A Reliable Low Bandwidth Email-based Communication Protocol

by

Janelle Kenty-Joan Prevost

S.B., Mathematics
Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2001

© Janelle Kenty-Joan Prevost, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ...........................................

Department of Electrical Engineering and Computer Science

May 29, 2001

Certified by ...........................................

Lynn Andrea Stein
Associate Professor
Thesis Supervisor

Accepted by ...........................................

Arthur C. Smith
Chairman, Department Committee on Graduate Students
A Reliable Low Bandwidth Email-based Communication Protocol

by

Janelle Kenty-Joan Prevost

Submitted to the Department of Electrical Engineering and Computer Science on May 29, 2001, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

In many places in the world, network infrastructure is characterized by intermittent connectivity, low-bandwidth and errors. As a result, web-browsing is slow. Because users frequently pay for each minute that they are connected, browsing the web can be an expensive proposition. In these places, what is needed is a communication protocol that can provide reliable end-to-end data delivery even in the face of episodic connectivity and low-bandwidth routes.

This thesis details the design and prototype implementation of a reliable, low-bandwidth, email-based communication protocol. It was designed for communication between principals in the TEK Search System, whose main goal is to improve access to the information available on the World Wide Web in low-connectivity areas. We provide a low-bandwidth message format and an error-recovery scheme, and address the issue of compression.

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor
Acknowledgments

- I thank my supervisors, Dr. Libby Levison, Prof. Saman Amarasinghe, and Prof. Lynn Stein for their guidance and contributions to this thesis. Libby was especially encouraging and always had time to discuss issues as they arose, even when she was many miles away in Peru. I also thank Dr. David Clark for the conversations he had with Libby and me about TEK and this thesis.

- My family has been extremely supportive of me during my 5 years at MIT. Their love, prayers, and words of encouragement were invaluable. I thank my mother, Joan Prevost, for proof-reading my thesis and for her excellent suggestions. I thank both parents for reminding me that it is ok to rest when one is tired and eat when one is hungry.

- I thank the Government of the Commonwealth of Dominica for its financial support for the past 5 years.

- Finally, I thank my friends (Kenroy, Kavita, Olu et al), especially those at MIT who were also going through the thesis-writing process at the same time as I was, for the countless laughs, frustrations, and all-nighters that we shared.

Janelle K. Prevost
Contents

1 Introduction .................................................. 13
  1.1 Motivation .............................................. 13
  1.2 Why Email? ............................................. 14
  1.3 Related Work ........................................... 15
  1.4 Outline of thesis ..................................... 16

2 Background: The TEK System ................................. 17
  2.1 Architectural Overview ................................. 17
  2.2 Related Work ........................................... 20
  2.3 Summary ................................................. 21

3 Protocol Design ............................................... 23
  3.1 Design Criteria ......................................... 23
  3.2 Design Details ......................................... 24
    3.2.1 Architectural Overview ............................. 25
    3.2.2 Message Format .................................... 26
    3.2.3 Error Detection and Recovery ..................... 27
    3.2.4 Message Compression ............................... 29
  3.3 TEK-specific Design Details ........................... 29

4 Implementation ................................................ 33
  4.1 Module Overview ....................................... 34
    4.1.1 Client ............................................... 34
4.1.2 Server ........................................ 37

4.2 Implementation Details ....................................... 40
4.2.1 Message Object ........................................ 40
4.2.2 Message Types .......................................... 41
4.2.3 Data Storage .............................................. 46
4.2.4 Classes .................................................. 49

4.3 Message Exchange Examples .................................. 51
4.3.1 Search ................................................... 51
4.3.2 Retransmission of a message .............................. 54

5 Testing .................................................................. 57

6 Conclusion .................................................................. 59
6.1 Future Work ...................................................... 59
6.1.1 Fragmentation and Reassembly .............................. 59
6.1.2 Message *piggybacking* .................................... 60
6.1.3 Adaptive Timeouts ........................................ 60
6.1.4 Client-side Database ....................................... 61
6.1.5 TEK-specific extensions .................................. 61
6.2 Possible Evaluation Techniques ............................... 62
6.3 Conclusion ....................................................... 63

A Database Tables ................................................... 65
List of Figures

2-1 TEK Client-Server Interaction .................................................. 18
2-2 TEK Server Modules ................................................................. 20
3-1 High-level protocol architecture ............................................... 26
4-1 Client-side Control Flow .......................................................... 35
4-2 Server-side Control Flow .......................................................... 39
List of Tables

3.1 General Message Format .............................. 27

4.1 Registration Message Body ......................... 42
4.2 Registration Reply Message Body .................. 42
4.3 Search Query Message Body .......................... 43
4.4 Search Query Reply Message Body .................. 43
4.5 URL Direct Fetch Message Body .................... 43
4.6 URL Reply Message Body ......................... 44
4.7 Hardware Update Message Body ..................... 44
4.8 Hardware Update Reply Message Body .............. 45
4.9 Software Upgrade Message Body ................... 45
4.10 Software Upgrade Reply Message Body .......... 45
4.11 Ping Reply Message Body .......................... 46
Chapter 1

Introduction

1.1 Motivation

The problem of how to exchange digital information across a stable network has been solved in many different ways. For real-time data transmission over the Internet, the de facto standard is the TCP/IP protocol suite. This protocol works well on the high-rate fibre networks of first-world countries where continuous connectivity is available typically with negligible error, loss or delays.

In contrast, in many developing countries, the network environment is quite different: low bandwidth, low connectivity, and data corruption are characteristic. Those few people who have access to the Internet do so by dialing up to a local Internet Service Provider (ISP) through a modem. In these countries, the telephone network has a sparse reach, and teledensity is extremely low. As such, accessing the Web and downloading information is slow. Emails sometimes get lost or take longer than normal to reach their destination. What is more, the cost of basic Internet access over this unreliable telecommunications infrastructure is expensive.

ISPs in many countries of the world distinguish between an account with Internet (WWW) access, and an account with only email access; Internet access accounts are typically much more expensive than email accounts. As such, a large majority of online users in these countries have access only to email. The store and forward nature of email makes it well suited for use in situations where connectivity is not
always continuous, since it affords the opportunity to take advantage of spurts when the network is expected to perform well\(^1\).

This thesis addresses the problem of how to deliver data reliably over unreliable, low bandwidth networks. The major challenge is to design a robust communication protocol that can recover from message loss and errors, and which minimizes both the number of messages that need to be exchanged in order to accomplish a task and the amount of data transmitted per message.

We will present a design and prototype implementation of a reliable, low bandwidth email-based communication protocol for the Time Equals Knowledge (TEK) system [1]. TEK is a search engine aimed at improving WWW access in developing countries where low bandwidth, unstable connectivity and high telecommunications charges make the use of current Internet search utilities unviable. The implementation of this email-based protocol is central to making TEK a reality.

1.2 Why Email?

Several reasons motivated the choice of email as the transport mechanism for our protocol. First, connectivity constraints made it clear that the protocol could not require real-time responses. In other words, the protocol had to be asynchronous. The communication environment of most developing countries is characterized by intermittent connectivity. This means establishing and maintaining a connection to a remote end-point is difficult. In order to work well, the protocol had to be designed in such a way that it could take advantage of time spurts when the network is expected to work well. The store and forward nature of email made it an obvious candidate. With email for transport, the protocol can package information for delivery, wait until the network is ready and then forward the message in an email to its destination.

Second, since Internet access and the cost of telephone calls in developing countries is expensive relative to per capita income, the length of time spent connected to the

\(^{1}\text{For example, this might be late at night, when there might be fewer users engaging lines on the local telephone network.}\)
ISP is a significant part of the cost of access. Checking email requires short on-line
time: the time to log on, download new mail, send composed messages, and log
off. Using a connection-oriented, synchronous communication method would require
a longer connection time, and would therefore be more expensive. Email does not
require maintaining a persistent connection, as such it fits as the best solution.

Third, recall that in many places in the world more people have access to email
than to the World Wide Web. This made email an even more attractive option. Since
it would be infeasible to try to change the underlying communication means between
machines with Internet access, the protocol leverages the existing store and forward
nature of email to produce a solution to the problem of low connectivity. In this way,
the sender can package messages in advance and send them via email to the recipient
at some later time.

1.3 Related Work

The Interplanetary Internet Project

A current research project with motivations and issues very similar to those addressed
by this thesis is NASA’s Interplanetary Internet Project [5]. The project aims to de-
fine the architecture and protocols needed to allow the Earth’s Internet to be able to
communicate with Internets located on other planets. The communication environ-
ment that underlies the Interplanetary Internet is faced with:

- long speed-of-light delays (from Earth to other planets);

- intermittent connectivity (communicating entities may lose their line of sight
due to interference from other planetary bodies);

- asymmetric data rates (inbound and outbound data may travel at different data
  rates);

- low signal to noise ratios (higher ratios imply a reduced likelihood of error when
  the receiver is decoding information from the sender)
• and high cost (using the Deep Space network is very expensive).

Durst, et al [6] cite the email model, with its store and forward nature, as “perfect for operating in environments in which there is not continuous connectivity between all hosts”, though they indicate that SMTP as a protocol is not.

1.4 Outline of thesis

In the following chapters, we discuss the design and implementation of an email-based communication protocol which addresses the issues of low bandwidth and low connectivity. We begin in Chapter 2 with a brief background on the TEK System, which motivated the development of this protocol. Next, in Chapter 3 we look at the protocol design. This will include the definition of the core functionality that the protocol should provide and design details. In Chapter 4 we discuss the implementation of the protocol for use by the TEK Search System. This is followed in Chapter 5 by a brief look at the testing done on the implementation. Finally, in Chapter 6 we look at future extensions to the protocol and give conclusions.
Chapter 2

Background: The TEK System

The TEK System is a low-connectivity search engine designed to accept search queries via email from a user at a remote location. It also returns the search results by email. While most existing search engines aim for instantaneous delivery of search results, TEK focuses on:

1. trying to extract a well-defined query from the user;

2. post-processing of search results, in order to return more relevant information, in a more compact form.

These are both extremely important because they translate to the difference between relevant and irrelevant search results. Because the TEK service is targeted to be deployed in the developing world, where the telecommunications infrastructure is poor and Internet access is expensive, every irrelevant search result returned to the user is a waste of precious bandwidth and money.

2.1 Architectural Overview

The TEK System follows a client-server architectural model. This is illustrated in Figure 2-1.
Figure 2-1: TEK Client-Server Interaction
TEK Client

The TEK client makes search requests to the TEK server via a local web proxy server. It connects via a web browser to the proxy server, which communicates (using our email-based protocol) with the TEK server on the client’s behalf when necessary. The client user interface is a set of HTML pages. This HTML-based user interface allows searching of the information stored in the local proxy server’s cache, and viewing of search results obtained from the local proxy server’s cache. Also, it allows for the creation of search queries for transmission to the TEK server, and presentation of query results returned from the TEK server. In addition, access to the other resources of the system (e.g. thesaurus, dictionary) is through the HTML user interface.

TEK Server

On receiving a query, the TEK server funnels the search terms to other search engines (e.g. Google [2]), selects the best\(^1\) results from what was returned, clusters related files, and removes extraneous information that would waste bandwidth. It then generates summaries of the resulting information, and compresses and sends the results back to the TEK client side using the protocol. The TEK server is comprised of 6 modules, which make use of a central database for storing intermediate results at different stages of the search.

1. **Controller module.** This is a central controller responsible for passing the search query along to the different processing modules of the Server.

2. **Query Generator module.** This receives the search query and records it in a database on the server side.

3. **Query Engine module.** This module performs the search by forwarding the query terms off to some Internet search engine\(^2\), and retrieving the URLs returned.

---

\(^1\)Natural language algorithms are used to select a subset of files from the preliminary collection of results.

\(^2\)For example, the TEK servers may forward a request for a news article to http://www.cnn.com.
Figure 2-2: TEK Server Modules

4. **Information Select module.** Here, HTML tags are removed from the result files and a clustering algorithm is run to group similar results files together.

5. **Information Refine module.** This module removes any unnecessary text (e.g. JavaScript code) and images (unless the search query explicitly instructed otherwise). It also generates summaries of the results.

6. **Transfer module.** This module picks a selection of URLs from each cluster up to the size limit for the client and packages them for return to the client.

### 2.2 Related Work

**GetWeb and www4mail**

Several web-to-email servers, such as GetWeb and www4mail currently exist. Both of these allow a user with basic email functionality to retrieve web pages.

GetWeb [3] permits retrieval of specific URLs and also provides a web search
utility. To retrieve a specific URL, GetWeb requires that the user send an email to a GetWeb server with the subject line blank, and the body of the email containing only "GET" followed by the exact URL to be returned. The GetWeb server then retrieves the page from the web and returns it in an email as text.

To perform a search, the user must send an email to a GetWeb server, with the subject line being blank, and the body containing the search instruction:

```
SEARCH Search_Engine_Name Search_Terms
```

where `Search_Engine_Name` might be ALTAVISTA, INFOSEEK, or YAHOO, and `Search_Terms`, the search terms to be processed. The server returns a list of URLs found by the search engine. The user must then compose another GetWeb email to request the URLs he wants to view from among those in the list.

www4mail [4] is similar to GetWeb, in that it requires that the user construct an email to the www4mail server in a specified format. To perform a web search, one has to know the exact format that the search engine expects.

The main advantage of the TEK system over these web-to-email servers is that it provides a search tool via a user interface that hides all formatting details from the user. This makes it easier to use, and in some sense, almost parallel with standard search engines. In addition, TEK uses intelligent filtering techniques to return the most relevant results, and removes high bandwidth content such as images. Furthermore, TEK performs a local search of cached pages before sending search requests out in an email to the TEK server, thus making efficient use of previously returned results.

## 2.3 Summary

As explained in this chapter, the goal of the TEK system is to provide a search tool that overcomes the connectivity problems being faced in many developing countries. To recapitulate, the following assumptions were made as the system was designed:

1. **Connectivity is unreliable.** This major assumption is that the client is unable to maintain a persistent connection to the server. This had a huge influence on
several key design decisions, in particular, the choice of email for communication between TEK clients and server.

2. *Bandwidth on the client side might be extremely constrained.* This implies that downloading may be slow.

3. *Transactions between the client and server should be accomplished with the minimal number of round-trip communications.*

Under these assumptions, it was clear that for reliable communication between a TEK client and the TEK server a robust, low bandwidth protocol that did not require real-time communication, and which could take advantage of times when the network on the client side would be most reliable was needed. The store and forward nature of email makes it ideal as a transport mechanism for such a protocol. In the subsequent chapter, we discuss in detail the design of the TEK communication protocol.
Chapter 3

Protocol Design

In this chapter we present the detailed design for the low bandwidth protocol motivated by the requirements of the TEK system. As much as possible, the design will be presented as application-independent. However, in the final section of this chapter, we will discuss some aspects of the design which are TEK-specific.

3.1 Design Criteria

The following design criteria were used to drive the design process:

1. *Simplicity.* The design should be low in complexity so as to minimize the chance of implementation errors. A simple protocol is easier to implement, debug, maintain and extend.

2. *Extensibility.* The design should be easy to extend. A modular design approach was chosen. Modular designs are easy to extend provided that the modules and the interfaces between them are clearly defined.

3. *Fault Tolerance and Reliability.* The protocol must be reliable in the face of errors. It must adapt gracefully to any of the failure modes of email.

4. *Ease of Integration.* The design should ensure that integrating the protocol into an application is simple and straightforward.
5. **Functionality.** To fully address the issues of low bandwidth and intermittent connectivity, the protocol must provide the following core data manipulation and transfer control functions [7]:

(a) *Message Formatting.* Before any information is transferred from the sender, it should conform to a fixed message format so that it can be read at its destination.

(b) *Error recovery.* The protocol should be able to recover from certain errors, including message loss and message duplication. For the TEK System in particular, the protocol should be able to recover from interrupted downloads. This is important because of how expensive Internet access is in the regions that TEK will be deployed. It is imperative that if a download operation is interrupted before the complete file has been obtained, the protocol restart the download from the point of interruption, and not repeat the download.

(c) *Message Compression.* To maximize the amount of information transmitted per email, large messages should be compressed before they are transmitted.

(d) *Message fragmentation and reassembly.* To reduce bandwidth constraints, the protocol should fragment large messages before they are sent, and reassemble the fragments when they have arrived at their destination.

(e) *Encryption.* To ensure privacy, the protocol should encrypt all messages before they are sent.

### 3.2 Design Details

In this section we give details of the overall design of the protocol. The protocol follows a simple request-reply paradigm. Each request originating from a sender is matched with a reply from the recipient. Replies are not explicitly acknowledged, however. This is primarily in order to minimize the number of messages that need to
be exchanged in order to complete a task. Secondly, if the protocol were to explicitly acknowledge replies, then it would have also to deal with the possibility of lost acknowledgments; this would add significant overhead.

Another important feature of the protocol is that it is asynchronous. This means that the sender does not block while it awaits a reply from its peer. This choice was preferred over an asynchronous design, due to the assumption that connectivity is episodic and unreliable. In the context of TEK, the server may take several hours before returning a set of search results in response to a search query, thus an asynchronous design is better suited for the protocol.

The protocol uses timeouts and retransmission for error recovery. Simply put, request messages are retransmitted if the timeout parameter associated with the request is triggered.

3.2.1 Architectural Overview

In general, we can think of the protocol as a series of layers below the application. More specifically, the protocol controls and abstracts the communication of the application with the outside world. As Figure 3-1 illustrates, the protocol is comprised of two layers: the Transport layer, and the Protocol Management layer.

The Transport Layer

This is the outermost layer of the protocol. It provides reliable, end-to-end data transport. The Transport layer formats content for delivery, and emails it to its destination. It also handles picking up of incoming messages. On receiving a message in an email, the Transport layer parses it, and dispatches it to the correct function in the Protocol Management layer. The Transport layer is also responsible for fragmenting large messages into smaller, fully addressed fragments, so that messages transmitted between endpoints are of an acceptable size for the network. Timers that trigger retransmission of messages, and other processes that aid with error detection and recovery, are also found in this layer of the protocol.
The Protocol Management Layer

Parsed messages from the Transport layer are passed to functions in the Protocol Management layer. Among others, functions responsible for checking whether clients and servers are up and running are found in this layer of the protocol. The Protocol Management layer is also responsible for passing application-specific messages to the appropriate application function. For example, when a TEK query arrives at the TEK server, the Protocol Manager passes the message parameters to the TEK search handler.

3.2.2 Message Format

In keeping with the design criteria that the protocol be simple and easy to extend, we designed a simple message structure for all messages to be transmitted by the protocol. Each message consists of two components (Figure 3.1): a Header, which contains information used to identify the message, and a Body, which, as its name implies, contains the bulk of the message content, which is often application-specific information.
Table 3.1: General Message Format

**Message Identifiers**

The protocol is designed to stamp each message with a unique Message Identifier (MID). This is to ensure that the messages can be tracked. Unique identifiers must be generated for each request and reply message sent using the protocol. The body of a reply message must also contain the MID of the request message to which it is a reply.

### 3.2.3 Error Detection and Recovery

The protocol provides a detection and recovery mechanism for messages that do not arrive at their destination (lost messages), duplicate messages, certain corrupt messages, and reordered messages.

#### Lost Messages

Messages may fail to arrive at their destination for several reasons:

1. **Network transmission failure.** The email (carrying the message) is dropped by the network.

2. **Send-omission failure** [8, p.56]. The message is deleted from the outgoing message buffer after it is packaged by the sender, but before it is emailed to the
receiver. This might be due to a system failure on the sending machine.

3. Receive-omission failure [8, p.56]. The message is deleted from the incoming message buffer on the receiving side before it can be parsed and dispatched to the appropriate process.

Each sender has a standard, configurable, timeout parameter that indicates how long it should wait for a response to a request. If a sender sends a message and does not receive a response within the timeout period, then it assumes that the original message is lost, and it reconstructs and retransmits the message, flagging it as a duplicate message, but assigning it the same MID as the original. This requires that each sender keep track of all messages that are unresolved i.e. those messages for which a response has not yet been received.

**Duplicate Messages**

Duplicate messages can be received for several reasons:

1. Network transmission error. The mail transport protocol causes duplicate messages to be sent in error.

2. The sender resends the message in error.

3. The sender resends the message because it never received a response to the original message.

The receiving entity will get a duplicate request either while it is still processing the original request or after it has already sent the reply (meaning the original reply was lost). If the receiver is still processing the original request when the duplicate arrives, the protocol will detect this (by recognizing that the new request has the same MID as the original), and will discard the duplicate. If however, the duplicate request message is due to a lost reply, the receiver simply re-executes the operation to regenerate a reply to the message.
Corrupt Messages

For the protocol, corrupt messages are defined as those emails that cannot be parsed. One case of this is bounced emails. The protocol will handle unrecognizable emails (i.e. emails that are rejected by the Protocol Management parser module) by automatically forwarding them to the application administrator, or by placing them in a special folder where they can easily be dealt with by the administrator.

Reordered Messages

The protocol is designed so that it does not require that messages are received in the order that they were sent. This is because each request is associated with exactly one reply. Furthermore, each request-reply message pair (unless it is fragmented) is independent of other request-reply pairs. Reordering of messages may be a problem when messages are fragmented and sent over several emails, instead of just one large one. This is discussed further in Chapter 6.

3.2.4 Message Compression

The protocol is designed to compress all messages before transmitting them over the network, and then decompress the messages when they have been received at the destination site. A disadvantage often cited when dealing with data compression is that it can be slow, since the compression algorithms are often time-consuming. This is not a concern for the TEK project since we are operating under the assumption that it is acceptable to make the tradeoff of speed for information.

3.3 TEK-specific Design Details

The protocol layers have little knowledge of what happens in the application. To the application, these layers abstract away all communication details. On the TEK server side, the protocol interfaces with the TEK search functionality. The TEK server forwards search results to the Transport layer of the protocol for packaging and
delivery. On the client side, the protocol is responsible for delivering search queries, and receiving search results. However, TEK needs to be capable of exchanging other information, i.e. not just search information, between TEK entities. We now look at the different classes of information that the protocol must be able to transport for the TEK System.

Information to be transported

1. **Search Information.** The main function of the TEK system is to provide search capabilities to its users. When a user types a search query into the TEK client user interface, the search terms, along with any other information specified by the user (e.g. the language to perform the search in) must be collected and sent to the TEK server for processing. The server must respond with the results of the search.

2. **URL Request Information.** Another function of the TEK System is the direct URL request function. The user types in a specific URL into the TEK client user interface, the client sends the URL address to the server, and the server must reply with the HTML file addressed by that URL. This is the basic functionality provided by web-to-email servers [3, 4].

3. **Client Registration Information.** All new clients must register with a TEK server. The client must send registration information to the TEK server; the server replies with a unique client identifier for that new client. This client identifier is used in all subsequent communication messages exchanged between that client and the server.

4. **Hardware Update Notification Information.** The TEK server uses information about the hardware specifications of each client to determine, among other things, the maximum file size that the client can download without high risk of failure. The client provides its hardware profile at registration. If any hardware information supplied at registration is later upgraded, then the TEK
server needs to be updated with this information. The server should respond with information to indicate whether or not the update was successful.

5. **Software Upgrade Information.** When small upgrades are made to the TEK Client implementation, it may be possible to send these changes from the TEK server to the client through the protocol, instead of having the client wait for a new version to be sent (on a CD-ROM via snail mail, for example). The client should respond with information to indicate whether or not the upgrade was successful.

6. **Data Upgrade.** An electronic dictionary, thesaurus and atlas are included with every TEK client. Whenever additions need to be made to this reference library, the new data could be sent to the client using the protocol. The client should reply with information to indicate that it received the data upgrade.

7. **Test of System Health.** Information used to ensure that the system is up and running, and to facilitate error-recovery may also need to be exchanged. For example, system health testing can be initiated from the TEK server to check whether a particular client is up, or from a TEK client, to check whether the TEK server is up.

8. **Database Synchronization.** The TEK system builds its own local database for the client as search results are returned. There must be a mechanism to update this database of information as time passes (since files may become outdated), and in the case that local files get deleted or corrupted.

In the following chapter, we discuss the implementation of the protocol specifically for use in the TEK System.
Chapter 4

Implementation

In this chapter we describe the prototype implementation of the protocol for the TEK system. The implementation was done in Java to ensure ease of integration with the TEK system, which itself is implemented in Java, and because of Java’s easy portability to both Windows and Unix platforms. For the prototype, implementation of the core features of the protocol were undertaken, namely, end-to-end data transport and reliability.

We begin in Section 4.1 with a descriptive overview of the implementation modules of the protocol on the TEK client and the TEK server. The discussion in this section is based on the control flow diagrams in Figures 4-1 and 4-2. From these figures, it is evident that the structure of the implementation on the server mirrors that of the client. However, there is one key difference between the client-side and server-side implementations, which we will discuss. In Section 4.2 we delve further into the implementation details. We define the specific message types supported by this implementation of the protocol, and their message body format. We will also look at the storage media used on the client and server, and the classes for the control flow modules. Finally, to demonstrate the use of the protocol, we run through some message exchange examples in Section 4.3.
4.1 Module Overview

In this section we discuss the control flow for the protocol, as illustrated in Figures 4-1 and 4-2. We will describe in detail each module involved in sending and receiving messages.

4.1.1 Client

The implementation modules of the protocol on the client side are illustrated in Figure 4-1. In our discussion, we follow the control flow from the time a TEK email arrives at the client.

At the TEK client, TEK emails are filtered out and placed in a special directory as they arrive. This directory is the incoming mail folder. The incoming mail folder is polled periodically for new mail by the IncomingMailDaemon module. This polling frequency is configurable. The IncomingMailDaemon reads mail messages from the folder in the order that they are placed there. This means that the oldest emails are read and removed first. The IncomingMailDaemon then passes all emails, in the order that it reads them, to the Parser module.

The Parser, as its name implies, is responsible for parsing TEK emails to determine their message type, since this indicates how the mail must be handled. If, for some reason, the email cannot be parsed successfully, the Parser assumes that it is an error message and stores it in the error mail folder for intervention by the administrator. When a message has been successfully parsed, the lastheardfrom file is updated with the date and time that the message was received. The lastheardfrom file stores the date and time that the last message from the server was received. The data from successfully parsed emails is housed in a Message object. The Parser module is responsible for dispatching the Message object to the correct handler function in the Protocol Manager.

The Protocol Manager has two major functions: handling incoming requests from the TEK server, and handling replies to messages the client had sent to the server. For application-specific messages, e.g. search results, the Protocol Manager
Figure 4-1: Client-side Control Flow
passes control to the TEK Client Application module. In handling replies from the
TEK server, the Protocol Manager removes associated request messages from the
pendingreplylist\(^1\) file, which serves to store all requests that have been sent to the
server, but have not yet received a reply from the server. This list of requests awaiting
a reply from the TEK server is kept in order to facilitate message retransmission where
necessary. To better understand what other functions the Protocol Manager must
provide, we briefly review the types of information that a TEK client must be able
to handle.

1. **Search Results.** The TEK client must be able to receive search results that
get sent back from the TEK server in response to a search query.

2. **URL Direct Fetch results.** The TEK client must be able to receive results
that arrive in response to a request for a specific URL.

3. **Registration Response.** The TEK client must be able to receive the registra-
tion response returned by the server with its unique CID.

4. **Hardware Update Response.** The server sends a Hardware Update response
to the client when it has received and processed a Hardware Update notification
message from the client.

5. **Software Upgrade.** The client must be able to receive Software Upgrades
from the server.

6. **Data Upgrade.** The client must be able to handle Data Upgrade messages
from the server.

7. **Pings and Ping Replies.** The client must be able to respond to pings, and
handle ping replies sent by the TEK server.

8. **Database synchronization response.** The client must be able to receive
database synchronization responses from the server.

\(^1\)When the client constructs a request to send to the server, the request is added to the
pendingreplylist file.
When the TEK client needs to send messages to the TEK server, these messages are constructed by the EmailBuilder. The EmailBuilder ensures that the message headers are included, and in the correct format, and that the message body follows these headers. The format for the messages supported by this implementation is discussed in Section 4.2. After the message has been created the EmailBuilder:

1. places the message, coupled with its destination, into the outgoingmailqueue file where it is picked up and emailed out by another protocol module;

2. in the case of a request message, appends the message to the list of messages on the pendingreplylist that are awaiting a reply from the server. This list is maintained for error recovery purposes, so that a duplicate request can be sent if the server is taking too long to send a response.

The EmailBuilder is also responsible for constructing duplicate messages, and ensuring that the body of a duplicate message is flagged as such.

The TEK client has a daemon which facilitates the retransmission of request messages that have been awaiting replies for an extended period of time. The Retransmit daemon can be configured to run at certain intervals. It polls the list of messages awaiting replies, i.e. the pendingreplylist file, searching for any message that has been on the list for a long time (e.g. more than one week, or more than one month). If it finds any messages that fall into this category, the Retransmit daemon hands it to the EmailBuilder, which flags the message as a duplicate, and puts it back in the outgoingmailqueue file where it can be picked up for delivery to its destination.

The OutgoingMailDaemon on the client polls the outgoingmailqueue file for new messages and emails these messages to their destination.

4.1.2 Server

Our discussion of the implementation on the server side will be analogous to the discussion of the client implementation. The server-side implementation is indeed quite similar to the client-side implementation, as such, many of the details explained
in the previous section will be left out of this discussion. The implementation modules of the protocol on the TEK server are illustrated in Figure 4-2.

At the TEK server, TEK emails are filtered out and stored in the incoming mail folder. The server’s IncomingMailDaemon module reads mail from this directory and passes it to the Parser module. Any messages that cannot be successfully parsed by the Parser are placed in the error mail folder. The Parser stores the data from successfully parsed messages in a Message object and dispatches it to the correct handler function in the Protocol Manager. The following messages are transmitted to the TEK server by the TEK clients:

1. **Search Queries.** The TEK server must be able to receive search queries from TEK clients.

2. **URL Direct Fetch Requests.** The TEK server must be able to receive requests from TEK clients for specific URLs.

3. **Registration Requests.** The TEK server must be able to receive registration requests from clients.

4. **Hardware Updates.** The client sends a Hardware Update to the TEK server when some aspect of its hardware configuration has changed.

5. **Software Upgrade Responses.** The server must be able to handle Software Upgrade responses from the client.

6. **Data Upgrade Responses.** The server must be able to handle Data Upgrade messages from the server.

7. **Pings and Ping Replies.** The server must be able to handle pings, as well as ping replies sent by TEK clients.

8. **Database Synchronization Requests.** The server must be able to receive database synchronization requests from clients.
Figure 4-2: Server-side Control Flow
The Protocol Manager passes search requests to the TEK application, and handles all other messages itself. If a message received by the Protocol Manager is a reply message, then the corresponding request is moved from the awaiting.reply.list to the closed.msg.list. Both lists are stored on the database and their functionality is described in Section 4.2.3, and Appendix A.

Like the TEK client, the TEK server uses its EmailBuilder module to construct messages that must be sent to the client. The EmailBuilder adds messages to an outgoing mail list (mail.q on the database). Request messages are placed on the awaiting.reply.list on the database. The server also has a Retransmit daemon that resends requests if they remain in the awaiting.reply.list for too long. The server’s OutgoingMailDaemon sends messages from the mail.q to their destination.

### 4.2 Implementation Details

In this section we introduce the Message object, and define the message types supported by the protocol. We will also look at how the protocol stores information on the TEK client and TEK server.

#### 4.2.1 Message Object

This object stores the parsed data from a TEK email. It is an immutable object that provides accessor methods to all its stored data, but does not allow editing of the data that it holds. The member variables of the Message object are:

```java
public class Message {
    ...
    // the unique string identifier for the message
    private String m_MessageID;
    // the unique string identifier of the client or server sending the message
    private String m_EntityID;
    // this indicates what type of information the message is housing
    private int m_Type;
    // the email address from which the message originated
    private String m_Source;
    // the fragment number of the message
```
private int m_FragNum;
// the total number of fragments the original message had been divided into
private int m_TotalFrags;
// a boolean flag to indicate whether this message is a duplicate
// of a previous message
private boolean m_Duplicate;
// this stores the message body contents
private Hashtable m_Body;
...

We use a Java Hashtable object to store the body of the message. This facilitates easy indexing of the message body parts, which are defined for each message type in the following section.

For messages created by a client, MIDs are of the form: CID.Timestamp. CID is the unique client identifier assigned to a client. It identifies the message as originating from a particular client. Timestamp is a GMT time-stamp obtained from the system clock. It is the number of milliseconds since midnight GMT January 1, 1970. Using this time-stamp function provides an increasing sequence of nonce identifiers. Timestamp makes the message unique among all messages originating from that particular client.

For messages created by a server, MIDs are of the form: RAND.Timestamp. RAND is a random number between 0 and 1. Timestamp is defined as for the client.

4.2.2 Message Types

We now look in detail at the messages transported by the TEK-specific implementation of the protocol, and the structure of the message body for each message.

Type: REGISTRATION

The format for this message type is shown in Table 4.1. This message is generated by the client. It is sent to the server when a new client is installed on some machine. It contains all the information needed to register a new client with the server.
Table 4.1: Registration Message Body

<table>
<thead>
<tr>
<th><strong>Message Body</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
</tr>
<tr>
<td>ADM</td>
</tr>
<tr>
<td>PFM</td>
</tr>
<tr>
<td>OS</td>
</tr>
<tr>
<td>DSA</td>
</tr>
<tr>
<td>SLG</td>
</tr>
<tr>
<td>RLG</td>
</tr>
<tr>
<td>LOC</td>
</tr>
<tr>
<td>MAX</td>
</tr>
<tr>
<td>ORG</td>
</tr>
<tr>
<td>MS</td>
</tr>
</tbody>
</table>

Type: REGISTRATIONREPLY

After processing a REGISTRATION message, the server assigns a unique ID to the new client. The server generates a REGISTRATIONREPLY message to send this ID back to the client. This Client identifier (CID) must be used in all subsequent messages from that client to the server. Table 4.2 below presents the format for the body of this type of message.

Table 4.2: Registration Reply Message Body

<table>
<thead>
<tr>
<th><strong>Message Body</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CID</td>
</tr>
<tr>
<td>AMID</td>
</tr>
</tbody>
</table>

Type: SEARCHQUERY

When the user types a search query into the TEK Client User Interface, the protocol builds a SEARCHQUERY message for transport to the TEK server. As Table 4.3 shows, these messages contain the query ID assigned to the query by the client, and the query string typed in by the user. In future upgrades to TEK the user will be able to specify more parameters for SEARCHQUERY messages e.g. the return type of the
results.

<table>
<thead>
<tr>
<th>Message Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>QID</td>
</tr>
<tr>
<td>QS</td>
</tr>
</tbody>
</table>

Table 4.3: Search Query Message Body

**Type: SEARCHQUERYREPLY**

This message is generated by the TEK server after it has processed a search query and is ready to return results to the client that sent the query. The format of the body of this message is shown in Table 4.4.

<table>
<thead>
<tr>
<th>Message Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMID</td>
</tr>
<tr>
<td>QID</td>
</tr>
<tr>
<td>RES</td>
</tr>
</tbody>
</table>

Table 4.4: Search Query Reply Message Body

**Type: URLREQUEST**

The URLREQUEST message (Table 4.5) is generated by the client. It is a request from the client for the server to return a specific URL.

<table>
<thead>
<tr>
<th>Message Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
</tr>
<tr>
<td>PIX</td>
</tr>
<tr>
<td>LINKS</td>
</tr>
</tbody>
</table>

Table 4.5: URL Direct Fetch Message Body
**Type: URLREPLY**

This message is generated by the server in response to a URLREQUEST message from the client. As Table 4.6 shows, it contains the HTML page addressed by the URL that the client requested.

<table>
<thead>
<tr>
<th>Message Body</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMID</td>
<td>MID of the message to which this is a reply</td>
</tr>
<tr>
<td>URL</td>
<td>URL that had been requested by the client</td>
</tr>
<tr>
<td>RES</td>
<td>compressed html page requested</td>
</tr>
</tbody>
</table>

Table 4.6: URL Reply Message Body

**Type: HWUPDATE**

This message originates from the client. It is sent to notify the server of upgrades to the hardware on the client machine. Any of the hardware information that was sent to the server when the client registered could be changed using one of these messages. The format for the body of the HWUPDATE message is shown in Table 4.7.

<table>
<thead>
<tr>
<th>Message Body</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HID</td>
<td>hardware code of the item to be updated</td>
</tr>
<tr>
<td>DATA</td>
<td>new hardware specification</td>
</tr>
</tbody>
</table>

Table 4.7: Hardware Update Message Body

**Type: HWUPDATEREPLY**

The HWUPDATEREPLY message, shown in Table 4.8, originates from the server. It is sent in response to a HWUPDATE message.

**Type: SWUPGRADE**

This message originates from the server. It is sent to the client when minor upgrades are to be made to the TEK client software. For example, it could be used to update
Table 4.8: Hardware Update Reply Message Body

Table 4.9: Software Upgrade Message Body

Type: SWUPGRADEREPLY

This message (Table 4.10) is generated by the client in response to a SWUPGRADE reply message.

Table 4.10: Software Upgrade Reply Message Body

Type: PING

This message can originate from either a TEK client or server. The sender uses it to test whether the recipient is up and running. The PING message contains only headers.

Type: PINGREPLY

This message can originate from either a TEK client or TEK server. The sender generates it in response to a PING message. The body of this message simply contains the MID of the ping message to which it is a reply, as shown in Table 4.11.
Table 4.11: Ping Reply Message Body

<table>
<thead>
<tr>
<th>AMID</th>
<th>MID of the message to which it this a reply</th>
</tr>
</thead>
</table>

NB: The database synchronization messages described in Chapter 3 were not implemented for this prototype.

4.2.3 Data Storage

A key difference between the implementation of the protocol for the client and for the server is that the client makes use of the file system to store information for the protocol. The server, however, makes use of tables in the TEK server’s central database. In this section we describe the files used by the TEK client and the database tables used by the TEK server for protocol storage. This section is supplemented in Appendix A with further detail on the server database tables.

TEK Client: Files

1. outgoingmailqueue. This file stores a serialized Queue\(^2\) object that has the queue of messages to be emailed out. The EmailBuilder module on the client constructs email messages and stores them in the outgoingmailqueue file. This file is polled by the OutgoingMailDaemon module periodically for new messages to be emailed out.

2. pendingreplylist. This file is used to facilitate retransmission of request messages for which there has been no response from the server. When the EmailBuilder constructs a request message to be sent to the server, it adds the message to the pendingreplylist file. This file stores a serialized Java Hashtable of the messages awaiting reply from the server indexed by their MIDs. When the ProtocolManager module is handling a reply message from the server

\(^2\)The Queue class provides a first in first out data structure. It is based on Java’s Stack object.
it checks the pendingreplylist file for the associated request, and removes it from the file. The Retransmit daemon examines the MIDs of all the messages on the pendingreplylist to check whether any of the messages have expired. (Recall that part of the MID consists of the timestamp of when the message was created. This is easily extracted and read by the Retransmit daemon.)

3. lastheardfrom. This file is used to store the date that the last email from the server was received by the client. It is updated by the Parser module and stores a serialized Java Date object.

4. ClientID. This file is created when the registration reply message is received from the server. It stores the unique ID for the client.

**TEK Server: Database Tables**

The TEK server has an Oracle database which the different TEK server application modules use for tracking the state of a query as it is being processed. The server-side protocol implementation makes use of this database to store certain information. To summarize, the following information is stored in tables in the database for use by the protocol:

1. client_info_table. This table stores client registration information, including the date and time that the last message was received from each client. When a TEK client registration request arrives at the TEK server, a new row is added to client_info_table for this new client. One of several functions that provides access to this table is the get_client_email(String client_ID) function, which returns the email address of the client identified by <client_ID>. Other functions used to update and query this table are listed in Appendix A.

2. mail_q. This table stores the queue of messages to be emailed out to different TEK clients. It is analogous to the outgoingmailqueue file on a TEK client. The EmailBuilder module on the TEK server calls insert_in_mailq(Vector in_msg) to add a new message to the list of outgoing messages stored on the
mail_q table. The `in_msg` argument is either a 4-element or 3-element Vector, depending on whether the message has a file attachment or not. If there is no attachment, `in_msg` is composed as follows:

- `in_msg.elementAt(0)` = destination email address (String)
- `in_msg.elementAt(1)` = message headers and body (String)
- `in_msg.elementAt(2)` = `FALSE` (Boolean) to indicate there is no attachment

However, if there is an attachment, `in_msg` is composed as follows:

- `in_msg.elementAt(0)` = destination email address (String)
- `in_msg.elementAt(1)` = message headers and body (String)
- `in_msg.elementAt(2)` = `TRUE` (Boolean) to indicate there is an attachment
- `in_msg.elementAt(3)` = attachment file name (String)

SEARCHQUERYREPLY and URLREPLY messages sent to the TEK client include their results as a file attachment to the email carrying the message. When a message has a file attachment, the EmailBuilder module on the TEK server stores the attachment in the TEK server’s attachments folder, and places the file name under which the attachment was saved into the mail_q. The file name of an attachment to a message includes the MID associated with that message. This ensures that no files in the attachments folder are overwritten.

3. awaiting_reply_list. This table is analogous to the client’s pendingreplylist file. It stores the request messages awaiting a reply from a TEK client. Recall that the protocol follows a request-reply paradigm. Therefore, each request message originating from the TEK server is matched with a reply message from a TEK client. The fate of a request message on the awaiting_reply_list depends on whether a response from the client ever arrives. If the reply does arrive after a reasonable amount of time (including after a number of retransmissions), then the message is moved to the closed_msg_list. However, if the
message remains on the list for an extended period of time, it is transferred to the problem.msg.list.

4. closed.msg.list. The list of closed messages, i.e. request messages whose replies have been received from a client.

5. problem.msg.list. A list of problem messages. Messages on this list include request messages which never received a reply.

6. sent.mail.table. This table logs all messages emailed out by the TEK server; this includes both requests and responses to TEK clients.

4.2.4 Classes

Each module on the client and server was implemented in Java by a class of a similar name.

IncomingMailDaemon
This module is implemented by CIncomingMailDaemon.java on the client, and by IncomingMailDaemon.java on the server. Both implementations extend the Thread Java class, and are identically coded.

Parser
This module is implemented by CParser.java on the client, and by the class ServerParser.java on the server. Each class is responsible for parsing email files that get downloaded. The ProcessMail(String mailFileName) function of each class initiates the parsing process. After a file is successfully parsed, the data from the file is stored in a Message object, which gets dispatched to the appropriate CProtocolManager.java handler function, in the case of the client, and to the appropriate ServerProtocolManager.java handler function in the case of the server.
ProtocolManager
This module is implemented by CProtocolManager.java on the client side and by ServerProtocolManager.java on the server. Each class provides a set of handler functions for the different types of messages that can be passed to it by the Parser module. For example, some handler functions included in CProtocolManager.java are:

```java
public void HandlePing(Message m)
public void HandlePingReply(Message m)
public void RegisterClientReply(Message m)
public void HandleHWUpdateReply(Message m)
public void HandleSearchQueryReply(Message m)
public void HandleSWUpgradeReply(Message m)
```

Some of the handler functions included in ServerProtocolManager.java are:

```java
public void HandlePing(Message m)
public void HandlePingReply(Message m)
public void RegisterClient(Message m)
public void HandleHWUpdateMsg(Message m)
public void HandleSearchQuery(Message m)
public void HandleSWUpgradeReplyMsg(Message m)
```

EmailBuilder
This module is implemented by the class EmailBuilder.java on the server, and by the class CEmailBuilder.java on the client. It is used to construct the headers and body of messages so that they can be emailed out. On the client, CEmailBuilder has the following important functions:

- ```java
BuildTEKRequest(int type, String parameters). This is used to construct Search messages and URL request messages.
```

- ```java
BuildEmail(int type, Hashtable parameters). This is used to construct all message types except Search and URL request messages.
```

- ```java
BuildEmail(String eaddr, String body). This is used to build messages for retransmission.
```

On the server, EmailBuilder provides the following important functions:
• BuildTEKReply(String AMID, String CID, int type, ByteArrayOutputStream b). This is used to construct Search query replies, and URL direct fetch replies. AMID is the MID of the message being replied to, b contains the zipped results to be sent as an attachment to the message.

• BuildEmail(String AMID, String CID, int type, Hashtable Parameters). This is used to build messages of all other types which can be sent from the TEK server.

OutgoingMailDaemon
This module is implemented by the class OutgoingMailDaemon.java on the server, and by the class COutgoingMailDaemon.java on the client. It polls the queue of messages constructed by the EmailBuilder module and emails them to their destination.

Retransmit Daemon
This module is implemented by ServerRetransmit.java on the server, and by the class Retransmit.java on the client. It is responsible for detecting expired requests, flagging them as duplicates, and passing them to the EmailBuilder module for delivery.

4.3 Message Exchange Examples

To demonstrate the use of the protocol, we will go through, in detail, the example of a TEK client submitting a query to the TEK server, and the scenario where this search request message is retransmitted.

4.3.1 Search

Consider the scenario where a user, joesuer types in the query “Dominica banana industry” into the user interface of a registered TEK client where the client’s CID is 79, and its email address is tekclient@world.com. Suppose that this query is query number 365. Here are the steps which follow:
1. The TEK client application uses the CEmailBuilder class to construct the search message to be sent to the TEK server.

2. CEmailBuilder.BuildTEKRequest( ) is called with the search parameters, and message type to construct a SEARCHQUERY message. The pseudocode for this method is given below:

```java
CEmailBuilder.BuildTEKRequest(int msgtype, String parameters)  
- read client ID from ClientID file  
- >>> call CEmailBuilder.GenerateID( ) to generate MID  
- >>> MID = CID + '_' + timestamp  
- construct message headers  
- assign <parameters> as the message body  
- add message to outgoingmailqueue  
- >>> call CEmailBuilder.AddToOutgoingMailQueue( )  
- add message to pendingreplylist  
- >>> call CEmailBuilder.AddToPendingReplyList( )
```

The message constructed is:

```plaintext
MID:79_99087184199  
EID:79  
TYPE:Message.SEARCHQUERY  
SOURCE:tekclient@world.org  
FN:1  
TNF:1  
DUP:false  
*** Body ***  
QID:365  
QS:Dominica banana industry
```

3. COutgoingMailDaemon, which periodically polls the client’s outgoingmailqueue for messages to deliver, picks up the SEARCHQUERY message and emails it to the TEK server.

4. The email containing the message above arrives at the TEK server and is filtered into the incoming mail folder.

5. The IncomingMailDaemon, which is running as a background daemon on the
TEK server picks up the new email from the incoming mail folder and hands it off to the ServerParser.ProcessMail(...) method.

6. ServerParser parses it into a Message object and gives it to ServerProtocolManager.HandleSearchQuery(...). The pseudocode for this method is given below:

```java
ServerProtocolManager.HandleSearchQuery(Message m)
- if m.GetType() = SEARCHQUERY
- then extract the body from m
- searchInfo = body parameters
- pass control to the TEK application
- >>> call Controller.TEKController(m.GetMessageID(), m.GetClientID(), searchInfo)
```

7. TEK controller processes query and calls EmailBuilder.BuildTEKReply to send results to client. The pseudocode for this method is:

```java
EmailBuilder.BuildTEKReply(String AMID, String CID, int msgtype, ByteArrayOutputStream b)
- generate a message ID for the message
- >>> call EmailBuilder.GenerateID(..)
- construct message headers
- extract email address of client we’re replying to
- >>> call get_client_email(CID) to get it from client_info_table
- create a a new Vector object v
- add the destination email address to v
- add the headers and body to v
- add the TRUE to v indicate there is an attachment
- store attachment in a file
- add the filename to v
- add message to mail_q table
- >>> call insert_in_mailq(..)
```

8. The TEK server’s OutgoingMailDaemon class polls the mail_q table, obtains the new message to mail out, and emails it to its destination.

9. At TEK client 79, the search response is filtered into the incoming mail folder and then it gets picked up by CIncomingMailDaemon which hands it to the
CParser for parsing.

10. The CParser parses the email contents into a Message object, determines that it is a SEARCHQUERYREPLY message, and calls CProtocolManager.HandleSearchQuery(Message m) which passes the search results files to the TEK Client Application for display.

4.3.2 Retransmission of a message

Suppose in the previous scenario, a week goes by without the client receiving a response to its “Dominica banana industry” query. Assuming that messages on the pendingreplylist file timeout after one week the search query message will be retransmitted. Here are the steps:

1. Retransmit daemon reads the pendingreplylist file and sees that the message with MID:79_99087184199 has expired.

2. Retransmit reads the message associated with MID:79_99087184199, and flags it as a duplicate. Here is the pseudocode:

```plaintext
Retransmit()
- let h be an empty Hashtable object
- loop:
  - if pendinglistfile exists
    - begin if
      - read the Hashtable object stored in this file into h
      - MIDList = h.keys()
      - for each MID in MIDList
        - begin for
          - date = extract date from MID
          - if date is older than one week
            - get email address, body for each for each MID in MIDList
            - newBody = old body flagged as duplicate
            - EmailBuilder.BuildEmail(dest, newBody)
        - end for
      - end if
    - end if
  - end loop
```
3. The message becomes:

```
MID:79.99087184199
EID:79
TYPE:Message.SEARCHQUERY
SOURCE:tekclient@world.org
FN:1
TNF:1
DUP:true
*** Body ***
QID:365
QS:Dominica banana industry
```

4. The new message is added to the outgoingmailqueue file by the EmailBuilder.

5. The OutgoingMailDaemon picks up the message from outgoingmailqueue file and emails the duplicate message to the TEK server.
Chapter 5

Testing

Testing of the protocol was done on two levels:

1. **Unit testing.** Each individual class has its own test method which tests all the functions that the class provides.

2. **Integration testing.** This involved testing how the protocol functioned in delivering a message from sender to recipient. We generated a set of test messages, with and without errors, for each message type that could originate from a client, and each message type that could originate from a server. We also tested the following cases:

   (a) Non-TEK email placed in the incoming mail folder.
   (b) TEK email with a non-existent CID.
   (c) TEK email with an invalid MID.
   (d) TEK email with a non-existent message type
   (e) TEK email with an invalid value in the DUP field.
   (f) No ClientID file at the client.
   (g) No outgoingmailqueue file at the client.
   (h) No pendingreplylist file at the client.
   (i) No lastheardfrom file at the client.
(j) Duplicate request message to client.

(k) Duplicate request message to server.

(l) Duplicate reply message to client.

(m) Duplicate reply message to server.

We believe that we have tried a reasonable set of tests. However, testing the protocol under the actual conditions in which it will be used, i.e. with a client in a developing country, was not possible because the TEK clients have not yet been deployed at their destination. Also, resource limitations did not allow us to perform stress tests of the protocol to determine how it would perform under heavy traffic. Further testing is planned as implementation of TEK continues.

Because the implementation of the TEK System itself is not yet complete, there were some features of the protocol which could not be tested for this thesis. This was the case despite our efforts to abstract the protocol functionality away from that of the TEK application. At the time of writing, most of the protocol functionality has been implemented and tested. To follow will be code optimizations, and several protocol extensions which are discussed in Chapter 6.
Chapter 6

Conclusion

Having discussed the protocol design, and built an implementation of it for use in the TEK Search System, we conclude by considering some possible extensions to the protocol. In this chapter, we will also explore different measures that can be used to evaluate the protocol.

6.1 Future Work

There are several possible extensions which could be made to the design and implementation of the protocol described in this thesis. We present some extensions which can be added in future upgrades of the protocol.

6.1.1 Fragmentation and Reassembly

The current implementation of the protocol does not fragment large messages for transfer between end points. Due to bandwidth constraints, the protocol must be able to fragment larger messages into smaller ones, transmit these smaller messages over the network, and then reassemble the fragments at the receiver site.

The idea is simple. The sender breaks up a large message into a set of smaller fragments. Each fragment of a message is assigned the same MID, and is fully addressed to the sender. In addition, each fragment has a fragment number, and a number
indicating the total number of fragments that the original message was split into. On receiving the first fragment of a multi-fragment message, the receiver creates an array to hold the first fragment, and other fragments as they arrive. If all the fragments are received, the receiver reassembles them into a complete message and then processes this larger message as it would a single-fragment message. If, however, one of the fragments is not received after a certain period of time, then the receiver requests that the sender retransmit the lost fragment. The identification mechanism used for the fragments can easily be used to detect misordered fragments, and the reassembly algorithm can be responsible for reordering them.

6.1.2 Message piggybacking

Another interesting extension to the protocol would be to use a “piggybacking” strategy to combine several messages into one email. This would be extremely advantageous to the client because it means that the client would receive several messages in one email, instead of only one. This could imply more efficient use of online time, and could possibly result in lower Internet access costs for TEK clients, since less emails would need to be exchanged between TEK entities in general.

Before messages are emailed from a sender, the outgoingmailqueue, which buffers messages created by the EmailBuilder until they can be emailed to their destination, can be examined. The size of all messages on this buffer can be calculated to check whether it is possible to package (“piggyback”) any messages to the same destination into one email, instead of delivering each separately. Smaller messages, such as some reply messages that essentially contain just the MID of the message to which they are responding, are good candidates for “piggybacking” atop one another.

6.1.3 Adaptive Timeouts

The current implementation of the protocol uses a fixed timeout for determining when to retransmit requests. On the server, an interesting extension to the protocol would be to use client-specific information such as average round-trip delay, to develop an
algorithm to provide timeout values which adapt to network conditions.

6.1.4 Client-side Database

We explained previously that the implementation of the protocol used a database on the server side for storage, while on the client side, we used files. One disadvantage of using the file system is that we had to be concerned with file concurrency if two processes were trying to edit the same file. For example, on the TEK client, the Parser module could be trying to remove a request from the pendingreplylist file at the same time that the EmailBuilder is trying to add a request to that file. Additionally, file access is slow.

An extension to the protocol would be for the client to use a simple database for storage of outgoing messages, incoming messages and messages that are awaiting a reply from the server. Apart from isolating concurrency issues, a database would facilitate more easily the maintenance of a client-side history of messages exchanged with the server. This would mean that on receiving a duplicate request from the server, the client would not have to regenerate a response, instead it could look it up in the history log.

6.1.5 TEK-specific extensions

**TEK Implementation:** The current implementation of the protocol for TEK includes the framework to handle the database synchronization message-pairs, software upgrade message-pairs, and data upgrade message-pairs. However, the existing version of the TEK Search Engine does not yet provide such functionality; it will be supported in future versions of TEK.

**Server-to-Server Communication:** The design of the TEK system allows for more than one TEK server. Hence, if in the future there is a need for exchange of information among TEK servers, the protocol will have to be capable of handling this. For example, if a server is overloaded with requests, it may forward a search query to
another TEK server for processing. The protocol can be extended to support this.

### 6.2 Possible Evaluation Techniques

In the previous chapter we discussed the testing strategies used to test the implementation of the protocol, however we did not evaluate it. Evaluating the protocol without all of its features being fully implemented and without the experience of its use in its intended environment is somewhat difficult. However, determining how to evaluate the protocol, that is, what evaluation metrics to use, is relatively straightforward.

A good starting point to motivate the evaluation process would be to review the original aims of the protocol. The overall goal was to ensure reliable end-to-end delivery of data over a low-bandwidth, low-connectivity, high-cost network. We can extrapolate several micro-goals from this goal:

- First, to ensure that data could be sent over a low-bandwidth network, one aim was to keep the size of messages small.
- Second, to overcome the problems resulting from low connectivity, one aim was for the protocol to be able to take advantage of spurts when the network would be most reliable.
- Third, to reduce the cost to the user, one aim was to ensure that the amount of time the protocol had to be connected to the Internet was short; furthermore the number of round-trip communications needed to complete a transaction had to be low.

With these micro-goals in mind, we identify several metrics which can be used to evaluate the protocol:

1. **User satisfaction.** This metric is closely coupled with the application into which the protocol is integrated. In the case of TEK, one might ask the following question: If a user sits before a TEK client terminal and types a search query
into the user interface, when he returns an hour later (or two hours, a day, or a week later) have the search results arrived?

2. **Efficiency.** We can measure how efficiently the protocol makes use of available bandwidth by studying the average size of the emails it sends, in the context of the network conditions at the client, and more specifically, in the context of the bandwidth available at the client.

3. **Recovery mechanism.** How does the protocol behave if a client machine suddenly dies? Does it keep delivering messages to the client, thereby potentially flooding the client when it comes back up? Should it require re-registration of clients that have been inactive for long periods of time? To some extent, this behavior should be specified at the end-points, that is, by the application into which the protocol is integrated.

4. **Behavior in the face of failure.** We can measure how successfully the protocol handles failure. This would give an idea of the reliability of the protocol. Not only is it important to look at how well the protocol responds to the different failure modes of email, but also, we need to examine how often the protocol retransmits messages in error.

### 6.3 Conclusion

As usage of the Internet continues to grow, disparities between the quality of the network infrastructure in the developing world and in the developed world persist. In this thesis, we addressed the problem of how to exchange digital information across an unreliable, low bandwidth, low connectivity network. We presented a design and prototype implementation of an email-based, low-bandwidth protocol designed to work under network conditions characterized by episodic connectivity. With little effort, the protocol can be tailored for reliable end-to-end data delivery for other applications directed at tackling issues similar to those that the TEK Search System serves to address.
Appendix A

Database Tables

In this appendix, we describe the fields of the tables used by the protocol on the TEK server, and specify the methods used to manipulate the data that gets stored.

**client_info_table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>client_id</td>
<td>not null varchar2(20)</td>
</tr>
<tr>
<td>email</td>
<td>varchar2(20)</td>
</tr>
<tr>
<td>administrator</td>
<td>varchar2(50)</td>
</tr>
<tr>
<td>platform</td>
<td>varchar2(20)</td>
</tr>
<tr>
<td>opsys</td>
<td>varchar2(20)</td>
</tr>
<tr>
<td>disk_space</td>
<td>varchar2(20)</td>
</tr>
<tr>
<td>modem_speed</td>
<td>varchar2(20)</td>
</tr>
<tr>
<td>country</td>
<td>varchar2(20)</td>
</tr>
<tr>
<td>search_lang</td>
<td>varchar2(50)</td>
</tr>
<tr>
<td>results_lang</td>
<td>varchar2(50)</td>
</tr>
<tr>
<td>timeout</td>
<td>varchar2(50)</td>
</tr>
<tr>
<td>reg_date</td>
<td>date</td>
</tr>
<tr>
<td>reg_status</td>
<td>varchar2(20)</td>
</tr>
<tr>
<td>organization</td>
<td>varchar2(50)</td>
</tr>
<tr>
<td>last_heard_from</td>
<td>date</td>
</tr>
</tbody>
</table>

public static void **update_modem_speed** (String client_ID, String modem_speed)

public static void **update_platform** (String client_ID, String platform)

public static void **update_opsys** (String client_ID, String opsys)

public static void **update_disk_space** (String client_ID, String disk_space)

public static void **update_administrator** (String client_ID, String administrator)

public static void **update_results_lang** (String client_ID, String results_lang)

public static void **update_reg_status** (String client_ID, String reg_status)

public static void **update_email** (String client_ID, String email)
public static void set_last_heard_from (String client_ID, Date msg_date)
public static Date get_last_heard_from (String client_ID)
public static String get_client_email (String client_ID)
public static String register_client (String email, String administrator, String platform, String opsys, String disk_space, String modem_speed, String country, String search_lang, String results_lang, String reg_status, String timeout, String organization)

mail_q

mail_qid not null number(38)
eaddr varchar2(100)
msgbody varchar2(1000)
has_attachment_p char(1)
attachment_filename varchar2(255)

public static Vector remove_from_mailq ()
public static void insert_in_mailq (Vector in_msg)

awaiting_reply_list

mid not null number(38)
date_sent date
eaddr varchar2(100)
msgbody varchar2(1000)
has_attachment_p char(1)
attachment_filename char(255)

public static Vector find_awaiting_msg (String MID)
public static String [] find_awaiting_msg (Date search_date)
public static String [] find_awaiting_msg (String eaddr, int i)
public static void remove_awaiting_msg (String findMID)
public static void close_awaiting_msg (String find_MID, Date client_recd_date, Date close_date)
public static void insert_awaiting_msg (String MID, Date date_sent, String eaddr, String msgbody, Boolean attach_p, String filename)
closed_msg_list

mid not null varchar2(50)
date_sent date
date_recd date
date_confirmed date
eaddr varchar2(100)
msgbody varchar2(1000)

public static Vector find_closed_msg (String MID)
public static String [] find_closed_msg (Date search_date)
public static String [] find_closed_msg (String eaddr, int i)
public static void remove_closed_msg (String find_MID)
public static void insert_closed_msg (String MID, Date date_sent, Date date_recd, Date date_confirmed, String eaddr, String msgbody)

problem_msg_list

mid not null number(38)
date_sent date
eaddr varchar2(100)
msgbody varchar2(1000)
has_attachment_p char(1)
attachment_filename varchar2(255)
problem_date date

public static Vector find_problem_msg (String MID)
public static String [] find_problem_msg (Date search_date)
public static String [] find_problem_msg (String eaddr, int i)
public static void remove_problem_msg (String find_MID)

sent_mail_table

sent_msg_id not null number(38)
email varchar2(100)
msg varchar2(2000)

public static void store_sent_msg (String email, Object newmsg)
Bibliography


   http://www.ipnsig.org/home.htm

   http://www.ipnsig.org/reports/TCP_IP.pdf.

