP. A typical mail transaction is shown in Figure 2.5.

```
R: 220 BBN-UNIX.ARPA Simple Mail Transfer Service Ready
S: HELO USC-ISIF.ARPA
R: 250 BBN-UNIX.ARPA

S: MAIL FROM:<jane@USC-ISIF.ARPA>
R: 250 OK

S: RCPT TO:<Jones@BBN-UNIX.ARPA>
R: 250 OK

S: RCPT TO:<Green@BBN-UNIX.ARPA>
R: 550 No such user here

S: RCPT TO:<Brown@BBN-UNIX.ARPA>
R: 250 OK

S: DATA
R: 354 Start mail input; end with <CRLF>.<CRLF>
S: mail data sent here .....
S: ...etc. etc. etc.
S: .
R: 250 OK

S: QUIT
R: 221 BBN-UNIX.ARPA Service closing transmission channel
```

Figure 2.5: Sample Mail Transaction

As soon as the SMTP server receives a request for a connection it forks a child process to deal with this new connection. The child process now acts as the receiver (server) SMTP. The server responds with a 220 reply code and the fully qualified domain name of the server's host, such as silicon.csci.csusb.edu. Once the

transmission channel is established, the sender-SMTP sends the HELO command to identify itself to the receiver (server). The argument to the HELO command must be the fully qualified domain name of the client host e.g. www.sanbernardino.net. Next the sender-SMTP sends the MAIL command indicating the sender of the mail. If the receiver-SMTP is ready to accept mail it replies back with an OK (250 reply code) reply. The sender-SMTP then sends the RCPT command identifying the recipient of the mail. If the SMTP server can accept mail for that recipient it responds with an OK reply; otherwise it rejects that recipient but not the whole mail transaction. If there are multiple recipients to the mail message, the client may send multiple RCPT commands. Once all the recipients have been negotiated, the client SMTP sends the DATA command, followed by the mail data, terminating with a special <CRLF.CRLF> sequence. If the SMTP server is able to successfully process the mail data it responds with an OK reply. The communication is purposely achieved in a lock-step, one-at-a-time manner. SMTP specifies the MAIL-RCPT-DATA sequence of commands as a mail transaction or a mail procedure. Thus, there are three steps to SMTP mail transactions. The transaction begins with the sender sending the MAIL command which provides sender identification. A series of one or more RCPT commands follows, providing information about the recipient. The DATA command delivers the mail data. Finally the end of mail indicator (the <CRLF.CRLF> sequence confirms the transaction and marks the end of the procedure. There may be multiple mail procedures between a client and a server SMTP in the duration of a connection.

The argument to the MAIL command is a reverse path specifying the mail originator. This informs the server MTA how to send error messages back to the e-mail sender. The reverse path includes the mailbox address of the sender, like anand@csci.csusb.edu. Similarly, the argument to the RCPT command is a forward-path, which specifies the receiver of the mail. The forward-path is a source route and includes the mailbox address of the recipient. In case the mail recipient is not acceptable the SMTP server responds with a 550 reply code. An SMTP 550 reply code implies that the SMTP server could not fulfill the client's request since the mailbox was not available. While the server is obligated to notify the client of the nonexistence of a recipient, it is not incumbent upon the client to act on this information in any particular way. SMTP commands and replies have a rigid syntax. They are not case sensitive, however this is not true for user names. The case of user names must be preserved. Replies must have a numeric code. Commands are character strings terminated by <CRLF>. The command codes are alphabetic characters terminated by <SP> if followed by parameters and <CRLF> otherwise.

In addition to HELO, MAIL, RCPT, and DATA there are three more commands that are required in the minimum implementation of SMTP. These are RSET, NOOP and QUIT. RSET specifies that the current mail transaction is to be aborted. Whenever the SMTP client sends a RSET command the SMTP server must discard any stored sender, recipients, and mail data and send an OK reply back to the client. The NOOP command has no effect on any parameters or any previously sent

commands. It specifies no action except that the receiver send an OK reply back to the sender. Lastly the QUIT command dictates that the receiver must send an OK reply and then end the transmission channel. The server SMTP must not close the transmission channel until it receives a QUIT command and replies back to it. Likewise, the SMTP client must not close the transmission channel till it sends the QUIT command and receives a reply back from the server. In any case, if a connection is closed prematurely the SMTP server must behave as if it had just received a RSET command and cancel any pending transaction. All completed transactions still hold good. The client SMTP behaves as if the transaction in progress received an error in reply.

There are additional, optional commands like VRFY, EXPN, HELP, TURN, SEND, SOML, and SAML that are briefly explained for the sake of completeness. The VRFY command requests the server to validate the address of a recipient. EXPN expands a mailing list. The HELP command allows the client SMTP to get useful information from the server SMTP. TURN allows the client and server to switch roles. The SEND, SOML and SAML commands allow combinations of the mail being delivered directly to the users terminal.

### 2.2.2 POP3

Post Office Protocol (POP) is used to retrieve e-mail from an Internet mailbox. POP looks and works very much like SMTP. Figure 2.6 shows a typical POP setup.

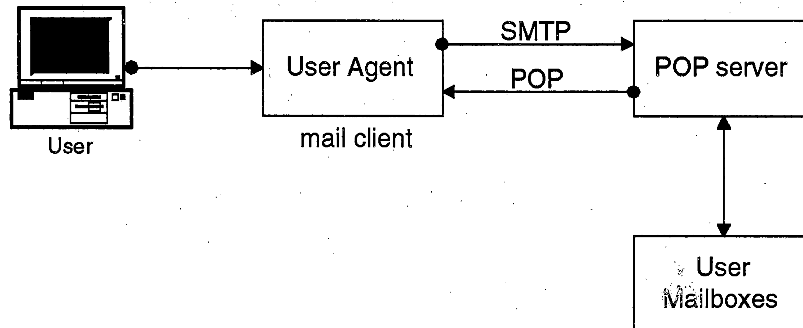


Figure 2.6: POP Client/Server Configuration

The POP server acts as an interface to the mailboxes for the user agent. There are two versions of POP currently in use: POP2 and POP3. POP3 is specifically related to retrieving mail from PC-based remote systems. Most commonly, e-mail systems deliver mail to mailboxes located on e-mail server systems. The practice of delivering mail to individual destination workstations, is becoming less and less popular. It may not be practically possible to permit a SMTP server and associated mail delivery system to be continuously operational on a workstation. Similarly, Internet connectivity is expensive - keeping a personal computer interconnected to the Internet for long lengths of time may not be easily feasible. To solve this problem, many e-mail systems support a node that has a mail server running and offers a mailbox service to the less endowed nodes. Post Office Protocol -version 3

(POP3) is designed to allow a workstation to retrieve mail by accessing a mailbox on a server that is holding the mail for it [7].

The basic operation is very similar to SMTP and consists of a server host starting the POP3 service by listening on TCP port 110. A client wishing to retrieve mail, establishes a TCP connection with the POP3 server. Once the connection is established, the POP3 server responds with a greeting message. The client and server can then exchange commands that consist of a keyword, possibly followed by one or more arguments. Like SMTP all commands and responses are terminated with a <CRLF> and keywords and arguments consist of printable ASCII characters, separated by a single SPACE (<SP>) character. Responses consist of a status indicator and a keyword that may be followed by additional information. Currently, the two status indicators that POP3 recognizes are positive ("+OK") and negative ("-ERR"). Multiline response must be terminated with ("CRLF.CRLF") termination sequence.

A typical POP3 scenario is described in Figure 2.8. POP3 sessions progress through three states or stages. The first is the *authorization* state in which the client identifies itself to the server using the USER <username> and PASS <password> command combination. The server then determines whether to allow the client access to the specified mailbox. After successful authorization, the server acquires an exclusive-access to the mailbox and the session enters the *transaction* state. In the transaction state, the client may issue the STAT, LIST, RETR, DELE, NOOP and

RSET commands. STAT returns the number of messages plus the total size of the messages in the mailbox, back to the client.

```
S: <wait for connection on TCP port 110>
C: <open connection>
S:    +OK POP3 server ready <www.sanbernardino.net>
C:    USER mrose
S:    +OK password required for mrose
C:    PASS abcdef
S:    +OK mrose's maildrop has 2 messages (320 octets)

C:    STAT
S:    +OK 2 320
C:    LIST
S:    +OK 2 messages (320 octets)
S:    1 120
S:    2 200
S:    .
C:    RETR 1
S:    +OK 120 octets
S:    <the POP3 server sends message 1>
S:    .
C:    DELE 1
S:    +OK message 1 deleted
C:    RETR 2
S:    +OK 200 octets
S:    <the POP3 server sends message 2>
S:    .
C:    DELE 2
S:    +OK message 2 deleted

C:    QUIT
S:    +OK POP3 server signing off (maildrop empty)
C:  <close connection>
S:  <wait for next connection>
```

Figure 2.7: Sample POP3 Transaction

The LIST command has an optional message number argument. If specified the server issues a positive response with a line containing information for that message. If no message number was specified the server sends back a multiline response with each line containing the message number of the message and the exact

size of the message in octets (8 bit unit). RETR requires a message number as argument and retrieves the message from the mailbox. DELE also requires a message number argument and marks a message for deletion. The message is not physically removed from the mailbox till POP3 enters the next stage. RSET unmarks all messages marked for deletion. NOOP requires no action on part of the server except to respond with a positive response. Finally the POP3 session enters the *update* state whenever the client issues a QUIT command from the transaction state. However, if the QUIT command is issued from the authorization state , the POP3 session terminates without entering the update state. The POP3 server removes all messages marked as deleted and releases any exclusive-lock on the mailbox. The TCP connection is then closed. If the session terminates for some reason other than the client issuing a QUIT command, the POP3 server does not enter the update state and no messages are removed from the mailbox. These are the commands required in a minimal implementation of POP3. The optional commands that may be implemented are TOP, UIDL, and APOP. TOP requires a message number followed by a number of lines argument. POP3 sends the headers of the message and then the number of lines indicated, from the message's body. UIDL returns the "unique-id" for each message. This unique-id of a message is an arbitrary server determined string that uniquely identifies a mail message within a mailbox. APOP is a alternate method of authentication.

# CHAPTER 3. DATA COMPRESSION

Compression may be defined as "the art or science of representing information in a compact form" [8]. The compact representation may be created by identifying and using regularities that exist in data. Data compression involves the conversion of data with the purpose of reducing its size. A compression technique or algorithm actually consists of two algorithms. One is the compression algorithm that takes the data and generates a representation that is smaller in size, and the other is the reconstruction algorithm that operates on the compressed representation and generates the reconstruction. There are two kinds of compression mechanisms: *lossless* compression techniques in which the reconstruction is identical to the original, and *lossy* compression techniques in which the reconstruction is not identical to the original but that achieve higher compression. In other words, no information is lost in lossless techniques as opposed to lossy techniques which provide much better compression at the cost of loss of information. Lossless compression is more commonly used for discrete data like text, computer-generated data and some kinds of image and video information. Lossless compression preserves data integrity and does not allow any difference to appear as a result of the compression process. In this thesis we shall only be considering lossless mechanisms of data compression. Compression algorithms can be evaluated in many different ways. One very logical way of measuring the performance of an algorithm is to

calculate the ratio of the number of bits required to represent data before compression to the number of bits required to represent the compressed data. This is called the *compression ratio*. Performance could also be evaluated by measuring the time taken to compress data.

## 3.1 COMPRESSION TECHNIQUES

The different data compression algorithms that can be used for compressing mail within the Internet Mail server need to be reviewed. Compression techniques can be classified into statistical methods and dictionary methods. The statistical methods can be divided into two parts, namely modelling and coding; the model reads in the characters and generates statistics (probabilities of character occurrence) to the coding-part to code the characters. The modeling methods try to extract information about any redundancy that exists in the data and describe the redundancy in the form of a model. These models may then be used to obtain compression. The common modeling methods are Physical Models, Probability models and Markov Models. Knowledge about the physics of data generation is used to construct the Physical model. A good example of this is speech-related applications, in which the information about the physics of speech production may be used to create a mathematical model. Statistical models assign a probability of occurrence to each letter in the alphabet. Markov models are the most popular and provide models for representing the dependence of elements of the data sequence on each other.

Arithmetic coding and Huffman coding are coding techniques that employ a modeling method to actually code the characters. While both Huffman and arithmetic coding exploit the statistical structure present in the data to obtain compression, the dictionary-based coding techniques make use of the existence of repetitive patterns by building a dictionary of such patterns.

### 3.1.1 Huffman Coding

Huffman coding is a very popular coding algorithm. Coding refers to the process of assigning binary sequences to symbols or elements of an alphabet. Huffman compression is a statistical data compression technique which gives a reduction in the average code length used to represent the symbols of an alphabet. The set consisting of the binary sequences is called a *code* and the elements of the set are termed as *codewords*. The collection of symbols is called an *alphabet*. Symbols are called *letters*. The ASCII code for the letter 'a' is 1000011. A *uniquely decodable* code can be decoded in one, and only one, way. If none of the codewords in a particular code is a prefix of any other codeword the code is called a *prefix code*. A prefix code will always be uniquely decodable. The Huffman procedure is based on two important observations. First, in an optimum code more frequently occurring symbols will have shorter codewords than less frequently occurring symbols, and second, in an optimum code the two least frequently occurring symbols will have the same length. In addition, the Huffman procedure adds one simple requirement to these observations. The Huffman procedure requires that the two least frequently

occurring symbols have codewords that differ only in the last bit. Huffman coding is not practical in cases where the size of the alphabet is very large. Huffman coding is very suitable for text compression. It is also used for image compression. Huffman compression also leads to some reduction in the capacity of audio data. One of the main advantages of Huffman coding is its simplicity. However, it has its limitations in terms of the compression ratios that it can achieve and some of the other compression techniques score better results.

### 3.1.2 Arithmetic Coding

Another method of generating variable length codes for compression purposes is called *arithmetic coding*. Arithmetic coding is particularly useful when dealing with sources with small alphabets, such as binary sources and alphabets in which the probability of occurrence of the elements ranges widely. It is more efficient to generate codewords for groups or sequences of symbols rather than generating a separate codeword for every symbol in a sequence. The Huffman procedure is not very practical for long sequences of symbols since it requires codewords for all possible sequences of that length. This causes the number of codewords to grow unmanageably large. The arithmetic coding technique provides a way of assigning codewords to particular sequences without having to generate codes for all sequences of that length.

Arithmetic coding generates a unique identifier called a *tag* for the sequence to be encoded. This tag is then assigned a unique binary code. A unique binary

(arithmetic) code can be generated for a sequence of a particular length $l$ without being compelled to generate codewords for all sequences of length $l$. This is the big advantage of arithmetic coding over Huffman coding. Generation of a Huffman code for a particular sequence of length $l$ requires the generation of codewords for all sequences of length $l$.

Arithmetic coding is more complex than Huffman coding. In cases where the alphabet is relatively large and the probabilities do not vary a great deal, Huffman coding might be a better solution. However, there are a number of sources, such as facsimile, in which the alphabet size is not very large and the probabilities vary greatly. In such cases, arithmetic coding would produce better results. Arithmetic coding has been recommended by the Joint Bi-Level Image Processing Group (JBIG) [8] as part of the standard for coding binary images.

### 3.1.3 Dictionary Techniques

Dictionary techniques incorporate the structure inherent in the data to achieve higher levels of compression. A dictionary of the most frequently occurring patterns is created and the code actually consists of the index of the pattern in the generated dictionary. For sources containing a relatively small number of recurring patterns, such as text data and computer commands, this method works extremely well.

Dictionary techniques may be divided into two main categories: static and adaptive techniques. Static techniques make use of a known data dictionary, which is

some statistical distribution of the source data, to accomplish compression. An initial pass over the data may be used to build the data dictionary which is used to encode the data in the second pass. Adaptive techniques, as the name suggests, adapt to the source data and construct the data dictionary on-the-fly. The adaptive dictionary techniques are of interest to us, in the context of this thesis. Most of the adaptive-dictionary-based techniques are derivations of the algorithms proposed by two Israeli researchers, Abraham Lempel and Jacob Ziv in their landmark papers in 1977 [9] and 1978 [10]. The Lempel-Ziv 77 algorithm (based on the 1977 paper) makes use of adaptive compression. The Lempel- Ziv method of compression is described in [12] as follows:

> "[The Lempel-Ziv 77 algorithm] makes use of adaptive compression - a kind of dynamic coding where the input is compressed relative to a model that is constructed from the data that has just been coded. By basing the model on what has been seen so far, the algorithm is able not only to encode in a single pass through the input file, but is also able to compress a wide variety of inputs effectively rather than being fine-tuned for one particular type of data such as English text."

The Lempel-Ziv 77 algorithm is also referred to as LZ77. In the LZ77 approach the dictionary entries are simply previously encoded sequences. Symbols are examined one at a time. Compression is performed symbolwise. The encoding consists of a length and an offset for a sequence of one or more symbols. The length

denotes the count of matching symbols in the sequence, and offset is the distance of the sequence being examined from a previous matching sequence. The dictionary itself is the source output. LZ77 assumes that the recurrence of a sequence is a local phenomenon.

The LZ77 algorithm is fairly fast in compression and decompression and the amount of memory used is moderate. It is a also a simple algorithm to implement. Both text and image data can be compressed easily and quickly. Popular compression packages like PKZip and Zip and Lharc all use an LZ77 based algorithm.

The Lempel-Ziv 78 approach (based on the 1978 paper) makes use of an explicit dictionary. The dictionary has to be generated by both the encoder and the decoder. The inputs are encoded as a pair. The first element is the index into the dictionary entry that was the longest match to the input. The second element is the code for the character in the input following the matched portion of the input. Each new entry into the dictionary is one new symbol concatenated with a previously existing dictionary entry. This has the drawback that the dictionary keeps growing without bound. To implement the LZ78 approach the growth of the dictionary has to be stopped at some point. Several modifications to the LZ78 approach have been suggested and of these the LZW algorithm is the most well-known [13]. Terry Welch suggested this modification to the LZ78. Welch proposed an encoding method that does not require the encoding to contain the second element, i.e. the code for the character immediately following the matched sequence. The encoding consists only

of an index into the dictionary. To begin with the dictionary is initialized with all the letters of the input alphabet. From then on the dictionary is dynamically constructed from patterns observed in the source output. The LZW algorithm is a popular variant of the Lempel-Ziv algorithm. The algorithms used in both UNIX COMPRESS and GIF use the LZW algorithm. The LZW algorithm provides good results for text compression as well as computer-generated graphical images. The presence of repetitive patterns in such data make them good candidates for compression using the LZW algorithm.

# CHAPTER 4. E-MAIL SYSTEM WITH DATA COMPRESSION

This thesis aims to compress E-mail data near the top network layer, thus reducing the amount of data that the network must transport. As such, efficient data compression can significantly boost overall network performance. Network bandwidth (or throughput) refers to the amount of data that can flow through a communication channel in a given unit of time. One obvious way of increasing bandwidth is to widen the communication channel by adding more network connections for a single transport. Another method to increase effective network bandwidth, and the one explored in this thesis, is to reduce the size of data that the network must transport. For example, if an effective data compression technique allows three mail messages to be reduced to size of one, three mail messages can be transported for the price of one. This would increase throughput by a factor of three. It is the goal of this thesis to achieve such an increase in effective bandwidth for an Internet E-mail system.

This chapter describes the design of the Internet E-mail system. The algorithm used to incorporate compression within the mail server is presented. The modification to the SMTP protocol necessary for negotiation of compression between two Internet Mail systems is also described.

## 4.1 PRELIMINARY INVESTIGATIONS

The study of the various lossless data compression techniques provided an insight into their strengths and weaknesses. In this thesis, the data compression mechanism will be employed to compress mail data prior to transmission. Since mail data may be in the form of text data or image data or speech data, the technique of choice, must be capable of compressing all these types of data efficiently. The file compression ratios for a variety of files, using the different compression schemes is shown in Table 4.1. Based on these results the LZ77 data compression scheme has been chosen for effective compression of mail data. The mail server has no prior knowledge of data and the LZ77 scheme does not require any either. Since its a simple adaptive scheme, that does not make any assumptions about data characteristics, it is suitable in that respect. It also has the advantage that it does not require large amounts of memory and demonstrates good speed and compression ratio for both small and large files. Mail data may vary greatly in size and this property makes it a good candidate for use in the Internet mail server.

| Original | | Huffman | | LZ77 | | LZW13 | | LZW15 | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Size | Size | Ratio | Size | Ratio | Size | Ratio | Size | Ratio |
| mbox.cpp | 45k | 30k | 66% | 9k | 20% | 18k | 40% | 14k | 31% |
| msrvr.exe | 449k | 324k | 72% | 151k | 34% | 453k | 101% | 266k | 69% |
| thesis.doc | 288k | 226k | 78% | 111k | 39% | 252k | 87% | 179k | 62% |
| rfc.txt | 76k | 47k | 61% | 10k | 14% | 29k | 38% | 30k | 39% |
| excite.htm | 13k | 9k | 71% | 3k | 25% | 7k | 52% | 5k | 37% |
| alska.htm | 6k | 4k | 69% | 1.5k | 20% | 3k | 50% | 2k | 40% |
| ftr.dll | 256k | 206k | 80% | 110k | 43% | 213k | 83% | 137k | 63% |
| res.001 | 317k | 224k | 70.8% | 67k | 21% | 311k | 98% | 106k | 33% |
| back.pcx | 65k | 37k | 56% | 27k | 42% | 38k | 58% | 33k | 51% |
| tt25.res | 1085k | 1010k | 93% | 778k | 72% | 1512 | 139% | 968k | 89% |

Table 4-1: File Size and Compression Ratios for Different Kinds of Files

## 4.2 Mail Server Architecture

The block diagram of the designed Internet e-mail system with embedded compression is shown Figure 4.1.

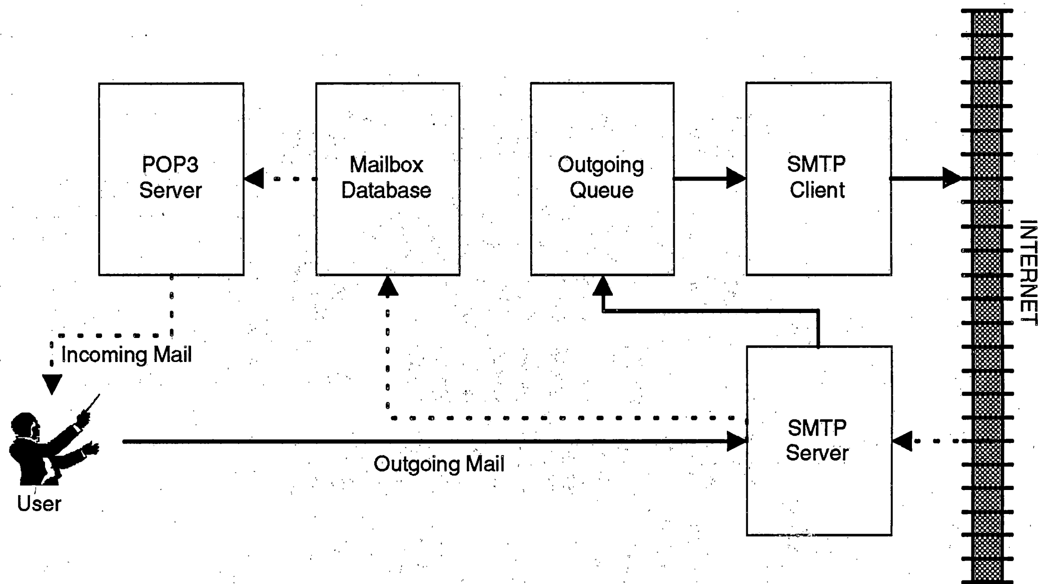


Figure 4.1: Internet Mail Server

Following is a brief description of the components of this e-mail system:

### 4.2.1 The Mailbox Database

The mailbox database contains a set of mailboxes, one per user (client) of the system. The incoming mail for the users are stored in their respective mailboxes. This mail could be sent by another user on the same mail system, or could be forwarded by another mail server on the Internet. Each mailbox can store any number of messages. These messages are stored in individual files - in uncompressed format. The mailbox database provides the following set of operations:

- Add a mailbox user
- Delete a mailbox user

- Authenticate a user
- Open a mailbox after proper authentication
- List the number of messages in the mailbox
- Add a message
- Retrieve a message
- Mark a message deleted
- Purge a message
- Peep into message
- Reset the mailbox
- Get UID for a message
- Close mailbox and unlock resources

### 4.2.2 Outgoing Queue

The outgoing queue contains all the Internet bound messages. These messages could be sent by the mail clients, or may be forwarded by another mail server on Internet. There is only one outbound queue in the mail system. The messages in this queue are stored in individual files. These files store uncompressed data. The outgoing queue provides the following set of operations:

- Put message in Queue
- Get message from Queue
- List number of messages in Queue

### 4.2.3 SMTP Server

The SMTP server communicates and receives the mail from either a mail client on the same host system, or from another mail system on the Internet. If the mail is bound to a mail user on the same host system, it is stored in the user's

mailbox. Otherwise, the mail is forwarded to the outgoing queue. If the same mail data is to be sent to multiple users on different hosts, the SMTP server replicates the mail message for each user in the outgoing queue.

Mail data is stored as a disk file and is ultimately sent to the address specified in the forward path. In case of any error in delivery, an undeliverable mail notification is sent back to the originator of the mail message using the reverse path stored in reverse path buffer.

It is the responsibility of the SMTP server to determine if the mail data coming from another mail server is compressed. If so, the SMTP server decompresses the data before sending it to either the outgoing queue or to the user's mailbox.

The operations provided by the SMTP server are:

- Accept connection from client SMTP
- Process MAIL command and store forward path
- Process RCPT command: check sequence and store reverse path
- Process DATA command: accept data and store or queue mail
- Process other SMTP commands like HELO, RSET, etc.

### 4.2.4 SMTP Client

The SMTP client keeps checking the outgoing queue at regular intervals. Messages on the outgoing queue are picked up by the SMTP client and sent to the appropriate mail server across the Internet. Before sending the messages, the SMTP client negotiates with the remote mail server to determine if the server supports

compression. If so, the SMTP client compresses the mail data. However, if the compressed data file size is larger than the original file size, the SMTP client sends the original data. The operations provided by the SMTP client are:

- Establish connection with server
- Send mail data by compiling and sending SMTP commands

#### 4.2.5 POP3 Server

The POP3 server is the primary interface between the mail user and his/her mailbox. In effect, the POP3 server interfaces with both the mail client and the mailbox database. It allows the mail clients to check the user's mailbox and download mail. It provides several operations on the mailbox. These operations include:

- Accept connection from client
- Authenticate mail user's name and password
- Open and lock mailbox for authenticated user
- Get status of user's mailbox from the mailbox database
- List mailbox messages
- Retrieve messages form mailbox database using message numbers.
- Delete messages from mailbox database
- Process all other POP3 commands like TOP, UIDL, RSET etc.

## 4.3 DATA COMPRESSION HANDLING IN THE MAIL SERVER

The designed mail server implements data compression transparently. The mail client sends mail data in any way it wants to the SMTP server. The SMTP server forwards this data to the outgoing queue. The SMTP client picks up this mail data from the outgoing queue and prepares to send it to the remote mail server across the Internet. It is at this stage that data compression takes place.

Similarly, the SMTP server receives (compressed) data from the Internet, and before sending it to another component, decompresses it.

As the above approach shows, all the data compression handling is encapsulated within the SMTP client / server components. There is no guarantee that the incoming (Internet) data is compressed data, or that the remote mail server could handle compressed data. Hence, the SMTP client and the server should be able to handle both compressed and uncompressed data. The SMTP client needs to know whether the remote mail server can handle compressed data. Only then can it send compressed data across the Internet. The SMTP server needs to publish its compression abilities. Even then, the incoming data could be uncompressed and it needs to know about it. To accomplish the above, an SMTP protocol extension is proposed and implemented.

### 4.4 SMTP Protocol Extension

SMTP Service extensions (RFC 1869) [14] provide a framework for extending the SMTP service by defining a means whereby a server may inform an SMTP client as to the service extensions it supports. Rather than describing the extension to the SMTP protocol required for incorporating data compression as a separate and haphazard entity, this framework was used to provide the enhancements in a straightforward fashion, consistent with all other extensions. In particular, this extension to the SMTP service allows compression of mail data prior to transmission using a commonly used compression technique.

RFC 1869 [14] introduces the EHLO SMTP command to be used instead of the HELO command by any SMTP client that supports the SMTP service extensions. A successful response by the SMTP server tells the client that it is able to perform the EHLO command. In case the server does not support the SMTP service extensions it will generate an error response. Normally, a successful response is a multiline reply, each line containing a keyword and optionally one or more parameters. These keywords denote the SMTP extensions that the server supports.

Consistent with RFC 1869 the definition of the compression extension is as follows:

1. the name of the SMTP service extension defined here is *compression*;
2. the EHLO keyword value associated with the extension is *XCOMP*;

3. the parameters used with the XCOMP EHLO keyword define the types of compression schemes supported by the server. At present only one scheme - LZ77 - is supported. Hence, there is only one keyword - *LZ77*. The syntax of the ehlo-line [14] using ABNF notation is as follows:

ehlo-line ::= "XCOMP" *( SP ehlo-param )

ehlo-param ::= "LZ77"

4. one optional parameter using the keyword *XCOMP* is added to the MAIL FROM command. The value associated with this parameter is a keyword indicating the compression scheme being used for compressing the mail data that is being sent. At present only one compression scheme is supported by the system. The syntax of the optional esmtp parameter using ABNF notation is:

esmtp-parameter ::= "XCOMP=" xcomp-value

xcomp-value ::= "LZ77"

5. no additional verbs are defined for this extension; and,

6. the next section specifies how support for the extension affects the behavior of a server and client SMTP.

The client SMTP that wishes to send compressed mail data should start an SMTP session with the extended SMTP service command EHLO. If the SMTP server responds with code 250 to the EHLO command and the reply includes the EHLO keyword value XCOMP followed by the parameter value LZ77, then the

server is indicating that it supports the extended MAIL command and supports the LZ77 compression scheme. At this point, the SMTP client is authorized to send LZ77 compressed data.

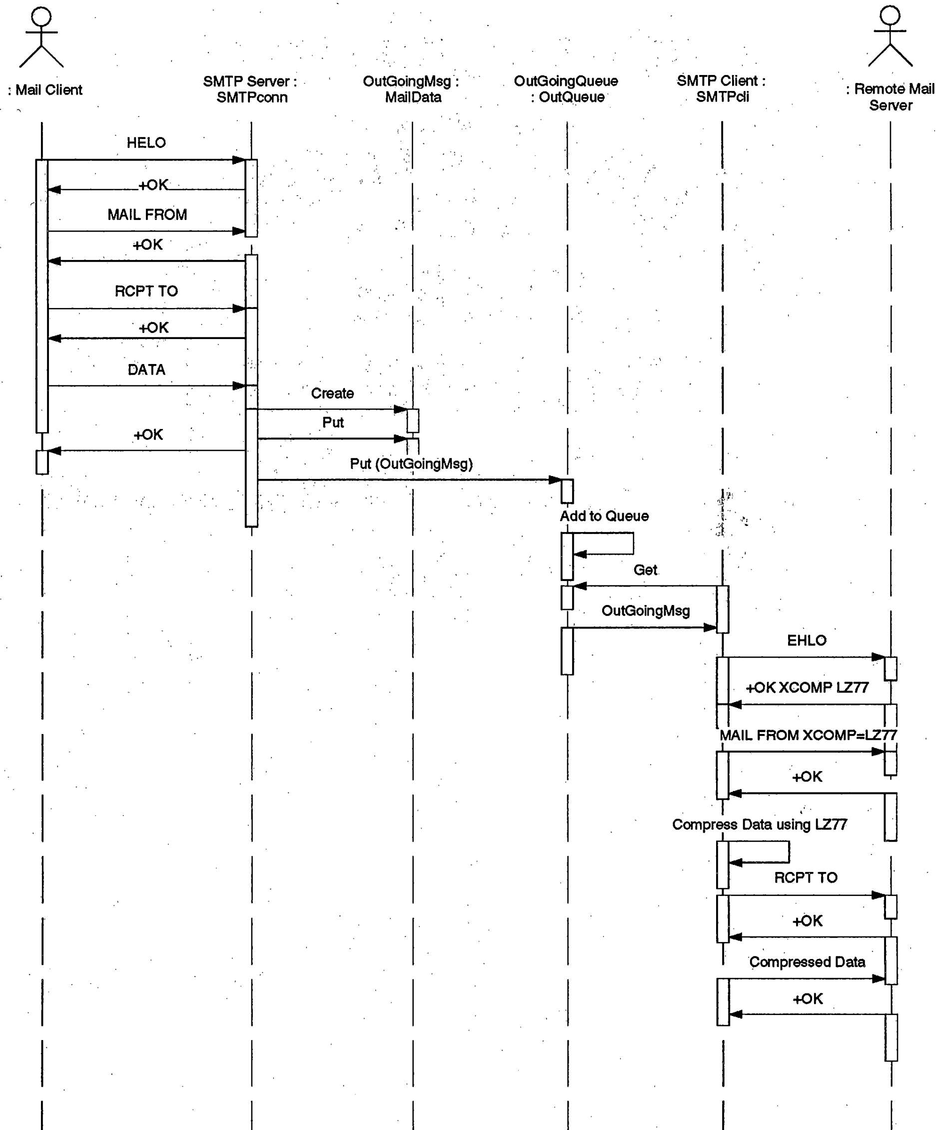


Figure 4.2: Outgoing Mail Use Case Diagram

If the client wishes to transmit LZ77 compressed data, it issues the extended MAIL command. The syntax for this command is identical to the SMTP MAIL command defined in [4] except that a XCOMP parameter must appear after the mail originator's address. Only one XCOMP parameter may be used in a single MAIL command. The value associated with the XCOMP parameter indicates that the mail data will be compressed using this (LZ77) algorithm. Although this information may seem redundant in the present e-mail system since it supports a single compression scheme, this design allows for multiple compression schemes in the future. The SMTP client compresses the data only after receiving a successful response from the server. The client then issues a DATA command to the server and promptly after receiving a successful response, sends the compressed mail data, terminating it with the usual <CRLF.CRLF> sequence. If a server SMTP does not support the SMTP compression extension (either by not responding with code 250 to the EHLO command, or by not including the EHLO keyword value XCOMP in its response), then the client SMTP does not compress the mail data but rather sends it as is.

The extended SMTP server accepts both HELO and EHLO commands. When it receives an EHLO command, it replies back indicating the compression schemes it supports. If it receives a XCOMP parameter in the MAIL FROM command, it understands that the SMTP client is sending compressed data. The SMTP server accepts the compressed data, and then decompresses it on the fly. If the SMTP server

does not support the compression scheme indicated in the MAIL FROM command, it returns an error.

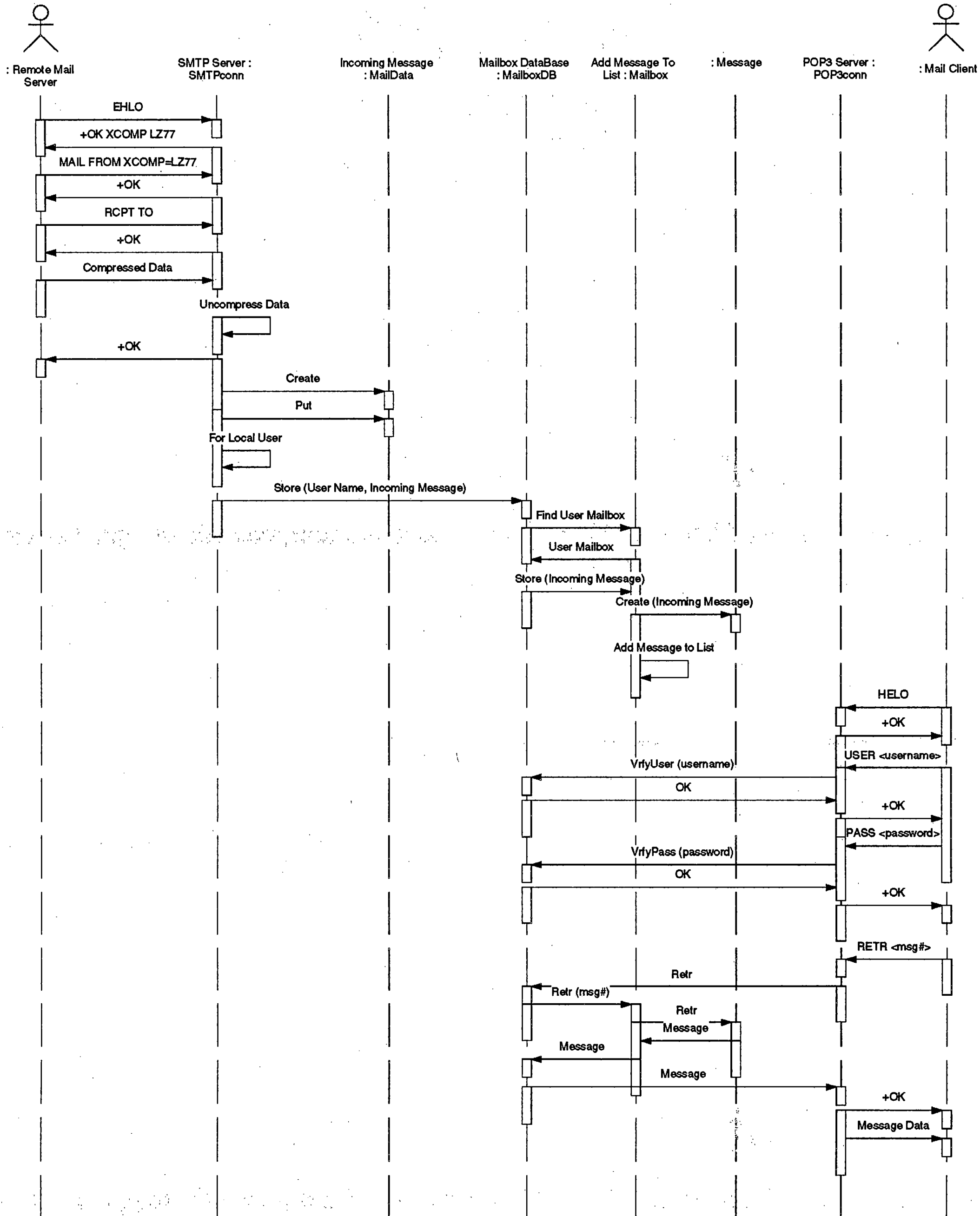


Figure 4.3: Incoming Mail Use Case Diagram

The following dialogue illustrates the use of the compression service extension:

```
R: 220 BBN-UNIX.ARPA Simple Mail Transfer Service Ready
S: EHLO USC-ISIF.ARPA
R: 250-BBN-UNIX.ARPA
R: 250 XCOMP LZ77

S: MAIL FROM:<jane@USC-ISIF.ARPA> XCOMP=LZ77
R: 250 OK

S: RCPT TO:<Jones@BBN-UNIX.ARPA>
R: 250 OK

S: RCPT TO:<Green@BBN-UNIX.ARPA>
R: 550 No such user here

S: RCPT TO:<Brown@BBN-UNIX.ARPA>
R: 250 OK

S: DATA
R: 354 Start mail input; end with <CRLF>.<CRLF>
S: LZ77 compressed mail data sent here .....
S: ...etc. etc. etc.
S: .
R: 250 OK

S: QUIT
R: 221 BBN-UNIX.ARPA Service closing transmission channel
```

Figure 4.4: Sample Transaction with SMTP Compression Service Extension

## 4.5 IMPLEMENTATION DETAILS

An Object Oriented approach was used to design the e-mail system. The various components of this system were treated like loosely coupled objects. Every component was designed keeping in mind the need to preserve data integrity and consistency. No loss of mail is allowed by the server. The e-mail system consists of three major threads, namely the SMTP server, the SMTP client and the POP3 server. Threads are light weight processes [1]. The Mailbox Database and the Outgoing

Queue are the two other major classes. The following figure shows the major classes and their relationships.

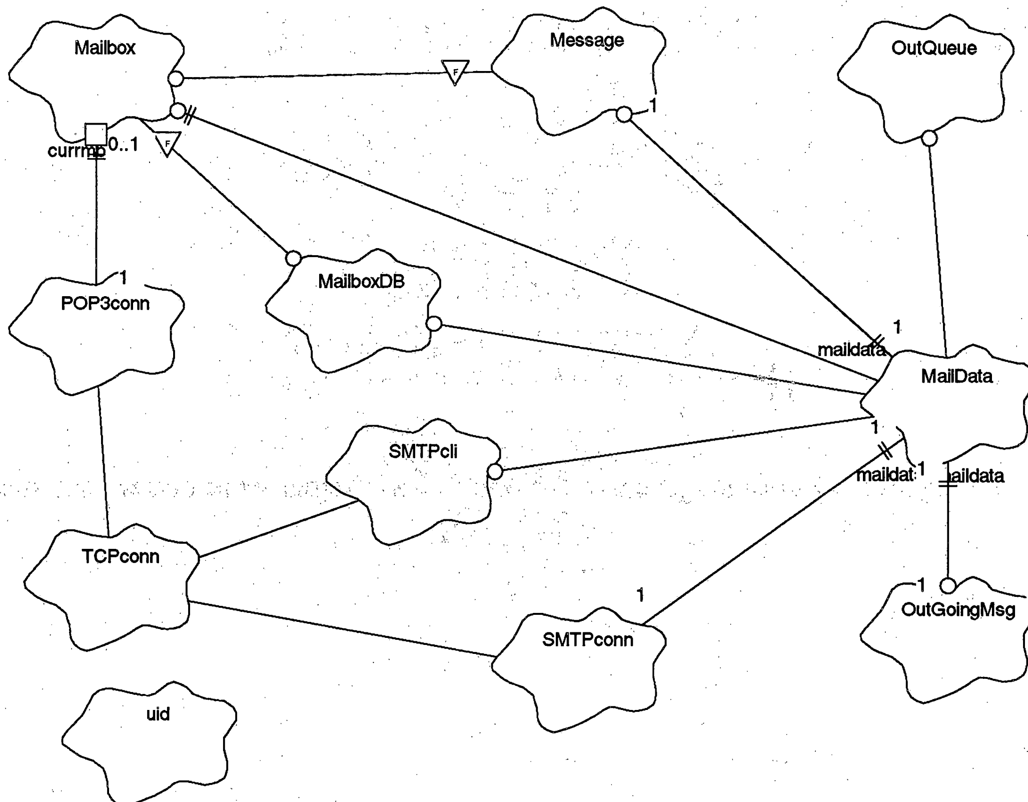


Figure 4.5: Main Class Diagram

The Mailbox Database is of particular importance in the e-mail system. It serves as a repository for mail data. Figure 4.1 provides a block diagram of the Mailbox Database design. The *MailboxDB* class provides the interface to the mailbox database.

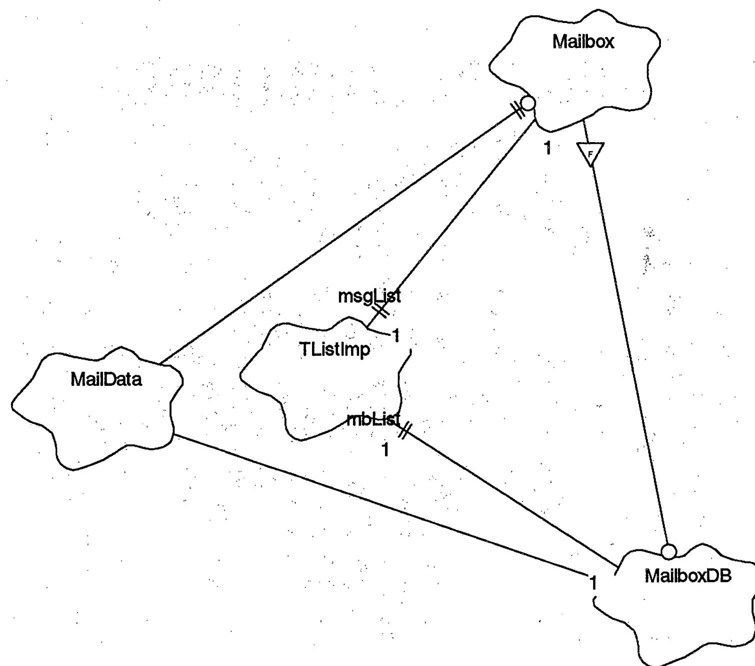


Figure 4.6: MailboxDB Class Diagram

There is a single global instance of the MailboxDB class. *MailboxDB* contains a list of *Mailbox* class objects. Each time a new mailbox user is added, a new *Mailbox* class object would be created and added to the list. Each *Mailbox* class object maintains a list of *Message* class objects. An instance of the *Message* class is created and added to the list every time a new message is stored in the mailbox. The message data is stored as *maildata* objects. The *maildata* class manages the mail data, allowing reading and writing of mail data in segments. Storing mail data in a disk file saves the overhead of excessive memory usage. Also, this imposes no limits on the length of mail data. The mailbox database is also stored on disk and every time the system is rebooted, all database files are uploaded. Every time the mailbox

database changes state the new state is stored on disk. This has the advantage that no data is lost even if the system is shut down temporarily.

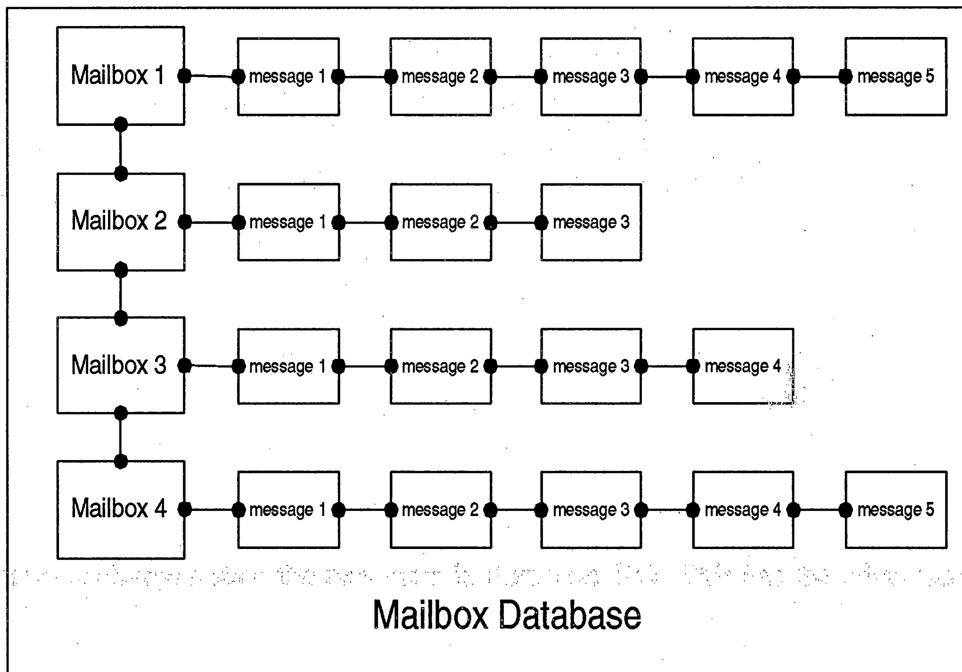


Figure 4.7: Mailbox Database Design

*MailboxDB* maintains exclusive access to the mailboxes. Mutexes [20] are used to synchronize data access to the mailbox database across the multiple processes/threads.

The SMTP server is implemented as a thread that is started at system startup. The server process creates a socket and then configures it using the local IP address and the SMTP protocol port. The SMTP server then waits for client requests. Whenever the server receives a new connection from an SMTP client, it starts a child

thread that creates an instance of the *SMTPconn* class. The *SMTPconn* class has been derived from the TCP connection class *TCPconn*. *TCPconn* manages all the TCP connection functions such as read and write to socket. The *SMTPconn* class contains methods to process all the SMTP commands received from the client.

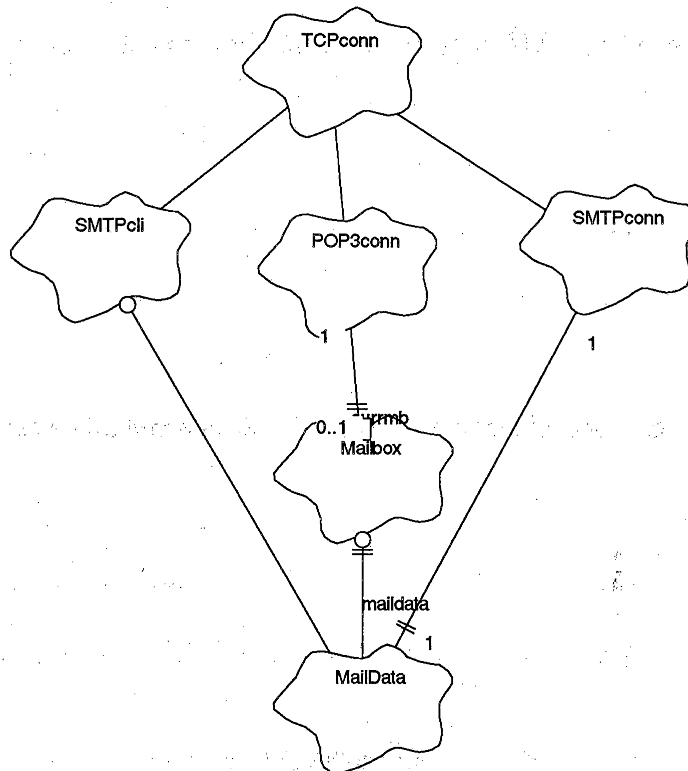


Figure 4.8: TCP Class Diagram

Once a connection has been accepted the mail server reads in a command at a time and invokes the appropriate *SMTPconn* method to process the command. Since a mail transaction involves multiple steps, the commands are parsed and arguments are treated as data objects to be used in future processing. The argument to the MAIL command is the reverse-path which needs to be held pending not only for insertion at

the beginning of the mail data as the return path line (in case of final delivery of data), but also for sending an "undeliverable mail" notification to the originator in case the mail could not be delivered. Similarly, in case of multiple recipients for the same mail data, both the forward path and reverse path need to be preserved till the end of the mail transaction. The SMTP supports the extended SMTP service extensions command EHLO and the SMTP compression service extension. If the client begins a transaction with the HELO command the server treats the mail data received with the DATA command as normal uncompressed data. However, if the client sends the EHLO command at the onset of the mail transaction, the SMTP server responds by sending 250 reply code along with the compression keyword XCOMP. Only in such cases does the server expect the mail data to be in a compressed format. The argument to the MAIL command is also checked to make sure the compression scheme specified is supported by the server. The data following the DATA command is then decompressed using this same scheme. In any case the server processes the stored mail transaction information. If the mail is for a local recipient, the SMTP server just puts it in the users mailbox using the Mailbox database interface. In case of a remote client, the mail needs to be sent across the Internet. In such cases, it is the responsibility of the SMTP server to put the mail message along with the stored mail transaction information in an outgoing queue.

The outgoing queue is a global object of the class *OutQueue*. The *OutQueue* class maintains a queue of messages that need to sent to remote users. New messages

are added at the tail of the queue, while messages are retrieved from the head of the queue. All *OutQueue* data is saved to disk every time it changes state. For example every time a new message is added to the queue, the copy of the queue on disk is updated. This ensures no loss of data even if the system goes down. The SMTP client thread checks the *OutQueue* periodically for pending messages to be transmitted on the Internet. Whenever it detects a new message in the queue, the SMTP gets it and prepares to set up a TCP connection with the SMTP server of the remote machine. Once a message has been retrieved from the *OutQueue*, it is removed from the queue.

The SMTP client thread is another independent process that is created at startup. Just like the SMTP server keeps listening for a TCP connection, the SMTP client keeps waiting for outbound messages to appear in the queue. Each message waiting to be sent to a remote mailbox is processed in the order it was put into the queue. An SMTP client object belonging to the class *SMTPcli* is created. This handles all SMTP connection details right from establishing a connection with the remote server to sending the mail data and terminating the connection. The forward path specified in the mail message is used to get the hostname of the recipient. If the SMTP client is unable to set up a direct connection with the host, it tries to connect to one or other of a set default hosts. Upon successful connection, the client begins the mail transaction with the extended SMTP EHLO command. If the SMTP server at the other end supports the SMTP service extensions it will give a positive response along with the extensions it supports. The client then parses the reply to determine if

the server will accept a compressed file. SMTP commands are compiled by extracting arguments from the mail message, and sent to the server in the proper sequence. The compression scheme to be used is conveyed to the server as an argument of the DATA command and if acceptable, the mail data is compressed before transmission. Finally, the QUIT command terminates the connection.

The POP3 server process also begins at system startup and analogous to the SMTP server spends most of its time listening for a new POP3 connection from a client. It begins a new thread for each new connection. The POP3 connection thread creates an instance of the class *POP3conn*. *POP3conn* is also derived from the class *TCPconn* and inherits all TCP connection attributes and functionality from it. For each POP3 command *POP3conn* contains a function that does all the processing associated with it. POP3conn class interfaces with the Mailbox Database to access mailboxes and mail messages. *POP3conn* provides a function whose sole purpose is to accept client commands, parse them and accordingly invoke the appropriate function. The POP3 connection progresses through three states in the duration of a transaction. Initially it is in the *authorization* state in which it only accepts the USER, PASS and QUIT command. The arguments to the USER and the PASS command are used to authenticate the user. The POP3 connection then acquires exclusive access to the mailbox, assigns a message number to each message and enters the *transaction* state. The commands STAT, LIST, RETR, DELE, RSET, TOP, UIDL and NOOP are accepted in this state. For each of these commands the POP3 connection interfaces

with the mailbox database to acquire the required information to pass back to the POP3 client. Finally it enters the *update* state when the client POP3 issues a QUIT command. Messages marked deleted are purged and the status of the mailbox at the time of closing is returned to the client. The POP3 connection releases any exclusive lock on the mailbox.

# CHAPTER 5. PERFORMANCE EVALUATION

## 5.1 COMPARING TRANSMISSION SPEEDS

Different kinds of mail data was used to test the E-mail system. Textual data, binary data, and graphical data was transported across the Internet using the designed E-Mail system. Several Windows NT hosts were identified for this experiment. These hosts were connected with Internet. On each of the hosts, two E-Mail systems were installed - the implemented E-Mail system with embedded compression, and a standard E-Mail system. Mail data was sent from one host to another. First the mail data was transferred from one host to another using the standard E-Mail system on the receiving end, and the designed E-Mail system on the transmitting end. The uncompressed mail data was transmitted in this case. The designed E-Mail system displays the time taken to transmit the mail data using 'time' system call. This time was noted. Then the designed E-Mail system with embedded data compression was started at both the hosts and the same mail data was transported between the same hosts. The compressed mail data was transmitted this time. The time taken in the transfer was again noted. This time included the time to compress data. This exercise was repeated ten times for each file and each pair of connected mail hosts. Again, the same test was repeated at different times in the day. Finally, an average transmission

time was computed for each file for transmission with compression, and for transmission without compression.

In all cases the time taken to E-Mail a file using the designed server was considerably less than the time taken to E-Mail the same file over the standard E-Mail system. This reduction in time is attributed to the data compression employed by the designed mail server. The table below shows the size of the file and the time taken on both the standard E-Mail system and the designed E-Mail system. The ratio between these two times demonstrates the speed up.

| **File Name** | **File Type** | **Size** | **Average Time on Standard E-Mail System** | **Average Time on Designed E-Mail System** | **Average Transmission Time Ratio** |
|---|---|---|---|---|---|
| mbox.cpp | Text File | 45k | 63 sec | 41 sec | 65% |
| msrvr.exe | Binary Executable | 449k | 527 sec | 379 sec | 72% |
| thesis.doc | MS-Word Document | 288k | 317 sec | 190 sec | 60% |
| rfc.txt | Text File | 76k | 81 sec | 38 sec | 47% |

| **File Name** | **File Type** | **Size** | **Average Time on Standard E-Mail System** | **Average Time on Designed E-Mail System** | **Average Transmission Time Ratio** |
|---|---|---|---|---|---|
| excite.htm | HTML Text | 13k | 45 sec | 25 sec | 55% |
| alska.htm | HTML Text | 26k | 160 sec | 91 sec | 57% |
| ftr.dll | Binary DLL | 256k | 276 sec | 220 sec | 80% |
| res.001 | Binary | 317k | 332 sec | 235 sec | 71% |
| back.pcx | Binary Graphics | 65k | 68 sec | 38 sec | 56% |
| tt25.res | Binary Resource | 1,085k | 1345 sec | 672 sec | 50% |
| ***Average*** | | ***260k*** | ***309 sec*** | ***185 sec*** | ***59%*** |

Table 5-1: Comparison of Transmission Time

As illustrated by the table, the compressed mail data took considerably less time to reach its destination. The reduction in time has to be attributed to the compression of data being performed by the mail server. Although network factors may affect transmission speeds, the consistent reduction in transmission time

indicates the superiority in performance achieved by the mail server with embedded data compression.

## 5.2 TASKS ACCOMPLISHED

The specific tasks that were achieved during the thesis are listed.:

1. The effect of incorporating data compression before transmission was studied. Mail data was compressed at the user level prior to transmission. The reduction in transmission time of a compressed file indicated the advantage of incorporating data compression within the mail server.

2. Different data compression algorithms were studied and evaluated. LZ77 was considered the most suitable for use in the Internet Mail server.

3. The E-Mail system was designed using Object-Oriented methodology. Each component of the system was treated as an object and the functionality and relationships were identified.

4. The Post Office Protocol Version 3 (POP3) clients and servers were designed and implemented using Object-Oriented methodology. POP3 is used to retrieve mail.

5. The Simple Mail Transfer Protocol (SMTP) clients ands servers were designed and implemented.

6. Based on previous evaluations, the LZ77 compression scheme was incorporated within the SMTP clients and servers. The SMTP protocol was extended to allow both a mail client and server that support compression to recognize each other as such and to negotiate compression between the two.

7. The various components were integrated and the fully operational E-Mail system was tested using a number of different kinds of mail data files.

8. A simple User Agent (UA) that allows addition of mailbox users was designed and implemented.

9. The performance of the designed system was evaluated by comparing it with the existing E-Mail systems. As expected the implemented mail server demonstrated enhanced transmission speeds.

10. Future enhancements were identified. The E-Mail system may be further improved to support more than one compression scheme. The SMTP protocol extensions necessary for such a system were designed.

# CHAPTER 6. FUTURE ENHANCEMENTS AND CONCLUSION

The E-Mail system may be further enhanced by incorporating certain extensions. These enhancements are described in this chapter.

## 6.1 ENHANCEMENTS TO DESIGNED SERVER

### 6.1.1 Allowing Multiple Compression Schemes

The SMTP service extensions for compression could be further enhanced by including other compression schemes such as Huffman coding, LZ77 and arithmetic coding. In such a case the EHLO keyword value associated with the compression extension would still be XCOMP, but the parameters associated with this could have multiple keyword values. The syntax of the value using ABNF notation would then be:

xcomp-value ::= ("LZ77") * (SP "LZ77" ) * (SP "HUFF")

* (SP "ARITH")

For instance, an EHLO line such as, *250 XCOMP LZW LZ77 HUFF*, would imply that the server supports the compression service extension and is capable of dealing with data that has been compressed using either the LZW compression scheme or the LZ77 scheme or the Huffman coding scheme. The client would then have a choice of compression schemes to choose from. For each mail transaction, the

particular scheme being used to compress the ensuing mail data would be specified in the extended MAIL command.

In particular, one optional parameter using the keyword XCOMP may be added to the MAIL FROM command. The value associated with this parameter would then be a keyword indicating the specific compression scheme (from the ones supported by the server) being used to compress the mail data being sent with the DATA command. The syntax of the value using ABNF would then be:

xcomp-value ::= "LZW" / "LZ77" / "HUFF" / "ARITH"

The SMTP server would then expect mail data compressed using the specified scheme. The server would decompress the data before storing.

### 6.1.2 Automatic Selection

Another useful feature would be automatic selection of the most optimal algorithm. This would require the system to analyze the data and according to the structure of the mail data, determine which compression scheme would produce the best results. Depending upon the type of mail data being transferred, a suitable scheme would be selected. In such a scenario, text mail data could be compressed using the LZW scheme and Bi-level image data could be coded very effectively using arithmetic coding.

## 6.2 Extending The Design to Other Servers

The concept of embedding data compression within the mail server could be extended to other servers such as the FTP server to compress file data before transmission. The FTP protocol would need to be extended to incorporate compression. The FTP server would check the file being transferred and if it is not already in compressed format, the server compress it before sending it across the Internet.

## 6.3 Conclusion

In conclusion, incorporating the task of data compression within the E-Mail system achieves the goal of increasing effective transmission bandwidth and reducing network traffic. By embedding an efficient data compression scheme within the mail server, the time for transmitting mail data across the Internet, is significantly reduced. If all mail servers were designed to handle data compression, it would result in a substantial boost in overall network performance. Network bandwidth refers to the amount of data that can flow through a communication channel within a given period of time. Since the designed Internet mail server with embedded data compression reduces the size of data that the network must transport, it helps in increasing the effective bandwidth. Mail data compression also helps in reducing network traffic congestion. An Object Oriented approach is effective in designing

such a system. Contemporary Windows features like multi-threading and DLL’s enhance the flexibility and stability of the system.

APPENDIX A: MAJOR CLASSES

The C++ code for the thesis may be obtained from the author or from Dr. Tong Yu. The author may be contacted at alka_nand@ftw.paging.mot.com. Dr. Tong Yu may be contacted at tongyu@csci.csusb.edu.

```
//******************************Message class ****************************

class Message                  //Manages messages stored for each mailbox
{
    friend class Mailbox;
    private:
    //Data
        enum    status         //message marked as deleted or notdeleted
        {
            deleted = 1,
            notdeleted = 0
        };

        int     msgnumber;     //number of message in mailbox
        status  stflag;        //status of message: deleted/notdeleted
        char    *username;     //mailbox user owning message
        char    *msgfile;      //name of disk data file
        MailData maildata;     //mail data object
        char    msguid[MAXLENOFUID];   //The Unique Id for the message

    //functions

        int     retrmsg(MailData & md );
                               //compiles response to RETR cmd
        int     delemsg( );//mark as deleted current message
        char    *retrmsgid( char * ) const;   //gets the unique Id for this msg

    public:

        //Constructors
        Message();
        Message( char *usrnam, int msgno, MailData maildata );
        Message( const Message &message); //copy constructor
        Message( char *usrnam, int msgno ); // for loading from file

        //Destructor
        ~Message( );

        //Operators
        const Message &operator=( const Message &message);
        BOOL operator==( const Message &message) const;

        int     listmsg(MSGLIST *msglist) const ;
        int     rsetmsg( );        //unmark message if marked deleted
        int     vrfymsgno( int msgno ) const;
                                   //verify if msgno matches msgnumber
                                   //compiles response to list POP3 command
        int     retrmsgnumber( ) const { return msgnumber;}
                                   //retrieves message no.
        int     isdeleted( ) const { return stflag; }
                                   //returns status of message
                                   //(deleted/notdeleted)
```

```
        long    retrsize(void);         //retrives size of message
        int     modifymsgnum( int msgno );
                                        //changes msgnumber to msgno
        int     rmfile();               //Removes data file if msg marked deleted

        void    printmsg( void ) const;
        int     savemsg(void);          //Saves messzage data in file
        int     loadmsg(const char *filnam, int fdread );
        int     retruid( MSGUID *uidstruct ) const;    //get uid of msg

};


//*****************************Mailbox class ***************************

class   Mailbox             // Manages the mailbox for a particular user
{
    friend class MailboxDB;
    private:

        //Data

        enum        state       //state of mailbox
        {                       //[autorization or //transaction or update]
            authorization,      //still verifying mailbox user
            transaction,        //mailbox user identified and mailbox openend
            update              //this state entered when user issues quit
                                //when in update mode
        };
        char    *username;      //name of mailbox user
        char    *password;      //password of mailbox user
        state   currstate;      //current state of mailbox
        long    size;           //size of mailbox in octets
        int     noofmsgs;       //no. of messages in mailbox
        char    *mbfilename;    //Name of Mailbox data file(=username.mbx)
        BOOL    storeflg;       //==1 => message can be stored in mailbox
                                //even if it is not in transaction state
                                //Reqd to allow smtp server to store msgs

        TListImp < Message > msgList;//need to instantiate a list container
                                    //called msglist to maintain list of
                                    //messages

        //Functions
        int     loadmb(const char *filename); //Load Mailbox data from file
        int     savemb(void) const;           //Save Mailbox data into file
        int     storemb( MailData maildata );
        int     listmb(MBLIST *mblist) ;//returns listing of mailbox
        int     listmbmsg(int msgno, MSGLIST *msglist) const;
        int     statmb(MBSTAT *mbstat) const;//return status of messages
                                        //return listing of msgno
        int     retrmb(int msgno, MailData & maildata) const;
                                        //retreive the massage for msgno
        int     delemb(int msgno);
                                        //mark as deleted the specified msgno
        int     rsetmb();               //unmark all messages marked deleted
        int     quitmb( MBSTAT *mbstat);//if in transaction state removes msgs
                                        //marked as deleted from mailbox
        int     retrnoofmsgs( ) const { return noofmsgs; }
        void    changetotxstate( ) { currstate = transaction; }
                                        //change to transaction state
        int     assignmsgnomb( );       //Assign a msgno to each msg in mbx
        char    *getfilename(char *filename) const;  //Returns name of data
                                        //file in filename
        int     uidmsg( int msgno, MSGUID *uidstruct ) const;
                                        //get uid for specified msg nummber
        int     uidlist( MBUID *uidlist ) const;
                                        //get uid for all msgs in mb
        void    setstoreflgon( ) { storeflg = TRUE; }
                                        //Set store flag on for storing
        void    setstoreflgoff( ) { storeflg = FALSE; }
```

```
                                    //Set store flag OFF

    public:

        //Functions

        //Constructors
        Mailbox ();             //Default constructor
        Mailbox ( const char *usr, const char *passwd );
                                  //type conversion constructor
        Mailbox ( const char *filename);
                                  //type conversion constructor that
                                  //loads data from file
        Mailbox ( const Mailbox &mb); //Copy constructor

        //Destructor
        ~Mailbox ( );

        void    printmb(void) const;

        //Operators
        const Mailbox &operator=(const Mailbox &mb);
        BOOL     operator==(const Mailbox &mb) const;

        BOOL    vrfyusrmb( const char *usr) const;    //verify user
        BOOL    vrfyusrmb( const char *usr, const char *passwd) const ;
                                    //verify user with this passwd

};


//*****************************MailboxDB class ****************************

class MailboxDB
{
    private:

        //Data

        int noofmailboxes;              //no of mailboxes in database
        char *dbfile;                   //name of database file;
        TListImp < Mailbox > mbList;  //need to instantiate list container
                                        //called mbList to maintain list of
                                        //Mailboxes

        //Functions
        public:

        //Constructor
        MailboxDB() ;
        MailboxDB(int dummy);

        //Destructor
        ~MailboxDB() { free(dbfile);}


        int createMB( const char *usrname, const char *passwd);
                                    //creates a new mailbox
        int deleteMB( const char *usrname, const char *passwd);
                                  //deletes a mailbox
        int saveMBDB( );                //saves Mailbox database onto disk
        int loadMBDB( );                //loads Mailbox database from disk
        void printMBDB( void );

        BOOL    vrfyuser( const char *usrname )const; //verify username
        Mailbox *vrfypass( const char *usrname, const char *passwd );
                                    //verify name and pass
        int store(  Mailbox *currmb, MailData maildata );
                                    //store message for mailbox currently
```

```
                                        //being usedfor transaction
        int store( const char *usrname, MailData maildata );
                                        //store message in mbx for <usrname>
        int stat( const Mailbox *currmb, MBSTAT *status  );
                                        //get status of mailbox currently
                                        //being used for transaction
        int list( Mailbox *currmb, MBLIST *listing );
                                        //list of messages in mailbox
                                        //currently being used for transaction
        int listmsgno( const Mailbox *currmb, int msgno, MSGLIST *listing );
                                        //list of mess msgno in mailbox
                                        //currently being used for transaction
        int retr( const Mailbox *currmb, int msgno, MBRETR *retrmsg );
                                        //retreive message msgno from mailbox
                                        //currently being used for transaction
        int dele( Mailbox *currmb, int msgno);
                                        //mark as deleted msgno from mailbox
                                        //currently being used for transaction
        int rset( Mailbox *currmb );
                                        //unmark deleted messages in mailbox
                                        //currently being used for transaction
        int quit( Mailbox *currmb, MBSTAT *mbstat );
                                        //if in transaction state removes msgs
                                        //marked as deleted from mailbox
                                        //currently being used for transaction
        int quit( );                    //Not in transaction state- just
                                        //send quitting message to user
        int uidmsg( const Mailbox *currmb, int msgno, MSGUID *uidstruct ) const;
                                        //get uid for specified msg nummber
        int uidlist( const Mailbox *currmb, MBUID *uidlist ) const;
                                        //get uid for all msgs in specified mb
};


/***************************************************************************
    This is the superclass for the TCP connection classes e.g. POP3
    and SMTP. This is an abstract class.
***************************************************************************/

class TCPconn
{
    protected:                      //allow subclasses to inherit data

    //Data

    char        inbuf[MAXBUF+1];   // Buffer to store incoming data
    char        outbuf[MAXBUF + 1];    // Buffer to send outgoing data
    int         sockhnd;           // socket descriptor for connection
    int         bytecount;         // Keeps count of bytes read from sock


    //functions

    void    init(void);            // Does all the initializations
    int     read_data(void);       // reads data from socket
    int     write_data( int n );   // writes data to socket

    public:

    TCPconn(void) { init(); };
    TCPconn( int sock );
    ~TCPconn(void) { };
    int     connEstbGreeting(void);    // Send a greeting message to client

};


/*******************************************************************************
```

```
   The class definitions for the POP3CONN (POP3 connection) are contained
   herein. All data and functions pertaining to a new POP3 connection recd.
   by the server are handled by this class.
*****************************************************************************/

class POP3conn                         // New POP3 connection
{
   private:

   //Data

   static const CMDTBL   cmdtbl[];
   const int MAXNUMCMDS;
   static const char sepstr[];     // chars used to separate words
   enum state                      //state of POP3 connection -
   {                               //authorization or transaction or update]
      authorization,               //still verifying mailbox user
      transaction,                 //user identified and mailbox openend
      update                       //this state entered when quit command recd.
   };
   state       currstate;          //current state of mailbox
   char        inbuf[MAXBUF+1];    // Buffer to store incoming data
   char        outbuf[MAXBUF + 1];// Buffer to send outgoing data
   int         cmdno;              // command no. of cmd being processed
   int         sockhnd;            // socket descriptor for connection
   Mailbox     *currmb;            // mailbox being accessed currently
   int         bytecount;          // Keeps count of bytes read from sock
   char        *username;          // mailbox user currently being accessed
   char        *password;          // passwd of user currently being accessed


   //functions

   void    init(void);             // Does all the initializations
   int     read_data(void);        // reads data from socket
   int     write_data( int n );    // writes data to socket
   int     usercmd(void);          // processes USER command
   int     passcmd(void);          // processes PASS cmd
   int     quitcmd(void);
   int     statcmd(void);
   int     listcmd(void);
   int     retrcmd(void);
   int     delecmd(void);
   int     noopcmd(void);
   int     rsetcmd(void);
   int     topcmd (void);
   int     uidlcmd(void);
   int     notOK(void);

   public:

   POP3conn(void);
   POP3conn( int sock );
   ~POP3conn(void) { };
   int     connEstbGreeting(void);    // Send a greeting message to client
   int     getcmd(void );          // recv cmd from socket and parse it

};

/*****************************************************************************
   The class definitions for the SMTPcli (SMTP client) are contained
   herein. All data and functions pertaining to a SMTP client are
   are handled by this class. A new SMTP client object is created by
   the SMTP client process everytime it discovers that a message has
   to be sent to a remote recipient. The SMTPcli object then takes care
   of communicating with the receiver SMTP server and transmits the
   message.

*****************************************************************************/
```

```
class SMTPcli      :public TCPconn
{
   private:

   //Data
   bool       isLZ77comp;          // LZ77 compression supported
   static const SMTPCMDS smtpcmds[];
   const int MAXNUMCMDS;
   char    hostname[MAXHOSTNAMELEN];

   //functions

   int     call_socket( const char * );  //Tries to connect to server

   public:

   //functions

   SMTPcli( );
   int     sendmail( const char *, const char *, MailData &);

};


//****************************************************************************
//  The class definitions for the OutGoingMsg are contained
//  herein. All data and functions pertaining to a new outgoing message that
//  is to be transmitted to a remote site are handled by this class.
//****************************************************************************

class OutGoingMsg     // A new outgoing message to be added to outgoing Q
{
   private:

   //Data

   char    *reversepath;       // Path to be used For replying to sender
   char    *forwardpath;       // path of mail recievers
   MailData maildata;          // mail data
   int     msgnumber;          // message number

   //functions

   int     savemsg(void);                //Saves messzage data in file
   int     loadmsg(const char *, int );  //load msg data

   public:

   //functions

   void    putrevpath( const char *revpath );
   void    putfwdpath( const char * );
   char    *getfwdpath( char * );
   long    getsizeofmsg( ) {return maildata.GetSizeofMailData(); }
   int     getmsg( char *, char *, MailData & );    //Returns msg data
   BOOL    operator==(const OutGoingMsg ) const;
   const OutGoingMsg  &operator=( const OutGoingMsg & );
   int     getfilename( char *filename );

   //Constructors
   OutGoingMsg( );
   OutGoingMsg( int msgno );     //This loads file from disk
   OutGoingMsg( const char *, const char *, const MailData &, int);
                              //takes fwdpath, rev path, & data as input
   OutGoingMsg::OutGoingMsg( const OutGoingMsg &msg);  //copy constructor
   //Destructor
   ~OutGoingMsg();

};
```

```
/*****************************************************************************
    The class definitions for the SMTPCONN (SMTP connection) are contained
    herein. All data and functions pertaining to a new SMTP connection recd.
    by the server are handled by this class.
*****************************************************************************/

class SMTPconn     :public TCPconn       // New SMTP connection
{
    private:

    //Data

    static const CMDTBL    cmdtbl[];
    const int MAXNUMCMDS;
    static const char sepstr[];   // chars used to separate words
    int         cmdno;             // command no. of cmd being processed
    char        *senderSmtp;       // name of SMTP-sender (Parameter to HELO)
    char        *reversepath;      // Path to be used For replying to sender
    char        *forwardpath;      // paths of mail recievers
    bool        isLZ77comp;        // LZ77 compression supported
    bool        data_is_compressed;   // incoming data compressed
    MailData    maildata;          // mail data
    int         prevcmdno;         // Prev cmd reqd to check proper sequence
                                   // of cmds ( Mail-RCPT-DATA)
    int         nooflclrcpts;      // # of local recipients for this mail data
    int         noofremrcpts;      // # of remote recipients for this mail
    char        *myhostname;       // My domain name
    LPSTR       myIPAddr;          // My IP address
    FWDREVPATH rempaths[MAXRCPTS];    //Array of structs containing fwd path and
                                   //rev path for remote recipients

    //functions

    void    init();                     // Does all the initializations
    int     helocmd( );            // processes USER command
    int     ehlocmd( );            // processes USER command
    int     mailcmd( );            // processes PASS cmd
    int     rcptcmd( );
    int     datacmd( );
    int     rsetcmd( );
    int     noopcmd( );
    int     quitcmd( );

    public:

    SMTPconn( );
    SMTPconn( int sock );
    ~SMTPconn() { };
    int     connEstbGreeting(void);    // Send a greeting message to client
    int     getcmd( );             // recv cmd from socket and parse it

};


//*****************************************************************************
//  The OutQueue class manages the outmsgQ object. Whenever a new message is
//  put in the Q it saves it in a disk file and then adds it to the outmsgQ.
//*****************************************************************************

class OutQueue
{
    private:

    short   noofmsgsinQ;
    TQueueAsDoubleList < OutGoingMsg > outmsgQ; //Queue of outgoing messages

    //Functions
    int     loadQ(const char *filename); //Load outmsgQ messages from file
```

```
    int     saveQ(void) const;           //Save outmsgQ data into file

    public:

    //Constructor
    OutQueue ( void );             //default constr loads outmsgQ from disk
    //Destructor
    ~OutQueue ( void );

    int put( const char *, const char *, const MailData & ); //creates
                                    //an OutGoingMsg and puts it in
                                    //the outmsgQ
    int get( char *, char *, MailData &); //Gets OutGoingMsg from Q and
                                    //returns the fwd & rev paths & data
    int isEmpty( );                 //Returns True/False

};
```

## ACRONYMS

| | |
|---|---|
| DLL | Dynamic Link Library |
| FTP | File Transfer Protocol |
| GIF | Graphic Interchange Format |
| ICMP | Internet Control Message Protocol |
| IGMP | Internet Group Management Protocol |
| IP | Internet Protocol |
| ISO | International Standards Organization |
| JBIG | Joint Bi-Level Image Processing Group |
| LZ77 | Lempel Ziv's algorithm based on 1977 paper |
| LZW | Lempel Ziv algorithm with modifications by Terry Welch |
| MIME | Multipurpose Internet Mail Extensions |
| MTA | Message Transfer Agent |
| OSI | Open Systems Interconnection |
| POP | Post Office Protocol |
| POP3 | Post Office Protocol Ver. 3 |
| RFC | Request For Comments |
| SMTP | Simple Mail Transfer Protocol |

| | |
|---|---|
| TCP | Transmission Control Protocol |
| UA | User Agent |
| UDP | User Datagram Protocol |

## REFERENCES


[1] Jamsa, Kris and Ken Cope, "Internet Programming", Las Vegas: Jamsa Press, 1995.

[2] Borenstein, N., and N. Freed, "Multipurpose Internet Mail Extensions", RFC 1521, Bellcore, Innosoft, September 1993.

[3] Stallings, William, "Data and Computer Communications", 4th ed., Macmillan Publishing Company, New York, NY, 1994.

[4] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, USC/Information Sciences Institute, August 1982.

[5] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, UDEL, August 1982.

[6] Postel, J., and J. K. Reynolds, "Telnet Protocol Specification", RFC 854, May 1983.

[7] Myers, J., Rose, M., "Post Office Protocol - Version 3", RFC 1725, November 1994.

[8] Sayood, Khalid, "Introduction to Data Compression", San Francisco, Morgan Kaufmann Publishers, 1996.

[9] Ziv, J and A. Lempel, "A Universal Algorithm for Data Compression", *IEEE Transactions on Information Theory*, IT-23(3):337-343, May 1977.

[10] Ziv, J and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding", *IEEE Transactions on Information Theory*, IT-24(5):530-536, September 1978.

[11] Cheung, Ada Ying Dee, "Data Transfer Using Controlled Compression", Masters Thesis, Waterloo, Ontario, Canada, 1996.

[12] Witten, I. H., A. Moffat, and T. Bell, "Managing Gigabits: Compressing and Indexing Documents and Images", Van Nostrand Reinhold, 1994.

[13] Welch, T. A., "A Technique for High-Performance Data Compression", *IEEE Computer*, 8-19, June 1984.

[14] Klensin, J., N. Freed, M. Rose, E. Stefferud, and D. Crocker, "SMTP Service Extensions", RFC 1869, MCI, Innosoft, Dover Beach Consulting, Inc.,

Network Management Associates, Inc., Silicon Graphics, Inc., November 1995.

[15] Yu, Tong Lai, "Data Compression for PC Software Distribution", Software Practice and Experience, Vol. 26(11), 1181-1195 (November 1996).

[16] Booch, Grady, "Object Oriented Analysis and Design With Applications", 2nd Ed., The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[17] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, "Object Oriented Modeling and Design", Englewood Cliffs, New Jersey: Prentice Hall 1991.

[18] Stevens, W. Richard, "TCP/IP Illustrated, Volume 1", Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[19] Wright, Gary R. and W. Richard Stevens, "TCP/IP Illustrated, Volume 2", Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[20] Richter, Jeffrey, "Advanced Windows, The Developers Guide to the WIN32 API for Windows NT 3.5 and Windows 95", Microsoft Press, Redmond, Washington, 1995.