An XML Messaging Protocol for Multimodal Galaxy Applications

by

Justin H. Kuo

S.B., Massachusetts Institute of Technology (2001)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

© Massachusetts Institute of Technology 2002. All rights reserved.
An XML Messaging Protocol for Multimodal Galaxy Applications

by

Justin H. Kuo

Submitted to the Department of Electrical Engineering and Computer Science on May 24, 2002, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented an XML-based messaging protocol for interfacing GALAXY systems with external applications. This protocol is TCP-based, and data are sent in the form of messages. Messages are well-formed and easily-parsed lines of text that encode separate ASCII and binary payloads. Protocol specifications are defined that provide robustness and flexibility. The ASCII portions of messages are encoded using XML to offer extensibility for using the protocol across a wide variety of applications. The binary portions carry raw data and can be used to stream data across several messages. We demonstrated the C++ implementation of this protocol in several test applications. These included simple variations of instant messaging that let users send text messages to each other, and a modified GALAXY audio application that transmits audio from one machine to another.

Thesis Supervisor: James R. Glass
Title: Principal Research Scientist

Thesis Supervisor: D. Scott Cyphers
Title: Research Scientist
Acknowledgments

First and foremost, I would like to sincerely thank my thesis advisors, Jim Glass and Scott Cyphers, for their guidance throughout this project. Starting from my senior year when I joined the Spoken Language Systems group as a UROP student, they have helped me at every step of the way to develop and complete this thesis. I am grateful for all of their time, patience, and understanding.

I would also like to extend my thanks to all the SLS staff and students. They have provided a very friendly and stimulating environment in which to do my thesis, and it has been a pleasure getting to know everyone during the past year here. I wish everyone the best.

In addition, thanks to all the friends I have met during my time here at MIT. They have been an important part of my memorable college experience and I have learned a great deal from them. I will miss them when I leave MIT, but I know they will always be with me in the future.

Finally, my sincere thanks go out to my parents who have always been there for me the past 22 years. I thank them for all of their love and understanding that only parents could give to their son. Without them, I truly would not be who I am and where I am today. Thanks, Mom and Dad!!

This research was funded by an industrial consortium supporting the MIT Oxygen Alliance.
## Contents

1 Introduction 13
   1.1 Motivation ........................................... 14
   1.2 Goals ................................................... 14
   1.3 Approach .............................................. 15
   1.4 Outline ............................................... 16

2 Background 19
   2.1 GALAXY Architecture .................................... 19
      2.1.1 Interaction with External Applications .............. 21
   2.2 Remote Procedure Calls .................................. 22
   2.3 Message Passing .......................................... 24
   2.4 XML .................................................... 24
      2.4.1 Highlights of the XML Format ......................... 25
   2.5 Galaxy Frame Relay ..................................... 27

3 Protocol Description 29
   3.1 Overview ............................................... 29
   3.2 Control Flow ............................................ 29
   3.3 Message Payload ......................................... 30
      3.3.1 ASCII .............................................. 30
      3.3.2 Binary ............................................ 31
   3.4 Message Format .......................................... 31
      3.4.1 Envelopes .......................................... 32
6.1.2 Text Messaging, Version 2

6.1.3 Text Messaging, Version 3

6.1.4 Galaudio/XMLaudio

6.1.5 Other Programs

6.2 Observations

7 Conclusion

7.1 Discussion

7.1.1 Current State

7.1.2 Comparison with Frame Relay

7.1.3 Use in Pervasive Computing

7.2 Future Directions

7.2.1 Language Bindings

7.2.2 More on Envelopes

7.2.3 Namespaces

7.2.4 Integration with Existing Systems and Beyond

A Tag Source Code

B MsgWriter Source Code

C MsgReader Source Code
List of Figures

2-1 A typical GALAXY configuration of an SLS spoken language system with the central hub connected to various specialized servers .......................... 20
2-2 GALAXY semantic frames resemble this sample frame, generated by SpeechBuilder for the user query, “What is the weather in Boston Massachusetts on Monday?” ........................................................................ 21
2-3 Communication between client applications, multiple instances of resources, and a GALAXY system for an SLS conversational interface ......................................................... 22
2-4 Illustrative examples of well-formed XML .................................................. 26
2-5 Sample XML with well-formed and nested structure, freely-defined tags, and variable number of key-attribute pairs ......................................................... 26
3-1 Illustration of the ENVELOPE tag surrounding other XML tags ............ 32
3-2 Syntax of the three kinds of lines in a message ......................................... 33
3-3 Illustration of two- and three-line messages ............................................. 34
5-1 Inheritance tree with little tag hierarchy ................................................. 56
5-2 Inheritance tree with more tag hierarchy ............................................... 56
5-3 High-level diagram for the state machine for the message parser ........ 57
Chapter 1

Introduction

Recent developments in perceptual technologies like speech and vision recognition have driven the movement towards forms of input that allow more natural, human forms of interaction with computers. These modes of input can either replace or extend the set of traditional modes, including the keyboard, mouse, and stylus. Thus in a multimodal environment, with multiple input modes available at a person’s disposal, the user need not conform to the restricted types of input to the computer, but rather, the system will conform to the forms of interaction the user prefers.

The Spoken Language Systems Group (SLS) at the MIT Laboratory for Computer Science has been actively developing spoken language systems using their GALAXY architecture [17, 19]. GALAXY is a client-server architecture that uses a central “hub” to mediate among multiple human language technology (HLT) servers; it is an infrastructure for conversational systems, with potential to support multimodal systems. SLS has successfully developed, implemented, and deployed conversational systems in several domains such as weather information, airline travel, and traffic navigation, all within the GALAXY framework [17].

This thesis develops a messaging protocol to leverage current GALAXY-based systems for use in separate applications developed outside of the GALAXY framework. The motivation for this approach will be discussed, and the challenges and decisions during development will be addressed. This thesis will conclude with the current state of work as well as future considerations.
1.1 Motivation

The premise behind this work stems from the MIT Oxygen project [16], an initiative to bring about and further develop pervasive computing. The Oxygen infrastructure defines and supports the necessary devices and networks that enable multimodal environments for the integration of different forms of HLT.

Existing applications developed for Oxygen run on handheld Compaq iPAQ computers connected to a wireless network. The applications provide a front-end UI (user interface) that communicates with independent HLT servers. For example, an iPAQ can serve as a mobile microphone and speaker for a conversational interface, where all the speech technology is located on a remote spoken language server.

SLS has developed several conversational systems (e.g., Jupiter [24], Mercury [18, 20]) – most recently, the SpeechBuilder utility [12] – to aid in the process of building custom spoken dialogue systems. The relative success and flexibility demonstrated by these systems make them attractive for use in the Oxygen environment. However, the current systems reside entirely within the GALAXY framework, and lack direct support to interface with applications outside of GALAXY. Hence, there is a need for some method to facilitate communication between applications and GALAXY systems. Such a method would enable developers to build applications that rely on multimodal environments supported by the GALAXY framework, in particular, the SLS conversational interface for user speech interaction.

1.2 Goals

The overall goal of this work is to design and implement a protocol that will enable applications and GALAXY servers to communicate with each other in a structured and methodical fashion, using messages and data streams across a network. The aims for this protocol are that it: 1) be simple and easy to use, 2) be flexible, and 3) support basic debugging and error-handling functions.

The protocol must be easy to use so that application developers do not spend in-
ordinate amounts of time trying to interface their application with a GALAXY system. They should also find it relatively straightforward to integrate the protocol package into existing applications without rewriting significant portions of code. Therefore, the API for the protocol must be fairly simple and intuitive.

The protocol must also be flexible, with the ability to transfer many kinds of data relevant to the application that uses it. In general, both ASCII and binary data should be supported, along with some encoding of how the data are structured or how it should be dealt with. Although the addition of flexibility introduces a slight trade-off with simplicity of design, the highest priority at the expense of the first two qualities is placed on implementing features essential to basic functionality whose details can be abstracted away from developers and outside code.

Lastly, there must exist some form of debugging and error-handling in this protocol. Internal debugging aids in protocol development and testing; additional debugging flags can be exposed to facilitate application developers in monitoring the state of messages and other data transferred across the network. To preserve the protocol abstraction, the extent to which the debugging features expose the internals of the protocol mechanisms would go only as deep as the API exposes. Also, error-handling must be present so that the protocol would gracefully abort and exit, should an error occur. Coupling the error-handling routines with internal debugging statements will help during development and testing as well.

1.3 Approach

From the start, it was decided that this messaging protocol would be a high-level messaging system that worked on top of existing network protocols. To evaluate the feasibility of the protocol at an early stage, the initial design was simple and the implementation straightforward. From a high-level perspective, the role of this protocol was to shuttle events back and forth between two points on a network, and attach binary data associated with certain events.

The original idea for the protocol resembled HTTP headers [3]. Events encoded
as plain ASCII text were transmitted as cleartext across the network; the encoding was such that each event was placed on a separate line of text that would be parsed by the receiving end. In addition, binary data, delimited by some arbitrary "special" character, could also be transmitted in the same data stream, on separate lines from ASCII data.

The present protocol and the direction of its development have not deviated from the original idea in terms of basic concepts and foundation. The current approach has been to add certain features and to make refinements to certain aspects that make it simpler and/or more efficient in design and code. Some of these areas of focus include:

- separation of outgoing and incoming message functions
- representation and formatting of ASCII messages
- parsing of entire messages, extraction of payloads
- mechanisms for handling parsed messages

Note that certain considerations, such as security, threading, and multitasking, have not been dealt with at this stage. More emphasis has been placed on developing the basic framework, before scalability and performance issues are to be optimized.

1.4 Outline

Chapter 2 presents background on various protocol standards and concepts that were reviewed and that have influenced the design of the messaging protocol. On-going work in GALAXY is also presented, and its relevance and relation to the messaging protocol are discussed.

Chapter 3 gives a high-level overview of the messaging protocol. It describes the message functions, format, and structure, along with the mechanisms that deal with sending, receiving, and handling messages.
Chapter 4 presents the software API to implement the messaging protocol in C++, and describes how to use it in an application.

Chapter 5 gives more detail about the C++ implementation. It looks at the concerns and issues raised as the implementation was developed, and briefly mentions some of the specific structures and algorithms used.

Chapter 6 describes the various test programs that have been written to help debug and evaluate the protocol API, in terms of scalability and robustness.

Chapter 7 summarizes this thesis, discussing the current state of the protocol, and concludes with future considerations for extending or adding to the existing set of features.
Chapter 2

Background

This work draws its foundations from various concepts and existing systems, which are described here.

2.1 GALAXY Architecture

GALAXY is the architecture developed at MIT by SLS for developing conversational systems. The first implementation of GALAXY was used to build spoken language systems that access on-line information using spoken dialogue [13]. It was subsequently redesigned with enhancements for more flexibility in building conversational systems [17].

GALAXY is a client-server architecture that uses a central hub to mediate interaction between various HLT servers (Figure 2-1). The hub is programmable with a rule-based scripting language. Interactions between the hub and the other servers or components are mediated by these hub scripts. Although GALAXY has been used primarily for building conversational systems, it can support any HLT server, making it a platform for building multimodal systems.

Servers in the GALAXY architecture share a common language framework to represent any data they jointly process. Currently, the HLT servers built by SLS use semantic frames as the meaning representation format. Figure 2-2 illustrates a sample semantic frame generated by the SpeechBuilder utility. System developers who decide
Figure 2-1: A typical GALAXY configuration of an SLS spoken language system with the central hub connected to various specialized servers
Figure 2-2: GALAXY semantic frames resemble this sample frame, generated by SpeechBuilder for the user query, “What is the weather in Boston Massachusetts on Monday?”

to use other servers in the GALAXY framework are free to choose a different meaning representation protocol [17], as long as the different servers can still communicate with one another. Again, it is the hub’s responsibility to mediate communication between the servers, regardless of the format used.

In addition, a GALAXY hub maintains the notion of a session that is initiated every time a new user interacts through the front-end user interface component of the GALAXY architecture. Sessions are useful for managing interactions with multiple users simultaneously via a single hub instantiation. Distribution of system resources among multiple sessions is also mediated by hub scripts.

Finally, the GALAXY hub also handles a process called brokering. When one server sends binary data to another server, the data are not routed through the hub. Instead, the hub brokers a connection between the two servers through which one sends data directly to the other. This reduces the amount of traffic through the hub, leaving it free to process regular messages and frames that get passed between servers.

### 2.1.1 Interaction with External Applications

The paradigm for building multimodal GALAXY applications will closely follow the SLS framework for developing the clients for their conversational interfaces. In this distributed approach, client applications consist of abstract components that need not necessarily reside in the same physical location.

For example, a generic client for a conversational interface is composed of the main application and two principal resources: a microphone and a speaker. Resources
manifest themselves as virtual entities managed by software daemons. Thus, one microphone or one speaker appears as several virtual devices used by multiple clients. This property also enables clients to access more than just one of each type of resource (e.g., broadcasting audio to more than one speaker) as long as the control software is capable of managing connections. Any audio data transferred between the GALAXY hub and the client is relayed to the appropriate audio input or output device.

Management and integration of all these components requires precise communication between processes. In a setup like that shown in Figure 2-3, there is communication between the client and its resources, as well as the client and the main GALAXY hub. The need for such a communication mechanism is one of the primary motivations for developing a messaging protocol.

2.2 Remote Procedure Calls

Remote procedure calls (RPC) address the issue of communicating in a distributed computing environment. The RPC model provides the mechanism enabling procedure calls that occur on a single computer to also occur across a communication network.
Typically, one process calls a procedure in another process using RPC. In effect, RPC opens up an interface where client-server communication is achieved and where a program has access to procedures located in external systems.

RPC architectures utilize a technique often called *marshaling* to take arguments and other data relevant to a procedure call and package them into a request that is sent across the network. The receiver *unmarshals* the request into the original contents and calls the corresponding local procedure, after which the return value is sent back to the sender.

There are many examples of protocols that follow the RPC model, such as SunRPC [21], Common Object Request Broker Architecture (CORBA) [8], DCOM, Java RMI, and SOAP [14], all of which have advantages and disadvantages. These examples are worth glancing at since the high-level behavior of RPC, facilitating function calls and the transfer of data between networked machines, directly parallels one of the major goals of this messaging protocol.

Many of the details in these standards, however, are far too complex and low-level for the messaging protocol. For example, the External Data Representation (XDR), used by both SunRPC and the Network File System (NFS), describes a way to encode data for transfer between different computer architectures [22]. It offers a language to describe data formats (i.e., with data types and block sizes) at a low-level that exceeds the needs of the messaging protocol to reliably ship bytes – regardless of data type – across the network. Observations from these standards have more or less contributed basic ideas rather than specific implementation details for the messaging protocol.

Another issue to consider is that some RPC protocols use binary encoding formats. While binary gives rise to compact data formats, an ASCII protocol carries with it simplicity and portability. ASCII is easier than binary to debug, and does not suffer from interoperability issues (such as little-endian vs. big-endian conventions) like binary does when running on different architectures or operating systems. Such concerns are important during the design of the messaging protocol, which should remain simple as it is developed, evaluated, and refined.
2.3 Message Passing

The basic concept behind message passing is that of processes communicating through messages. This paradigm is well-understood [6, 10] and there are many implementations of this spanning various applications. For example, messages are used in a typical graphical windowing system (like X-windows or Microsoft Windows) to convey events or commands between the windowing components and subsystems. Message passing can be efficiently and portably implemented [6], which is an attractive quality for this thesis.

In a message passing system, messages can conceivably be very complex in structure, and not necessarily encoded in plaintext. The messaging protocol in this thesis favors simplicity, and thus the messages it uses are simple lines of formatted text that are carriers of both ASCII and binary data. Messages can be either broadcast over a common channel or sent point-to-point; having both options allows flexibility in the messaging protocol’s design.

Another advantage of adopting the message-passing paradigm for use in GALAXY applications is that it features a “push” model where messages push relevant information from senders to recipients. Unlike in some RPC protocols, application control flow is not driven by the arrival of a message. Instead, it can operate asynchronously, and execute other tasks between the time it is interrupted by an incoming message.

2.4 XML

The Extensible Markup Language (XML) is a subset of the Standard Generalized Markup Language (SGML) and was designed to provide structured formatting of data for the programs that process XML documents [5]. XML resembles Hypertext Markup Language (HTML) [2] in its use of tags and attributes (see Figure 2-5), but adds more power by letting programs that encode XML documents define what the tags mean. Moreover, XML provides a text formatting that is platform-independent, is extensible, and supports internationalization/localization.
XML is the preferred choice of encoding for ASCII data in the messaging protocol. Its extensible nature allows for flexible specifications that ensure that we could easily change the data format if the need arises, without much effect on application developers. Furthermore, XML allows generic support for variable message formats without having to build complex debugging tools before the full specification has been defined and implemented. Also, since XML is well-supported, there are a variety of tools (e.g., parsers) available, some open-source, for use during development and implementation.

2.4.1 Highlights of the XML Format

Relevant terminology (in boldface) and a brief overview of XML are presented here:

- **XML elements** are delimited by start-tags and end-tags which enclose more text or markup between them.

- A tag is an element name preceded by a less-than (<) symbol and followed by a greater-than (>) symbol. End-tags are identical to start tags except that the < is immediately followed by a forward slash (/) (Figure 2-4a). *Tag names are case-sensitive*, and start-tags and their corresponding end-tags must be consistent with each other.

- Elements without any data can be compressed into one tag preceded by a less-than (<) symbol, and followed by a forward slash and greater-than symbol, in that order (/>) (Figure 2-4b).

- Elements may have zero or more **attributes** embedded in their start-tags. Attributes consist of name-value pairs, also called attribute-value pairs. The name and value must be written in the form **name="value"**, and the value must be contained in matching single or double quotes (Figure 2-4c).

- Elements may be arranged in an infinitely nested hierarchy. The first element is denoted the **root** document element; all other elements are the children of the root element (Figure 2-4d).
(a) Sample XML element with a start-tag and end-tag:

    <element>This is text inside an XML element</element>

(b) Sample XML element with no data, written with one tag:

    <one-tag/>

(c) Sample XML element with three attributes in the start-tag, no data:

    <person name="Justin Kuo" ID="9876" phone="555-1234"></person>

(d) The same data as (c), formatted using nested tags; here, the root element is "customer":

    <customer ID="9876">
        <lname>Kuo</lname>
        <fname>Justin</fname>
        <phone>555-1234</phone>
    </customer>

Figure 2-4: Illustrative examples of well-formed XML

<person status="graduate student">
    <name>
        <first-name>Justin</first-name>
        <last-name>Kuo</last-name>
    </name>
    <school name="MIT" location="Cambridge, MA"/>
    <degree level="M.Eng." dept="EECS"/>
    ...
</person>

Figure 2-5: Sample XML with well-formed and nested structure, freely-defined tags, and variable number of key-attribute pairs
2.5 Galaxy Frame Relay

The frame relay, written by Scott Cyphers, is a recent addition to the GALAXY software suite. It is similar to the messaging protocol in that it allows GALAXY and non-GALAXY components to communicate with each other through messages [9].

A frame relay is a software mechanism that can send or receive messages across a TCP connection. As the name suggests, it passes data along from the source to receivers at the destination. Destinations in this scheme are URLs that support a naming hierarchy similar to HTTP URLs; the frame relay offers messages to successive receivers starting at the root and descending down the path in the URL until one accepts the message. There is also a remote relay, which is simply a frame relay that does not run in the same process as its client.

Messages in the frame relay consist of an envelope and body. The envelope provides fields that specify the destination and other routing information, and the body contains the rest of the message. Messages are sent as either frames or “blobs”. GALAXY or NIST representation of XML frames are supported. Blobs are used to send chunks of binary data. Binary data can span multiple blobs to support streaming of data: a header chunk is followed by zero or more intermediate chunks that comprise the blob, which is ended by a trailing chunk signaling the end of the blob message.

Frame relay messages are prepended with special two-character delimiters ($C for GALAXY frames and $F for NIST XML frames) and followed by a carriage return and newline character (\r\n). Messages can contain special commands directed to the frame relay that control Galaxy session settings. For example, audio settings can be configured for microphones, speakers, and endpoint detectors; event listeners can be added or removed; sessions can be activated or stopped, all through the use of frame relay messages. Most of these actions would normally be performed by hub scripts found on the hub. With frame relay, these actions can now be controlled by GALAXY and non-GALAXY components.

Frame relay is currently a temporary measure for dealing with resource location
and management. It is an integrated package that internally manages name servers and resource locations. In the future, resource management should be a separate component to enforce modularity, making routing mechanisms independent from actual content and delivery. This is an important point to keep in mind for the design of the messaging protocol.
Chapter 3

Protocol Description

3.1 Overview

This messaging protocol is TCP-based and transmits ASCII messages. Messages have an ASCII and optional binary payload, and are transmitted as cleartext. ASCII data are encoded using XML, making it extensible, customizable, and easy to parse.

The protocol is used for point-to-point communication. Messages are “composed” and sent by a source, then received and handled by the destination. These messages need not come in as a request and reply pair; sending messages using a reliable protocol like TCP means the sender can push messages across to its destination without expecting an acknowledgment. Two-way communication is also possible (if necessary) by reversing the roles of source and destination across the same TCP socket.

The rest of this chapter describes the messages, their operation, structure and formatting, and the actions involved in writing and reading messages.

3.2 Control Flow

The general flow of operation is fairly simplistic. This protocol operates by pushing information from point to point on the network. It requires that two independent points first establish a TCP connection (called a socket), which allows the two points
to communicate with each other. This is like a telephone system where two telephones connect to each other and establish a line for private conversation. Messages are then transmitted across the socket.

Both points listen in on the socket for incoming messages from the other end. One end communicates to the other by sending a message out on the socket, which is received if the other end is listening. By virtue of TCP ensuring reliable transport, messages sent in sequence are received in sequence. When a message is received, it is parsed and dispatched on the payload. Messages are dispatched immediately upon receipt to maintain correct sequencing of messages (also, most applications do not require delayed message dispatching).

As long as the socket remains open, both points are free to communicate with each other. When the socket is closed, the two points can no longer send messages to each other. This is as if the two telephones in the previous example hung up.

### 3.3 Message Payload

As mentioned before, messages can contain both ASCII and binary data. Each kind of data is physically separated from the other in a message. The ASCII portion of the message is used to hold “events” in the system. The binary portion is optional, and holds any data associated with particular events from the ASCII portion of the same message.

#### 3.3.1 ASCII

ASCII data refers to the fact that there is no binary encoding associated with this portion of the message. In other words, one could grab this data off the network, output it directly to the screen, and read its contents.

To provide structure and formatting that is easily parsed and processed when the message is received, ASCII data are encoded in XML. In this protocol, unique XML tags represent separate events; every event should have some associated behavior to dispatch when that event is present in the message. These are called “event handlers”
3.3.2 Binary

Binary data, on the other hand, are the raw bytes pertinent to whichever application it originated from. This protocol does not impose any structuring or formatting on the data since that would unnecessarily modify it.

Binary data in a message must be associated with an event (an XML tag) from the ASCII portion of the same message. Binary handlers (Section 3.5.2), similar to the aforementioned event handlers, perform some user-defined function on the data once it is received. The focus of this messaging protocol is to transmit events and ancillary data, therefore no message may contain only a binary portion. Instead, there should be an XML tag corresponding to the event of sending the binary chunk, and which can contain attribute-value pairs that describe that chunk.

3.4 Message Format

Messages are either two or three lines of text, where:

- the first line contains ASCII payload
- an optional second line contains binary data
- the third line (or second, if there is no binary data) is a special marker for end-of-message (EOM)

Lines also have specific formatting rules:

- all lines end with the following two-character sequence: a \r (carriage return) followed by \n (newline)
- all lines are preceded with one of two two-character sequences that signify what kind of payload is contained on that line
- lines preceded by two dollar signs (\$\$) contain ASCII payload
• lines preceded by a dollar sign followed by a hash ($#) contain binary payload
• the EOM line has empty content, and is simply two dollar signs appended with linefeed-newline pair: $$

3.4.1 Envelopes

All ASCII data (formatted in XML) are enclosed by an “envelope”, an idea borrowed from SOAP [14]. Analogous to paper envelopes, the XML envelope tag and its corresponding closing tag are the first and last XML tags in the ASCII portion, and whose attributes indicate delivery information such as the recipient and sender of the message.

The current implementation reserves the words FROM, TO, MSGID, and SESSID to refer to the sender, recipient, message ID, and session ID, respectively. These remain unused until a later time when they will be adapted for use in a GALAXY environment. Possible applications for envelope tags are discussed in Section 7.2.2. Figure 3-1 shows an example of an ENVELOPE tag surrounding other XML data.

```xml
<ENVELOPE FROM="sender" TO="recipient" MSGID="1234" SESSID="4321">
  <student>
    <name first="Justin" last="Kuo"/>
    <degree dept="EECS" type="M.Eng."/>
  </student>
</ENVELOPE>
```

Figure 3-1: Illustration of the ENVELOPE tag surrounding other XML tags

3.4.2 Binary Length Header

Lines that contain binary payload must also indicate the length of the binary chunk by prepending to the chunk a fixed-length header encoding the chunk length. The current implementation uses a 4-byte, or 32-bit, unsigned field to encode the length in bytes. Hence, a message can hold a chunk of maximum length $2^{32}$ bytes, which is a practical size for applications that use this protocol.
The header is needed because without prior knowledge of the length, it is difficult for a parser to correctly isolate the entire chunk. The only distinguishing sequence of characters marking the end of the chunk is the \r\n pair that terminate a line. This sequence, however, could be data in the binary chunk, and it would be incorrectly parsed as the end of the chunk.

Zero-length binary payloads are also allowed, and act as special markers to the application receiving messages. The application defines how to respond to these zero-value binary length headers. For example, a program that sends an audio file split up into several messages can use a zero-value header to signal the end of the data, at which point the receiving end knows to piece together the many binary chunks into a complete, reconstructed audio file. The zero-value header convention is not required, but is supported if an application adopts it.

### 3.4.3 Sample Messages

Figure 3-2 illustrates the general appearance of the three possible kinds of lines in a message. Figure 3-3 gives examples illustrating how these lines compose two- or three-line messages. Messages with two lines contain only ASCII payload and EOM, while messages with three lines contain both ASCII and binary payloads (ASCII line first, binary line second, EOM last).

- **line with ASCII payload:** $$<\text{ENVELOPE} ... >[\text{XML data}] </\text{ENVELOPE}>\r\n$$
- **line with binary payload:** $$\#[\text{fixed-length header}][\text{binary chunk}]\r\n$$
- **End-of-message line:** $$\$$

Figure 3-2: Syntax of the three kinds of lines in a message

### 3.5 Message Processing

Message processing refers to the actions involved in composing and receiving messages. These are treated as two separate tasks, each handled by its own dedicated
(a) An example of a two-line message

(b) An example of a three-line message

Figure 3-3: Illustration of two- and three-line messages

component. This is merely to allow for modularity in the implementation; conceptually, this has no impact on the protocol design. Both are equally important tasks, however, and are further described here.

3.5.1 Composition

Writing messages can be approached in several ways leading to many forms of implementation. In general, the ASCII and binary portions must be assembled before being formatted according to the syntax requirements. We present a straightforward mechanism to facilitate assembling message payloads. Here, “composing a message” refers to all actions or events performed during the process of assembling and accumulating a message’s payload, before the message is ready to be written and sent across the network.

Payload Queues

Assembling data requires some type of buffering mechanism, such as a queue, to hold the data before the message is written. The payload queue holds ASCII and binary data until the message is ready to be written. Once the message has been written, the queue is flushed (emptied), ready to hold payload for the next message. Hence, the queue only holds the payload for one message at any given time.

Concerning ASCII data, an entire ASCII payload is encoded in XML and enclosed
in an envelope tag. This means that multiple tags can reside within the ASCII payload, rather than sending one tag per message. This is important when considering network bandwidth as a limited resource: the amount of computational and network overhead imposed by syntax, formatting, and parsing would make the scheme that sends one tag per message inefficient. This protocol favors the scheme where multiple tags are packaged into one message, but also supports writing only one tag per message if so desired.

The payload queue is most applicable for accumulating ASCII data. The queue takes in some representation of an event that can be converted into XML, and stores these until the message is ready to be written. The payload queue may not be modified before it is flushed, though this feature is under consideration.

Binary data need not be queued as the message is assembled. Since a binary chunk is associated with one of the XML tags in the message, the message can be written and sent as soon as the binary chunk is passed in, assuming that its associated tag has already been queued in the ASCII payload. To ensure this, this protocol imposes the contract that the XML tag must be queued before its binary chunk is passed in. Note that the format of messages allows only one binary chunk, which can only be associated with one XML tag from the same message.

Procedure

The following is an informal algorithm for writing the message once all of the payload is available:

1. For each entry $i$ in the payload queue (in order from oldest to newest),
   
   (a) Write the XML encoding of $i$ to string $j$
   
   (b) Generate an envelope $e$ and enclose $j$ in $e$.
   
   (c) Prepend and append the appropriate special characters to $e$ to denote ASCII payload, ASCII

2. If there is a binary chunk, $b$,
   
   (a) Compute its length in bytes and store as 4-byte header field $h$
(b) Prepend h to b, and insert the special characters to denote binary payload BIN.

3. Generate an EOM, EOM

4. Concatenate ASCII, BIN (if it exists), and EOM in that order to produce a finished message M

5. Send M when ready

3.5.2 Dispatch and Handling

Messages are received in this “push model” when an application is listening to an established socket and the other end has sent a message. The entire message must be parsed to identify its payload, and then the payload must be dealt with in some way relevant to the application.

Parsing

Upon arrival, the message is passed to a parser that scans the message looking for the special characters that denote start and endpoints of the message’s lines. Ultimately, the parser will isolate and separate the ASCII and binary payloads from the message by scanning for the special characters that distinguish between the two kinds of payload.

ASCII payload is the text situated between the start of a line ($$) and the end of the line (\r\n). Since this text is XML, it is passed to an XML parser that will read the tags and dispatch their corresponding event handlers. This XML parser would also handle any envelope information once a standard for that is defined.

Binary payload is situated between the special characters $# and \r\n that mark the start and end of the line. The binary length header takes up the first four bytes. The parser uses this information to know how many bytes to read ahead and isolate the original binary chunk. Then the binary handler is dispatched and passed the chunk to process.
**Event Handlers**

Event handlers are functions that are called when certain XML tags are encountered while parsing a message. In the message passing paradigm, events encoded as XML tags trigger certain actions, and event handlers are the mechanisms that perform these actions.

Event handlers are user-defined and application-specific. In addition, applications have full control over event handler behavior. The application that parses an incoming message is responsible for assigning which event handlers are associated with which XML tags. Therefore, only the events that are relevant to the application (or that the application is aware of) are dispatched. It is not an error for a message to include a tag for which an application has not assigned an event handler; in this case, the application can either ignore the tag or dispatch some “default” or “generic” event handler.

Furthermore, the application is free to dynamically change event handler assignments such that one tag may be dispatched by different event handlers at different times. The application can also delete any assignments such that previously relevant tags no longer trigger an event handler. All these changes go into effect starting with the next message that is parsed.

**Binary Handler**

Binary handlers are similar in concept to event handlers, in that binary data triggers the handler just like XML tags trigger event handlers. Applications also have control over binary handlers, e.g., making or changing the binary handler assignment.

However, binary handlers differ from event handlers in operation. First, binary handlers process the binary chunk included in a message, whereas event handlers do not deal with binary data (or any other form of external data). In addition, there is only one binary handler assignment per message. While conceptually, each binary chunk is associated with exactly one XML tag in a message, a binary handler in this model is not required to listen specifically for tags; instead, it listens for the presence
of an incoming binary chunk. That is, binary handlers operate independently of what tag the binary chunk corresponds to. This is an important characteristic of the binary handler. This relaxed constraint allows XML tags to trigger event handlers that can dynamically change the binary handler assignment. In effect, this approach provides a hook into the application for external messages to change internal binary handler assignments while still giving the application full authority over the same task.

Hence, binary handlers offer a flexible method of handling binary data, from the point of view of both the application and its counterpart on the other end of the communication channel. The application can easily change the actions performed by the binary handler; in this sense, the application drives the behavior of the binary handler, as is the primary intention. On the other hand, messages coming into the application from the other end can also change the behavior of the binary handler in a controlled manner through the use of XML tags that can direct the binary handler to treat binary data in the manner specific to the tag. Here, the application's counterpart drives the behavior of the binary handler in a passive, non-intrusive way.

Error Handling

Malformed messages (messages that are not in the proper syntax described in Section 3.4) cannot be parsed correctly and therefore generate errors. This protocol calls for outputting meaningful and proper error messages that describe the specific problem before gracefully aborting.

The following are specific cases of when an error occurs:

- A message contains less than two (2) or more than three (3) lines. This is clearly prohibited by the protocol specifications.

- A binary payload is parsed before ASCII payload. Since messages are parsed in the order that they arrived from the network socket, this can only happen when the first line of the message contains binary and the second line contains ASCII, which is prohibited by the protocol specifications.

- The ASCII payload is not in correct XML format according to the XML spec-
ification. This case would result in incorrect behavior since the XML parser would not be able to identify and dispatch tags.

- A problem with the network socket prevents a message from being sent or received in full.

Note that the last case is at the network layer; it is beyond the control of the messaging layer, unlike the other cases that the messaging protocol can deal with.

A simplistic and conservative approach is taken to “gracefully aborting”. If there is an error while parsing a message, there are several ways to proceed. There are two issues to consider here:

1. whether or not the actions performed by the event and binary handler up until the error should be annulled, acting as if the erroneous message never existed, and

2. afterwards, whether or not to stop operation completely, or recover and wait to receive more messages.

This protocol states that if an error occurs, operation is stopped completely. If the error resulted from a malformed message generated by the other end, then it is futile to continue communications since the other end will continuously send malformed messages. Similarly, if the error resulted from a network error, then nothing can be done to recover other than to close the socket and open a new one. In either case, completely stopping operation is a justified course of action. This specification is a starting point to build upon and maintains simplicity with no method of recovery. More efficient (and complex) approaches exist, but investigation into these is beyond the scope of this thesis.
Chapter 4

C++ API

This messaging protocol has been implemented in C++ under the Linux operating system for both Intel x86 and StrongARM architectures. The C++ implementation defines three major classes that represent the major components of a message passing system:

1. Tags: an internal representation of all XML tags, needed to write and read messages

2. Message writer: this component handles all the functions described in Section 3.5.1 to compose and write messages

3. Message reader: similarly, this handles everything described in Section 3.5.2 to read, parse, and dispatch messages

These classes are named Tag, MsgWriter, and MsgReader, respectively. Their descriptions, function prototypes, and member variables are given below. Refer to Appendices A, B, and C, respectively, for a full listing of code.

4.1 Tag Class

The Tag class is an interface which all XML tags must implement. Hence, each XML tag is a separate class. Since XML tags have different attributes, Tag defines a general structure that derived XML tags customize with their specific data.
Tag contains the name of the tag, and a virtual function prototype that writes the tag in well-formed XML format. Currently, Tag can only support single-tag XML format (e.g., `<TAG/>`, not `<TAG></TAG>`). We also use the convention that XML tags are all uppercase, while their corresponding C++ classes are not.

Public members of Tag include:

- **Tag(char *name)**
  Constructor that must be initialized with a name for the tag. Allocates memory for variable _name to copy the value of name.

- **virtual ~Tag()**
  Destructor that frees the memory held by _name. All derived classes can also implement their own destructors but should also call this base destructor to ensure that _name is freed.

- **virtual int printOut(int fd) const**
  This function writes the tag into XML format to the given file descriptor fd. const is in the declaration because printOut can also be used for const Tag objects.

There is one protected member that is inherited by all derived classes:

- **char * _name**
  Holds a copy of the tag name. Dynamically allocated when Tag is instantiated, and deallocated when the Tag object is destroyed.

### 4.1.1 GenericTag Class

The library also defines a GenericTag class that, as the name suggests, represents any generic tag. It is derived from the parent Tag class, and holds additional private variables that manage the attribute-value pairs for the tag:

- **char **keys**
  Array of the strings for all the keys (attributes) for the generic tag.
- `char **values`
  Array of the strings for all the corresponding values, stored in the same order as the attributes.

- `int _numAttr`
  Internal counter of the number of attribute-value pairs in the generic tag.

### 4.1.2 Envelope Class

The `ENVELOPE` tag is a place-holder since a standard has not been defined for it yet. Nevertheless, the class holds private variables for the name ("ENVELOPE") and the reserved keywords FROM, TO, MSGID, and SESSID:

- `char *_from`
- `char *_to`
- `int _msgid`
- `int _sessid`

_from and _to are dynamically allocated by the constructor when an Envelope object is instantiated, and deallocated by the destructor when that object is destroyed.

The Envelope class also has an additional method printClose() to generate the closing XML tag `</ENVELOPE>`. This is the only tag that can support open-close tag format; all other tag classes are in single-tag XML format.

- `int printClose(int fd) const`
  Takes the same file descriptor fd as printOut(), and outputs the closing ENVELOPE tag.

### 4.2 MsgWriter Class

The MsgWriter class implements the payload queue (Section 3.5.1) that buffers tags to be written for a message. When tags are queued, MsgWriter holds them in queue
until one of three conditions: (1) the number of tags exceeds some predefined limit, (2) the application explicitly forces the message to be written, and, (3) a binary chunk is written, triggering the end of a message. Also, while tags can be added to the queue, they cannot be removed from the queue.

**MsgWriter** maintains an internal boolean variable \_init that indicates whether or not the object has been “initialized” with a file descriptor _fd. None of the functions in **MsgWriter** will run on the object until _fd has been set.

The following are public members of **MsgWriter**:

- **MsgWriter()**
  A dummy constructor. Since no file descriptor is passed in, the **MsgWriter** object is left in an uninitialized state (_init=false). This allows a program to declare a **MsgWriter** object but pass it a file descriptor at a later time.

- **MsgWriter(int fd)**
  The actual constructor that takes in a file descriptor and sets all internal variables and counters, including _init=true, thus initializing the object.

- **~MsgWriter()**
  The destructor will write out a message if the object is destroyed while there are still tags in queue.

- **void LateInit(int fd)**
  Performs a late initialization, as described above, by passing in a file descriptor. This is conceptually the same thing as the constructor **MsgWriter(int fd)**, except that initialization is done after the object is declared in code.

- **int writeTag(Tag *t)**
  Queues a tag in the buffer. A pointer to a Tag is passed in as an argument, so the object must still exist by the time the message written, otherwise the program will crash. Usually, t is created using the new operator as it is passed
in. Once the message is written, t will be destroyed with the delete operator as the payload queue is flushed.

- **int writeBinary(unsigned char *data, unsigned int len)**
  Attach a binary chunk data to the message, of length len bytes. This will cause the message to be written and sent immediately, even if the maximum tag limit per message has not been reached. If there are no queued XML tags when this function is called, an error is signalled, and the binary chunk is not inserted into the message payload. Completed messages are written to the file descriptor _fd that was set when MsgWriter was initialized. A generic envelope is generated for the ASCII payload; support for envelopes will be added in the future.

- **int endMsg(Envelope *e)**
  Manually force the message to be written. Takes a pointer to an Envelope object, which wraps around the XML tags in the ASCII payload. Again, e must still exist when the message is written, hence, use the convention of passing in a new Envelope object, which is destroyed when the message is written and sent to _fd. This method does not do anything if there are no tags in queued.

The private members of MsgWriter are not visible to the application programmer, since they reveal internal state:

- **bool _init**
  A boolean flag indicating whether or not the MsgWriter object has been initialized (i.e., _fd has been set). None of the MsgWriter methods will run unless _init is set to true.

- **int _fd**
  The file descriptor that the MsgWriter object will write to when it sends messages. Initially set to -1. Only values greater than or equal to zero have any meaning.

- **int _numTags**
  The number of tags currently queued for the current message. This counter is
incremented when a `writeTag()` is issued. It cannot decrement since there is no way to remove a tag from the queue. It is initially set to 0, and is reset to 0 when the payload queue is flushed (i.e., after the message is sent).

- **Tag *tg[MAXTAGS]**
  This array of Tag objects is the C++ implementation of the payload queue. It is created with a maximum size of MAXTAGS, a pre-defined value of 5. When `_numTags` reaches the value of MAXTAGS, the message is written out and sent to `_fd`, then `_numTags` is reset to 0. MAXTAGS is hard-coded into the source code, so it currently cannot be changed without having to recompile the library.

- **int writeEOM()**
  Generates the end-of-message line (Section 3.4) and outputs to `_fd`. This method is called when a message is written.

- **int sendASCIIMsg(Envelope *e)**
  Generates the first line of a message using the tags in the payload queue. Takes each Tag object in queue and generates XML for each by calling their respective `printOut()` methods. Also wraps the ASCII payloads with the appropriate opening and closing tags of e. `endMsg()` and `writeBinary()` call this method to generate the first line of the message before they generate the rest.

- **void destroyTagArray()**
  Destroys each Tag object in the payload queue and resets `_numTags` to 0. This method is only called by `sendASCIIMsg()`.

### 4.3 MsgReader Class

The MsgReader class implements the message parser and mechanisms responsible for managing event handlers and binary handlers. These are explained briefly before the API is presented.
4.3.1 Parser

Since messages come in over the network through a TCP socket, messages may arrive piece-meal rather than all at once, particularly long messages that exceed TCP packet length. The message parser is stream-oriented so that it parses the message as it arrives; as the parser recognizes parts of the message, it will call the appropriate handlers that manage that part of the message. For example, if a three-line message (with both ASCII and binary payload) arrives piece-meal, the message parser will not parse ASCII payload until all of the first line has arrived; similarly, the binary handled until all of the second line has arrived.

When the ASCII payload has been parsed and isolated, it is sent to a separate XML parser that reads XML tags. The **MsgReader** class uses the expat XML Parser Toolkit [11] written by James Clark to parse the XML tags in the ASCII payload. expat is also a stream-oriented parser that calls registered handlers when it receives opening and closing tags. When it has parsed an opening tag, it passes control to the event handler mechanism.

4.3.2 Event Handlers and Binary Handlers

There are three types of handlers in this implementation: tag handlers, binary handlers, and error handlers. They are aliased using **typedef**:

- **typedef void (**TAGHANDLER)(void *userdata, void *userdata2, const char *name, const char **attr)**

  A tag handler is called when an XML tag is parsed. It takes four arguments: two for user-defined data (**userdata** and **userdata2**), the name of the tag **name**, and a null-terminated array of key-value pairs **attr**. Key-value pairs are stored in sequential entries in **attr** (that is, keys are in even array indices, values are in odd array indices). Also, it is common for an application to pass the **this** pointer to a tag handler as **userdata** so that the handler will have access to the object’s member functions and variables.
typedef void (*BINHANDLER)(void *userdata, unsigned char *data, int len)

The binary handler processes a binary chunk data of length len bytes. It also takes in as its first argument user-defined data userdata, which, like in the case of the tag handler, is usually the application's this pointer so that the handler can access member functions and variables. len is provided for the handler to grab the correct number of bytes past the memory location pointed at by data. Note that len can also be zero; the handler should treat this correctly as well.

typedef int (*ERRHANDLER)(void)

The error handler is not very sophisticated in this implementation. It is simply a function with no arguments passed in, called whenever some error has occurred while parsing the message. The error handler has room for improvement in any future implementations.

Handlers are functions whose code is provided by the application using the messaging protocol. Handlers must be registered in MsgReader, otherwise MsgReader will not know when to call the handlers. Tag handlers are registered for each particular XML tag (referenced by tag name in all capital letters). There is only one binary handler in a MsgReader object, yet it still needs to be registered. Re-registering a different function as the binary handler is equivalent to changing the binary handler assignment. Similarly, there is only one error handler as well, and it is registered in the same fashion that the binary handler.

In particular, tag handler assignments are stored in a hash table, otherwise known as a map in C++ terminology. The C++ Standard Template Library (STL) provides a map data structure that is used here to map a tag name to a pointer to a tag handler. When an XML tag is parsed, its name is searched in the map to find the appropriate tag handler to execute.
4.3.3 Member Functions and Variables

The relevant private members are presented first (other private members are omitted, see Appendix C for the full listing of code):

- **bool _init**
  Boolean to indicate whether the `MsgReader` has been initialized with a file descriptor. Member functions will not run unless _init=true.

- **int _fd**
  The file descriptor from which the `MsgReader` object will receive incoming messages.

- **XML_Parser _p**
  Pointer to an instance of the expat XML parser.

- **bool _pActive**
  Boolean indicating if the XML parser is active (whether or not an instance of the expat parser has been initialized). XML cannot be parsed if _pActive=false. _pActive is set to true internally when the XML parser is needed, and set to false when the XML parser has finished parsing the ASCII payload.

- **map<string, TAGHANDLER> _tagHndl**
  The C++ STL hash table that maps the names of XML tags to pointers to tag handlers. It is managed internally and automatically when tag handlers are registered.

- **TAGHANDLER _defaultHndl**
  Pointer to the default tag handler. If an XML tag is parsed and does not have a registered tag handler listed in the map, the default tag handler is called instead. If no default handler is registered, the XML tag is ignored.

- **ERRHANDLER _errHndl**
  Pointer to the error handler, if any is registered. Whenever an error occurs during parsing, this handler is called, if one exists.
• BINHANDLER _binHndl
  Pointer to the binary handler. This is called when binary payload is parsed. If
  the binary handler has not been registered, the binary chunk is ignored.

• bool _defHndlSet, _errHndlSet, _binHndlSet
  Boolean variables indicating whether the default tag handler, the error handler,
  and binary handler have been registered, respectively.

• void *userData
  Analogous to the user data used by event and binary handlers. The application
  should register the this pointer to MsgReader, which it passes to the event and
  binary handlers.

The public members are methods that are accessible to the application:

• MsgReader()
  A dummy constructor that sets _init to false. Like MsgWriter, the MsgReader
  class must be initialized with a file descriptor or else none of the member func-
  tions will run. This constructor allows late initialization of a MsgReader object
  after it has been declared.

• MsgReader(int fd)
  Constructor that takes file descriptor fd and initializes the MsgReader object.
  Sets all internal variables, and sets _init to true.

• ~MsgReader()
  This destructor frees up memory taken up by the XML parser if the parser is
  active before a MsgReader is destroyed.

• void LateInit(int fd)
  Initializes the MsgReader object after declaration. Analogous to the LateInit() in
  MsgWriter.

• void RegisterHandler(char *tagname, TAGHANDLER th)
  Assign tag handler th to an XML tag with name tagname. Adds the appropriate
entry to the _tagHndl map; if an entry for tagname already exists, it is replaced with this new assignment.

- void RegisterDefaultHandler(TAGHANDLER th)
  Registers th as the default tag handler and sets the _defHndlSet flag to true.

- void RegisterErrorHandler(ERRHANDLER eh)
  Registers eh as the error handler and sets the _errHndlSet flag to true.

- void RegisterBinaryHandler(BINHANDLER bh)
  Registers bh as the binary handler and sets the _binHndlSet flag to true.

- void RegisterUserData(void *userdata)
  Sets userdata as the user-defined data that is passed to event and binary handlers. When registering, the application should ensure that userdata is not destroyed before the MsgReader object is destroyed, otherwise the program will crash.

- int parse(void)
  Reads data from _fd and parses as much of it as possible. Usually this is called when there is data available on _fd. This function will parse XML tags and call the corresponding tag handlers (or the default tag handler); it will also parse binary payload and pass the binary chunk to the binary handler. Returns -1 on an error, 0 if the file descriptor has been closed, and 1 if parse() expects more data to parse (i.e., only a partial message has been received).

4.4 Sample Usage

The following example shows how programs A and B can use the API to communicate with each other using messages. For simplicity, the example will illustrate one-way communication from A to B. Since the API uses a symmetric MsgReader and MsgWriter pair, two-way communication can be achieved by adding reciprocal code to both A and B, effectively reversing the roles of A and B.
4.4.1 Initialization

Since A sends messages and B receives messages, A will use the `MsgWriter` class and B will use the `MsgReader` class. Assuming A and B have established an asynchronous TCP connection on socket `fd`, A initializes a `MsgWriter` object by declaring:

```
MsgWriter mw(fd);
```

and B initializes a `MsgReader` object by declaring:

```
MsgReader mr(fd);
```

Note that initializing a `MsgWriter` or `MsgReader` object by passing `fd` as an argument is equivalent to declaring the object and performing a late initialization:

```
MsgWriter mw; // same for MsgReader, too
mw.LateInit(fd);
```

4.4.2 Sending a Message

To send a message, A must compose the message. In this example, A will send a tag called “SENDDATA” with some binary payload. Assume that a `SendData` class exists for the SENDDATA tag. The following code snippet will queue the tag and send a message with a chunk of data:

```
unsigned char chunk[] = "binary chunk";
...
unsigned int len = sizeof(char) * strlen(chunk);
mw.writeTag(new SendData());
mw.writeBinary(chunk, len);
```

With an asynchronous TCP connection, the message will be sent when the socket is available for sending data. Meanwhile, the application can continue execution.

4.4.3 Receiving the Message

In order to receive the message correctly, B must register a handler for the SENDDATA tag, as well as register the binary handler:
// Binary handler definition (type BINHANDLER)
void take-binary(void *userdata, unsigned char *chunk,
                 unsigned int len) {
    printf("\'Xs\' is %d bytes long\n", chunk, len);
}

// SENDDATA tag handler (type TAGHANDLER)
void senddata_handler(void *userdata, void *userdata2,
                       const char *name, const char **attr) {
    printf("Received the %s tag\n", name);
}

... mr.RegisterHandler("SENDDATA", senddata_handler);
mr.RegisterBinaryHandler(take_binary);

The application should be responsible for calling parse() when data has arrived
on the file descriptor. As long as there is data, continually calling parse() will
correctly parse incoming messages. The application can also check the return value
from parse() to get the status of the parse:

// this function is called whenever data is available on the
// file descriptor
void read_from_socket() {
    int status = mr.parseo;
    if (a == -1) {
        // network error, call the appropriate error-handler
    }
    // other values of a (0 and 1) are okay
}

When B receives the message from A, it should call senddata_handler() after
parsing the ASCII payload, and then call take_binary() after parsing the binary
chunk. When the message is fully parsed, B should output the following to the
console after running both handlers:

Received the SENDDATA tag
'binary chunk' is 12 bytes long

53
Chapter 5

Implementation Issues

Some issues and concerns that arose while writing the C++ implementation are brought to attention here. Relevant details about the implementation are also explained here.

5.1 Tags

Implementing tags as separate classes that derive from an interface Tag illustrates well the idea of inheritance in an object-oriented programming language like C++. However, the current usage of tags does not exhibit any sort of hierarchical behavior, hence the inheritance tree is very flat (large breadth but very little depth), as shown in Figure 5-1. It would seem like a waste of memory and overhead to represent tags as individual classes, especially if a large number of tags were defined in the system.

On the other hand, tags can be organized in a hierarchical fashion to reflect what function each tag represents. In this manner, the hierarchy provides the context for each tag. For example, future applications may devote one set of tags for messages relevant to speech and another set of tags relevant to vision. In that case, the inheritance tree (Figure 5-2) would have larger depth, with all tags deriving from either a SpeechTag or VisionTag, both of which derive from the interface Tag. Other examples like this could be derived.

In short, we retain the use of inheritance for representing message tags because
it allows for flexibility when application developers must decide how to use message
tags in the best way for their programs.

5.2 Parser

The stream-based parser reads its input one byte at a time when it is presented with
data. The algorithm to parse input is not complicated, and due to the straightforward
formatting of messages, input can be parsed in one pass with the use of a buffer to
hold partial message fragments.

The parser is a finite state machine that has seven (7) states. Each state represents
the current “stage” of parsing; state transitions occur based on the next input char-
acter, as illustrated in Figure 5-3. Each state and its possible transitions is described
below:

**LIMBO** The state when nothing useful has been parsed, i.e., when a complete mes-
sage has been parsed, and the parser is waiting for another message to arrive
Figure 5-3: High-level diagram for the state machine for the message parser. If errors occur during parsing, the state transitions back to LIMBO (these transitions are not pictured here for simplicity).
(the parser is “in limbo”). This is the parser’s initial state.

On ‘$’, state transitions to DOLLAR; otherwise, remains in LIMBO.

**DOLLAR** The state when a single dollar sign has been parsed, and the parser needs another character to determine the special two-character delimiter at the beginning of each line of a message.

On ‘$’, state transitions to TEXT; on ‘#’, transitions to BINREADLEN; otherwise, reverts back to LIMBO.

**TEXT** The state when two consecutive dollar signs have been parsed, indicating that the rest of the current line will contain text. More input is needed, however, to determine whether the line contains real ASCII payload or the special two-character delimiter marking the end of a line. This state can only be entered from the DOLLAR state.

On the special ‘\r\n’ delimiter, state transitions to EOM; otherwise, transitions to ASCII.

**ASCII** The state when the parser is currently receiving the ASCII payload. The parser remains in this state, accumulating the payload, until the \r\n delimiters are parsed. This state can only be entered from the TEXT state.

On the special ‘\r\n’ delimiter, state transitions to LIMBO; otherwise, remains in ASCII.

**BINREADLEN** The state when the parser reads the four-byte binary length header of the binary payload. This state can only be entered from the DOLLAR state.

This state automatically transitions to BINREADDRDATA.

**BINREADDRDATA** The state immediately following BINREADLEN. The parser uses the length obtained in BINREADLEN to read the binary chunk.

This state automatically transitions to LIMBO.
**EOM** The state when an end-of-message line has been parsed, indicating the end of the current message. Any bookkeeping mechanisms for parsing the current message are reset.

This state automatically transitions to LIMBO.

Other events happen during certain states or state transitions. For example, while in the ASCII state, the parser passes any ASCII payload to the XML parser for dispatch, which also runs the registered tag handlers. Also, when the parser moves from state BINREADDATA to LIMBO, it passes the binary chunk to the registered binary handler.

### 5.2.1 Parse Buffer

When data are available and the `MsgReader.parse()` function is called, data are read off the registered file descriptor `_fd` and stored in a parse buffer, which is the actual input to the parser. The parser scans characters from the parse buffer and takes appropriate action as certain characters are read.

The parse buffer behaves like a circular buffer in that data are buffered sequentially until the very end of the buffer, when it loops back and stores data at the beginning. The parser ensures that it has scanned the characters at the beginning of the buffer before overwriting them with new data.

Having a parse buffer enables the parser to perform “look-ahead” operations necessary for: (1) scanning for \r\n delimiters, (2) reading in ASCII payload, (3) reading the binary length header, and (4) reading the binary chunk. If, at any point, more characters are required than are currently in the parse buffer, the parser suspends operation, and resumes when the parser is called again with more data from the TCP socket. For example, if the parser is in state BINREADLEN where it expects to read 4 bytes, but only has 2 bytes available in the parse buffer, it suspends operation until more data is read from the network into the buffer, then resumes if 4 bytes are available. The parser maintains any bookkeeping variables and mechanisms that ensure integrity of the parse buffer (i.e., which parts of the buffer have been read and can be
5.3 Handler Registration

Tag handlers are registered by assigning them to specific tags. Rather than associating them with a specific tag class, tag handlers are keyed in the hash table to the tag name (a character string data type, char *).

The reasons for this are simple and straightforward. There is no dependency between tag classes and MsgReader; keying tag classes to tag handlers would unnecessarily introduce a dependency since MsgReader would need to have knowledge about a variable number of tags. In addition, the parser only receives the XML output of a tag and cannot perform a reverse translation from XML back to the original tag class. Hence, it would be pointless to key tag classes to tag handlers. Since each tag has a unique case-insensitive string representation of its name, it is more practical to key the tag’s name to a tag handler.

5.3.1 Managing Binary Handlers

Binary handler assignments are made by either the application receiving the message or a message tag whose tag handler calls the API function that sets or modifies the assignment (Section 3.5.2). In the latter case, a tag handler changes the binary handler for the current message instead of the next because ASCII payloads are parsed and dispatched before binary payloads of the same message.

The process of having an XML tag set the binary handler is somewhat roundabout in the C++ implementation. The application receiving messages must define an auxiliary function that changes the binary handler assignment for the message (using the API routine RegisterBinaryHandler()). The tag handler assigned to the XML tag must call this function at some point in its code. Therefore, for a message tag to change the binary handler, control must pass through three stages: from the parser to the tag handler to the auxiliary function. Alternatively, the tag handler can directly make the API calls and eliminate the auxiliary function.
5.4 User-defined Data

The `MsgReader` class allows registration of user-defined data. User data is not read or manipulated by `MsgReader` itself, but is passed as an argument to the event handlers which may or may not use it.

A common case when user data is needed in C++ is to pass in an object, so that event handlers have access to member functions and variables of that object. This approach is needed to deal with binary handler functions that are also member functions of a user-defined class. Although function `bar()` may be a member of class `foo`, when it is stored as a binary handler assignment (that is, a pointer to the function), `bar()` cannot access members of that instance of `foo` due to limitations of the C++ language. Instead, `foo.bar()` should be given a pointer to the `foo` object in order to access member functions and variables. It is also necessary for the object, passed in as type `void *`, to be explicitly cast to data type `foo` with the `reinterpret_cast` C++ operator.
Chapter 6

Evaluation

Since the messaging protocol is a mechanism to facilitate network communication, there is no quantifiable metric by which to judge success. Instead, we tried simple tasks and scenarios to debug the implementation as it was being developed and evaluate the effectiveness of the protocol in performing the tasks. These experiments and their results are described below.

6.1 Test Programs

Small programs were written to demonstrate the capabilities of the messaging protocol using the C++ implementation. They exercised the use of both the ASCII and binary payloads in messages. The informal test plan defined two sets of test applications. The first set was used primarily to test and debug the protocol library; they mimicked “instant messaging” programs by sending text messages that the user typed from a client to a server using the messaging protocol as the transport carrier. The second set, which is not complete due to time constraints, was intended to be a set of multimedia applications that would rely heavily on binary payloads to send audio and images. The test application to send images has not been written.
6.1.1 Text Messaging, Version 1

This application allows one user to connect and send text messages to another user. To avoid confusion, we use the terms “text messages” and “protocol messages” to differentiate between what the user types and what are sent by the messaging protocol, respectively. The application was developed for only one-way communication, but demonstrates the ability to embed the text within the ASCII payload of a protocol message.

There are two components: the client and the server. User A starts the client that connects to User B’s server through an asynchronous TCP connection. User A can then type and send one-lined text messages (text is not sent until User A presses the RETURN key) to User B, who receives that text message shortly after. Both the client and server interfaces run off the console without a GUI.

Text messages are embedded as attributes in a special XML tag called TEXTMSG (an abbreviation for “text message”). Only one text message can be stored in one tag. For example, when User A types “Hello, User B” and hits the RETURN key, the following XML tag is generated:

```
<TEXTMSG TEXT="Hello, User B"/>
```

The tag is immediately packaged into a protocol message and sent to User B. The server registered a tag handler to parse incoming TEXTMSG tags and output the text message to the screen. One protocol message contains exactly one TEXTMSG tag to ensure text messages are sent instantaneously, otherwise text messages would be waiting in the payload queue before being sent across the network at some later time.

This application performed as expected, enabling reliable and instantaneous one-way text communication between two users. Two-way communication was achieved by running both a client and a server on each user’s terminal and directing two instances of one-way conversation each on a separate TCP connection.

There was a limitation, however, on what a user could type: text messages could not include the double quotation mark since it is used for attribute encodings in
XML, and caused malformed XML tags that stopped the XML parser. In order to use quotes, they must be replaced with `&quot;`, which is translated to a double quotation mark by the XML parser. This would require some additional code to scan the text message and modify it prior to writing out the protocol message.

### 6.1.2 Text Messaging, Version 2

Rather than analyzing what the user types and modifying the input appropriately, we choose another approach to deal with this limitation. Instead, text messages are moved to the binary payload of protocol messages in version 2 of the application. Embedding the text message in the binary payload is a better option for sending text messages for the following reasons: 1) Binary data are not processed, effectively removing any limitation on what characters the text messages can and cannot contain, and, 2) protocol messages inherently contain only one binary chunk (and hence one text message per protocol message), making it well-suited for transmitting text messages instantaneously.

In version 2, the TEXTMSG tag is stripped to a simple placeholder, absent of any name-attribute pairs. This is to keep the protocol message within specifications, since a message with binary payload must contain some ASCII payload. In addition, the server no longer needs to register a tag handler for TEXTMSG because it is now the binary handler’s job to read the text message and output it to the screen.

The client for version 2 also features a GUI written using GTK+, the GIMP Toolkit [23]. The server, on the other hand, remains console-based and not quite as “aesthetically-pleasing” as the client. We took advantage of the Entry widget in GTK+ to enable a user to type longer, multi-lined text messages and send them at the push of a graphical button rather than with the RETURN key. Fundamentally, however, the primary difference between version 1 and version 2 is that in version 2, text messages are transported in the binary payload.

This version behaved exactly like version 1, with the added benefit that users could send text messages that included double quotation marks. Again, a client-server pair could only support one-way communication, but two-way communication
was achieved by loading both a server and a client for each user.

Version 2 exposes the issue that applications should transport any data that needs to remain unmodified in the binary payload of the protocol message. The ASCII portion is capable of sending any data subject to the restrictions of the implementation’s handling of the XML format. ASCII payload is better suited for encoding system events or any other piece of information for which formatting may be modified as fit.

The implementation lacks support for XML elements with start-tags and end-tags that enclose a CDATA field. Text messages with double quotation marks could be stored as CDATA. For example, for a user typing the text message I said "Hello, User B", the following XML element could be generated with start-tag and end-tag and placed in the ASCII payload (if it were supported):

\[
<TEXTMSG>I said "Hello, User B"</TEXTMSG>
\]

rather than placed in binary payload. However, this is also not a solution because the CDATA text cannot contain a pair less-than (\(<\)) and greater-than (\(>\)) symbols, which could be misinterpreted as another tag or even the end-tag. If User A types "<oops> this is wrong", then the following XML tag is generated:

\[
<TEXTMSG><oops> this is wrong</TEXTMSG>
\]

Here, "<oops>" is interpreted as a new start-tag when it is really part of the text message. The less-than and greater-than symbols can be substituted with \&lt; and \&gt;, respectively, to satisfy the XML parser, but we would again return to the problem of having to scan user input. There are also other characters that have to be translated to parse correctly, and this would be a tedious process.

In general, most applications using the current implementation of the messaging protocol should ship any form of data as binary payload (even if the content is ASCII), since binary imposes no restrictions on content and uses very little overhead above that of sending ASCII payload.
6.1.3 Text Messaging, Version 3

This version revisits text messaging to address the issue of scalability. It lets multiple clients connect to one server and send text messages to it. The server software was modified to accept multiple TCP connections, starting a separate instance of a `MsgReader` object for each client establishing a connection. Visually, the server still outputs each incoming text message on the screen, but with a number prepending each text message to identify its origin (the server assigned clients their number in the order that they established a connection, and does not reuse numbers for clients that have disconnected).

There was no formal or rigorous testing for this, since results can only be qualified, not quantified, although more attention was focused toward comprehensive testing with this application. It is reasonable to expect that the server should be able to handle a fairly large number of clients as long as there is enough physical memory to hold the `MsgReader` objects and any other classes used to keep track of connections.

A varying number of clients were connected to a server in over 20 separate trials. The number of clients connected ranged over time from 0 to approximately 50. A handful of volunteers started an arbitrary number of clients that connected to server on a host machine and proceeded to send messages “at will”. All the messages were received by the server (as expected, since TCP is a reliable protocol) and were intact (also as expected, since the messaging protocol does not alter anything in its binary payload).

6.1.4 Galaudio/XMLaudio

Galaudio (short for “Galaxy audio”) is a client application that lets users connect to and interact with SLS spoken language systems. It is existing software that is used on the Compaq iPAQ computers to connect to the Jupiter weather information system. The user speaks into the microphone to query the Jupiter system, and the Jupiter system sends a response back in the form of a computer-generated voice that is played over the iPAQ speaker.
Galaudio was modified to send the user’s voice recording via the messaging protocol. The new application was called XMLaudio (to indicate that it is “XML messaging protocol audio”). Because augmenting the Jupiter code to use the messaging protocol was too large a task, a separate and much simpler server (called XMLserver) was written to accept connections from XMLaudio and play back any sound samples it received from XMLaudio through the host machine’s speakers. This setup essentially implemented a one-way intercom from XMLaudio to XMLserver.

XMLaudio was cross-compiled to run on both the StrongARM architecture of a Compaq iPAQ and the native x86 architecture of a desktop PC. Users can connect to the desktop from the iPAQ and record their voice for a short period of time, which is streamed across the network to the desktop and played through its speakers. Audio plays fine on the desktop (i.e., no skips or distortion), which indicates that the audio, sent as multiple binary chunks, is being sent successfully via protocol messages.

The purpose of writing XMLaudio was to demonstrate that the messaging protocol is capable of sending actual binary data (in this case, digitized sound recordings), and not just ASCII text messages in the binary payload. Furthermore, it was the first time that the messaging protocol was used in a GALAXY application, since both XMLaudio and XMLserver used the same code as Galaudio to set up audio devices for real-time playback and recording. Given the success of this experiment, it is hoped that the messaging protocol will be used in future GALAXY applications as well.

6.1.5 Other Programs

The programs that were never written that were to use the messaging protocol are mentioned here because they have direct application towards multimodal computing. The messaging protocol has yet to be tested in an environment that sends images and video. Although images and video, from the point of view of the messaging protocol, are simply raw bytes in a data stream, empirical evidence should be gathered to verify that the messaging protocol is capable of handling transport of the data for multimedia and multimodal applications.

Time should be invested in writing applications that can take images or videos
and stream it to other destinations using the messaging protocol. Successful trials in these areas will further demonstrate that the messaging protocol would be well-suited for a multimodal GALAXY environment.

6.2 Observations

From the performance of these applications, several points are observed with regards to both the protocol specifications and the protocol implementation.

First, the specification states that messages cannot contain just binary payload. Version 2 of the text messaging client showed that this sort of application relies solely on shipping data in the binary payload, making the ASCII payload unnecessary. In fact, the TEXTMSG tag was retained just to meet specifications, even though it was a dummy tag that had no function at all. It is worth considering relaxing the specification by allowing messages to contain only binary payload, not just for this particular application, but also for other applications that may stream data (e.g., audio, video data). For such tasks, the bulk of the messages will contain primarily binary data, and adding extra overhead by introducing non-essential ASCII payload would be wasteful. On the other hand, other forms of streaming applications usually have some non-binary information associated with the binary data that is best encoded using XML tags. In these cases, binary data should require ASCII payload. Both sides of the argument should be examined in closer detail in the future.

Second, inclusion of the messaging protocol API in the code for these applications was relatively easy and unobtrusive. There were a modest number of functions calls made throughout the main programs, but the bulk of the API calls were made as the MsgWriter or MsgReader objects were initialized. We would conjecture that integration of the API into existing applications would not be a difficult task, especially if the program follows an asynchronous communication model like the API does.

Also, the implementation is moderately robust, but could still be improved in future revisions. Versions 1 and 2 of the text messaging applications were developed as the implementation was written, so whenever the programs crashed due to a bug
in the API, the code was fixed to resolve the specific problem. Many bugs in the code surfaced as more and more trials of the rest of the test applications were conducted; this significantly improved reliability. Error-handling in the implementation lacks sophistication at this point, such that any error will only result in an ambiguous error message on the screen followed by a graceful (if not abrupt) exit. This is also something to improve upon in future implementations.

Lastly, all these applications were run in Linux on both Intel x86 and StrongARM architectures. The messaging protocol library has been successfully compiled and run on x86 desktop and laptop PCs as well as the StrongARM Compaq iPAQ handheld PCs. The iPAQs are currently used to run multimodal applications in the MIT Oxygen Project, and is an important target system for supporting GALAXY applications. The ability to write a messaging protocol library on an iPAQ shows promise for its use in the GALAXY framework.
Chapter 7

Conclusion

7.1 Discussion

7.1.1 Current State

The C++ implementation of the messaging protocol offers basic functionality, and does not offer any fancy features or "bells and whistles". Due to time constraints, some of the error handling features have not been fully implemented, and remains on the "to do" list for future versions of the C++ implementation (an application will still gracefully exit on a fatal error rather than abruptly crashing).

The protocol specifications went through very little change as the implementation was written. Some of the encoding formats are superfluous, verbose, or redundant as a result of a conservative approach to designing the specifications (Section 6.2 considers revising the encoding to allow for messages to contain only binary without accompanying ASCII payload). It is expected that revisions or additions will be made as other issues arise. In other words, the current specifications are not final.

7.1.2 Comparison with Frame Relay

Both the messaging protocol and the GALAXY frame relay overlap in function and purpose. Both employ very similar message-passing interfaces whose messages share many "cosmetic" similarities. For example, both use messages that are delimited by
special ASCII character sequences (in fact, the use of the dollar sign to begin messages originated from MIT’s John Ankorn’s early attempts at writing external applications that interfaced with GALAXY before either the frame relay or the messaging protocol existed).

The frame relay is being used for a variety of current GALAXY projects while the messaging protocol is still being refined. The frame relay is only a temporary measure until the messaging protocol is ready to be deployed. As such, the frame relay does not provide as much flexibility as the messaging protocol. The frame relay lacks the ability to register application-specific functions as event handlers like in the messaging protocol. Instead, the frame relay uses specific commands in messages to perform GALAXY session operations.

In addition, frame relay implements its own form of resource and location management. The messaging protocol assumes some independent and external resource directory is available to use. Relying on an external system to process and map resources and their locations means that messages can still be delivered and received properly even if the resource directory changes (this may be the case for different research environments that develop GALAXY applications).

7.1.3 Use in Pervasive Computing

This messaging protocol is a transport mechanism that ships data between systems and notifies components of events in the GALAXY architecture. The direct application of this is to connect GALAXY and non-GALAXY applications that collectively form a multimodal interface used in a pervasive computing environment. Parallels can be drawn between this setup and that of the Metaglue system used to build intelligent environments.

Metaglue is a specialized language capable of dealing with the computational properties shared by intelligent environments [7]. Such properties include integration and dynamic management of distributed resources, much like how the GALAXY hub manages its resources on a smaller scale. A multimodal interface is just one component of an intelligent environment, yet it is a comparable microcosm in terms of high-level
tasks and organization.

At a lower level lies the inter-process and inter-system communication mechanisms and the control logic responsible for resource management and directory services, all dealt with by Metaglue. In comparison, within the multimodal interface facilitated by the GALAXY architecture, these tasks are handled by the hub, hub scripts, and messaging component (i.e., frame relay and the messaging protocol). In contrast, many of the same set of tasks are merely run on different processes running at various levels of the intelligent environment.

This observation suggests that the messaging passing interface implemented by the messaging protocol could be extended towards other applications in pervasive computing. While not nearly as sophisticated as Metaglue, the messaging protocol might be useful supplementing the Metaglue system or other similar "computational glues" in managing the various subsystems of a complex intelligent environment.

7.2 Future Directions

The work presented in this thesis is a "first try" at implementing a message-passing system suitable for GALAXY applications. While not perfect, it demonstrates the feasibility of using the message-passing paradigm for interfacing GALAXY and non-GALAXY components. With that in mind, we turn attention to future work that explores other features and enhancements for the protocol, as well as plans for integration in existing systems.

7.2.1 Language Bindings

The only implementation available for this protocol was written in C++. Other language bindings should be developed to make this protocol portable across languages and platforms. Other commonly-used languages include C, Java, Perl, Python, and Lisp. Many of the applications developed for the MIT Oxygen Project are written in one of more of these languages (including C++). Therefore, it would be essential to offer bindings for each language if the applications were to interface with GALAXY
systems using the messaging protocol. Language bindings should be developed as the messaging protocol is integrated into the GALAXY software suite, either replacing or coexisting with the frame relay.

7.2.2 More on Envelopes

In Section 3.4.1 we introduced envelopes, which remain an undefined standard, but were adopted in this protocol with the intention of managing GALAXY-specific operations. Envelopes currently reserve the keywords FROM, TO, MSGID, and SESSID.

FROM and TO refer to message sources and destinations. These fields deal with locations, but the specification leaves open the how locations are named. These may be IP addresses (used in current testing plans) or any other form of URI. Another possibility is to use new and experimental naming systems. Separating the tasks of message delivery and network routing (Section 2.5) makes this both possible and practical. The Intentional Naming System (INS) developed at the MIT Laboratory for Computer Science is an innovative naming system that maps descriptions of various resources with their physical network locations [1]. This form of resource discovery and management allows for users to ask for “the closest printer to their computer” while INS takes care of finding the network location (e.g., an IP address) for the user. This sort of application would work well in pervasive computing applications, and in particular, would integrate well with the notion of envelopes in the messaging protocol to deliver messages from one location to another. Regardless of how locations are named, envelopes should leave that to a separate naming system so that in case future naming systems come along, they can be easily integrated into the messaging protocol.

MSGID and SESSID are integer identifiers for messages and sessions, respectively. It is conceivable that session IDs should be embedded in each message so that messages can be distinguished between users who have initiated multiple sessions on one instance of a GALAXY hub. Similarly, message IDs are another way to distinguish between messages when there is high message traffic. TCP guarantees in-order and reliable transmission of packets, so message IDs may be redundant for that purpose.
On the other hand, message IDs can be used instead to trigger certain events on any receivers that are listening for particular message IDs within their own sessions. This feature, in particular, is worth exploring in future implementations of the protocol, once the envelope standard has been defined. Also note that messages are currently sent point-to-point, but if there is a need to globally broadcast messages, message and session IDs (or some variant) will be essential for messages to arrive at the appropriate destinations.

7.2.3 Namespaces

The notion of namespaces was alluded to during the discussion of tag hierarchy in Section 5.1. Namespaces for tags provide a way of organization (not necessarily in a hierarchical fashion) to cluster tags of similar context or function together. It would be similar to how C functions are grouped together by header files. Namespaces also provide isolation so that independently-developed tag schemes can coexist without clashing or conflicting with each other.

There are many approaches to implementing namespaces for tags. It is easy in C++ to utilize the hierarchical nature of classes to implement tags that inherit from parent classes; these parent classes essentially become top-level namespaces for the child tags.

An implementation-independent approach is to adopt a naming convention that uses namespaces in the tag name itself. For example, tag names should be preceded by the namespace followed by a colon (:). Hierarchy is denoted by forward slashes (/) as namespaces traverse deeper down the path. This alternative, as simple as it looks on paper, incorporates namespaces into the actual protocol specification; hence, each language binding must handle whatever namespace convention is used. The tradeoff is that changing the convention would entail significant rewriting of the language binding to handle the new convention.

Namespaces can also extend the protocol to use a library or repository of event handlers organized in a way that reflects the high-level organization of the namespace scheme. In this approach, event handlers are shared by multiple applications, reduc-
ing the amount of code space or overhead ordinarily required to define and register each event handler separately. There are many more extensions and applications for namespaces, which makes them an attractive topic to explore in the future.

### 7.2.4 Integration with Existing Systems and Beyond

A reasonable "next step" for this messaging protocol is to use it in existing systems. The XMLaudio application was an example of trying to integrate the messaging protocol into a GALAXY application. In that case, however, the messaging protocol was not integrated into the Jupiter system. In the future, it would be good to try this to establish a fully operational GALAXY system that interfaces with client applications with the messaging protocol. The other information domain servers are also good candidates for integration.

In particular, the WebGALAXY system is interesting to consider for integration with the messaging protocol. WebGALAXY is an extension of GALAXY that supports spoken user interfaces via a standard Web browser [15]. Moreover, it is a flexible multi-modal user interface system easily accessible via the Web. Revisiting WebGALAXY is an opportunity to explore how to use the messaging protocol to simplify or improve performance of tasks in a multi-modal interface, and possibly lead to development of a new version of WebGALAXY that incorporates these ideas for a more powerful, flexible, and accessible system. With the prevalence of XML in web technologies, it will also be interesting to see if the messaging protocol, with its usage of XML, can play a role in managing the web interface of WebGALAXY.

Recently, the MIT Oxygen Project has looked towards developing applications that can leverage GALAXY systems. Section 2.1.1 briefly mentioned the current paradigm for interfacing external applications with GALAXY, which involves clients establishing connections with a GALAXY system as well as virtual resources (e.g., microphone, speaker). For example, a speech application must acquire a virtual microphone as input, and may acquire multiple virtual speakers for output. Having this distributed design introduces more network traffic since the topology of sources and destinations has grown a significant amount. Now, the recording from the micro-
phone must be broadcast to multiple virtual speakers, and they must be the correct speakers assigned to the host client. The paradigm uses sessions to help manage resource-sharing among multiple clients, but the issue of managing communication among these clients is not addressed. To fill in this gap, we can augment the paradigm with the messaging protocol since it is an ideal mechanism for point-to-point communication and ensures reliable and precise transport of data among resources and clients in the same sessions.

In conclusion, this thesis demonstrated the potential for the messaging protocol to facilitate interaction between components in a multimodal system. There still remains room for this protocol to grow and improve, and the examples of future work discussed above are themselves significant projects. The protocol was developed from the start to be a simple yet flexible mechanism to help systems communicate with each other. As the protocol is still not fully developed, it is hoped that future implementations will better accommodate GALAXY applications while maintaining flexibility and ease-of-use.
Appendix A

Tag Source Code

The class definition and code for Tag:

class Tag {
    public:
        Tag(char *);
        virtual ~Tag();
        virtual int printOut(int) const;
    protected:
        char * _name;
    };

Tag::Tag(char *name) {
    _name=new char [strlen(name)+1];
    strcpy(_name, name);
}

Tag::~Tag() {
    delete [] _name;
    _name=NULL;
}

/* prints tag in XML message form to a specified file descriptor *
* returns non-zero number of bytes written, or -1 if error */
int Tag::printOut(int fd) const {
    // compose char buffer
    int len=3+strlen(_name);
    char *buf=new char [len+1];
    sprintf(buf, "<\%s/>", _name);

    // write to file descriptor
    return write(fd, buf, len);
}
Class definitions and code for GenericTag and Envelope, two classes that derive from the Tag interface:

class GenericTag : public Tag {
public:
    GenericTag(char *, char **);
    ~GenericTag();
    int printOut(int) const;
private:
    char **_keys;
    char **_values;
    int _numAttr;
};

GenericTag::GenericTag(char *name, char **attr) : Tag(name) {
    for (_numAttr=0; attr[2*_numAttr]; _numAttr++);
    _keys=new char* [_numAttr];
    _values=new char* [_numAttr];
    for (int i=0; i<_numAttr; i++) {
        _keys[i]=new char [strlen(attr[2*i])+1];
        _values[i]=new char [strlen(attr[2*i+1])+1];
        strcpy(_keys[i], attr[2*i]);
        strcpy(_values[i], attr[2*i+1]);
    }
}

GenericTag::~GenericTag() {
    for (int i=0; i<_numAttr; i++) {
        delete [] _keys[i];
        delete [] _values[i];
        _keys[i]=_values[i]=NULL;
    }
    delete [] _keys;
    delete [] _values;
    _keys=_values=NULL;

    // call base class' destructor
    this->Tag::~Tag();
}

int GenericTag::printOut(int fd) const {
    int len=3+strlen(_name); // leading "<", tag name, and trailing "/>
    for (int i=0; i<_numAttr; i++) {
        len++;// leading whitespace
    }
}
len += strlen(_keys[i]);
len += 3; // "=" between key and value, plus two quotes around value
len += strlen(_values[i]);
}

char *buf = new char [len+1];
sprintf(buf, "<%s", _name);
for (int i=0; i<_numAttr; i++) {
    sprintf(buf, "%s %s="Xs", buf, _keys[i], _values[i]);
}
strcat(buf, "/”);

return write(fd, buf, len);
}

class Envelope : public Tag {
public:
    Envelope(int, char *, char *, int);
    ~Envelope();
    int printOut(int) const;
    int printClose(int) const;
private:
    int _msgid, _sessid;
    char *to, *from;
};

Envelope::Envelope(int msg, char *to, char *from, int sess) :
    Tag("ENVELOPE") {
    _msgid=msg;
    _sessid=sess;
    _to=new char [strlen(to)+1];
    _from=new char [strlen(from)+1];
    strcpy(_to, to);
    strcpy(_from, from);
}

Envelope::~Envelope() {
    delete [] _to;
    _to=NULL;
    delete [] _from;
    _from=NULL;
    this->Tag::~Tag();
}
int Envelope::printOut(int fd) const {
    char *buf;
    int len=0;

    len=strlen("<ENVELOPE>")+strlen(" MSGID="")+\n    (int)(log10(_msgid)+1)+\n    strlen(" TO=")+strlen(_to)+\n    strlen(" FROM=")+strlen(_from)+\n    strlen(" SESSID=")+(int)(log10(_sessid)+1);

    buf=new char [len+1];
    sprintf(buf,\n            "<ENVELOPE MSGID="\n            TO="\n            FROM="\n            SESSID="",\n            _msgid, _to, _from, _sessid);
    return write(fd, buf, len);
}

int Envelope::printClose(int fd) const {
    char buf[]="/\n    return write(fd, buf, len);
}
Appendix B

MsgWriter Source Code

The full listing of the MsgWriter code:

```cpp
#define MAXTAGS 5  
class MsgWriter {
    public:
    MsgWriter();
    MsgWriter(int);
    ~MsgWriter();
    void LateInit(int);
    int writeTag(Tag *);
    int writeBinary(unsigned char *, unsigned int);
    int writeEOM();
    int endMsg(Envelope *);

    private:
    bool _init; // "initialized"?
    int _fd; // file descriptor
    int _numTags; // current number of tags in message
    Tag * _tg[MAXTAGS]; // queue of tags to write in message
    int sendASCIIMsg(Envelope *);
    void destroyTagArray();
};

MsgWriter::MsgWriter() {
    _init=false;
}

MsgWriter::MsgWriter(int fd) {
```
LateInit(fd);

MsgWriter::~MsgWriter() {
    endMsg(new Envelope(1, "end-of", "msg", 2));
    _fd=-1;
    _init=false;
}

void MsgWriter::LateInit(int fd) {
    _fd=fd;
    _numTags=0;
    _init=true;
}

/*@ queue a tag to be written in the current message */
int MsgWriter::writeTag(Tag *t) {
    if (!_init) return -1;

    // grow Tag array if full
    if (_numTags == MAXTAGS) {
        fprintf(stderr, "* Queue full, packaging message.");
        if (endMsg(new Envelope(13, "receiver", "sender", 4)) < 0)
            return -1;
        _numTags=0;
    }

    _tg[_numTags++]=t;
    return 1;
}

/*@ send a chunk of binary data with the current message */
int MsgWriter::writeBinary(unsigned char *bin, unsigned int len) {
    if (!_init) return -1;

    /* package and send ASCII message if necessary */
    if (_numTags > 0) {
        if (sendASCIIMsg(new Envelope(13, "receiver", "sender", 44)) < 0)
            return -1;
    }

    /* write binary chunk */
    unsigned long int nlen;
    int nlenstrlen;
    unsigned char *nlenstr;

    nlen=htonl((unsigned long int)len);
nlenstrlen = sizeof(nlen) / sizeof(unsigned char);
nlenstr = new unsigned char [nlenstrlen];
memcpy(nlenstr, (unsigned char *)&nlen, nlenstrlen);

int bytesWritten = 0;
int count;
if ((count = write(_fd, "$#", 2)) < 0) return -1;
else bytesWritten += count;
if ((count = write(_fd, nlenstr, nlenstrlen)) < 0) return -1;
else bytesWritten += count;
if (len == 0) {
    if ((count = writeEOM()) < 0) return -1;
    else bytesWritten += count;
} else {
    if ((count = write(_fd, bin, len)) < 0) return -1;
    else bytesWritten += count;
}
delete nlenstr;
nlenstr = NULL;

bytesWritten += writeEOM();
return bytesWritten;

/*@ send the End-of-Message, $$\r\n */
int MsgWriter::writeEOM() {
    return write(_fd, $$\r\n", 4);
}

/*@ flush the queue and send the message */
int MsgWriter::endMsg(Envelope *e) {
    int count = sendASCIIMsg(e);
    if (count <= 0) return count;
    else {
        int count2 = writeEOM();
        if (count2 < 0) return count2;
        else return count + count2;
    }
}

/*@ package the message and send to fd */
int MsgWriter::sendASCIIMsg(Envelope *e) {
    if (!_init) return -1;
    if (_numTags == 0) return 0;

85
int bytesWritten=0;
int count;

/* write out leading $$ and then ENVELOPE tag */
if ((count=write(_fd, "$$", 2)) < 0) return -1;
else bytesWritten += count;
if ((count=e->printOut(_fd)) < 0) return -1;
else bytesWritten += count;

/* write out all tags in queue */
for (int i=0; i<_numTags; i++) {
    if ((count=_tg[i]->printOut(_fd)) < 0) return -1;
    else bytesWritten += count;
}

/* close ENVELOPE tag and add trailing \r\n */
if ((count=e->printClose(_fd)) < 0) return -1;
else bytesWritten += count;
if ((count=write(_fd, "\r\n", 2)) < 0) return -1;
else bytesWritten += count;

destroyTagArray();
delete e;
e=NULL;
return bytesWritten;
}

void MsgWriter::destroyTagArray() {
    for (int i=0; i<_numTags; i++) {
        delete _tg[i];
        _tg[i]=NULL;
    }
}

._numTags=0;
Appendix C

MsgReader Source Code

The full listing of the MsgReader code:

```c
#define READBUFSIZE 200
#define PARSEBUFSIZE 100

enum parsestate {LIMBO, DOLLAR, TEXT, ASCII, BINREADLEN, BINREADDATA, EOM};

/* tag handlers take as arguments:
   void *: the 'this' pointer
   const char *: tag name
   const char **: attribute
*/
typedef void (*TAGHANDLER)(void *, void *, const char *, const char **);
typedef int (*ERRHANDLER)(void);

/* binary handlers take as arguments:
   void *: usually the 'this' pointer for the current object
   unsigned char *: the binary data
   int: length of binary data
*/
typedef void (*BINHANDLER)(void *, unsigned char *, int);
typedef unsigned long int UINT;

class MsgReader {
public:
   MsgReader();
   MsgReader(int, void *);
   ~MsgReader();
   void LateInit(int);
   void RegisterHandler(char *, TAGHANDLER);
   void RegisterDefaultHandler(TAGHANDLER);
   void RegisterErrorHandler(ERRHANDLER);
};
```

87
void RegisterUserData(void *);
void RegisterBinaryHandler(BINHANDLER);
int parse();

private:
    bool _init;
    int _fd;
    parsestate _state;

    char * _readbuf; // buffer reading from _fd
    int _readbufRemain; // number of chars left in buffer at end of parse()
    char * _parsebuf; // buffer to send to parser

    XMLParser _p;
    bool _pActive;

    map<string, TAGHANDLER> _tagHndl;
    TAGHANDLER _defaultHndl;
    ERRHANDLER _errHndl;
    BINHANDLER _binHndl;
    void * _userData;
    bool _defHndlSet, _errHndlSet, _binHndlSet;

    unsigned char _binlen[4]; // store the binary length field
    UINT _binLength;
    unsigned char * _bindat;
    int _binPos; // position of the _bindat buffer

    /* handlers for expat parser */
    static void start_handler(void *, const char *, const char **);
    static void end_handler(void *, const char *);

    void flushbuf(int&, int);
};

MsgReader::MsgReader()
{  
    _init=_pActive=false;
    _defHndlSet=_errHndlSet=_binHndlSet=false;
}

/* initialize MsgReader with a file descriptor and user data */
/* in most cases, user data will be the ‘this’ pointer, which is */
/* passed into the tag handler and binary handlers */
MsgReader::MsgReader(int fd, void *userdata)
{  
    LateInit(fd);
    RegisterUserData(userdata);
}
void MsgReader::LateInit(int fd) {
    // variable initializations
    _fd=fd;
    _state=LIMBO;
    _readbuf=new char [READBUFSIZE];
    _readbufRemain=0;
    _defHndlSet=_errHndlSet=_binHndlSet=false;
    _userData=_bindat=NULL;
    _pActive=false;
    _init=true;
}

void MsgReader::RegisterHandler(char *tagname, TAGHANDLER th) {
    _tagHndl[string(tagname)]=th;
    fprintf(stderr, "* Handler registered for tag \%s\n", tagname);
}

void MsgReader::RegisterErrorHandler(ERRHANDLER eh) {
    _errHndl=eh;
    _errHndlSet=true;
}

void MsgReader::RegisterUserData(void *udata) {
    _userData=udata;
}

void MsgReader::RegisterBinaryHandler(BINHANDLER bh) {
    _binHndl=bh;
    _binHndlSet=true;
}

int MsgReader::parse() {
int i;
int bytesRead, readbufLen;
int parsebufLen=0;
bool quit;

// read enough bytes from network to fill up _readbuf
int readThisMuch=READBUFSIZE-_readbufRemain;
bytesRead=read(_fd, _readbuf+_readbufRemain, readThisMuch);
if (bytesRead == -1) {
    if (_errHndlSet) return _errHndl(); // ERROR!
    else return -1;
}
if (bytesRead == 0) {
    // EOF
    return 0;
}
i=0;
readbufLen=_readbufRemain+bytesRead;
quit=false;

// process what's been read
while (!quit) {
    // initialize the XML parser
    if (!_pActive) {
        _parsebuf=new char [PARSEBUFSIZE];
        _p=XML_ParserCreate(NULL);
        if (!_p) {
            fprintf(stderr, "Couldn’t create parser!\n");
            exit(-1);
        }
        XML_SetElementHandler(_p, this->starthandler, this->endhandler);
        XML_SetUserData(_p, this);
        _pActive=true;
    }
    _readbufRemain=readbufLen-i-1;
    switch(_state) {
    case LIMBO:
        if (_readbuf[i] == '$') {
            _state=DOLLAR;
            i++;
            break;
        } case DOLLAR:
        if (_readbuf[i] == '$') 


_state=TEXT;
else if (_readbuf[i] == '#') {
    _state=BINREADLEN;
}
else
    _state=LIMBO;
i++;
break;
case TEXT:
if (_readbuf[i] == '\r') {
    if (_readbufRemain == 0) { // last character in buffer
        memmove(_readbuf, (char *)&_readbuf[i], 1);
        _readbufRemain++;
        flushbuf(parsebufLen, 0);
        quit=true;
    } else {
        if (_readbuf[i+1] == '\n')
            _state=EOM;
        else
            _state=ASCII;
    }
} else
    _state=ASCII;
break;
case ASCII:
if (_readbuf[i] == '\r') {
    if (_readbufRemain == 0) { // last character in buffer
        memmove(_readbuf, (char *)&_readbuf[i], 1);
        _readbufRemain++;
        flushbuf(parsebufLen, 0);
        quit=true;
    } else {
        if (_readbuf[i+1] == '\n') { // received \r\n
            // flush buffer with finish=1
            flushbuf(parsebufLen, 1);
            //flushbuf(parsebufLen, 0);
            XML_ParserFree(_p);
            delete [] _parsebuf;
            _pActive=false;
            _state=LIMBO;
        } else {
            _parsebuf[parsebufLen++]=_readbuf[i];
            if (parsebufLen == PARSEBUFSIZE) {

        }
flushbuf(parsebufLen, 0);
}
}
i++;
}
}
else { // normal text
_parsebuf[parsebufLen++]=_readbuf[i];
if (parsebufLen == PARSEBUFSIZE) {
    flushbuf(parsebufLen, 0);
}
i++;
}
break;
case BINREADLEN:
    if (_readbufRemain < 3) {
        memmove(_readbuf, (char *)&_readbuf[i], _readbufRemain+1);
        _readbufRemain++;
        quit=true;
    }
    else {
        memmove((void *)&_binLength, &_readbuf[i], 4);
        _binLength=ntohl(_binLength);
i+=4;
        if (_binLength == 0) {
            fprintf(stderr, "\t*** zero-byte length read ***\n");
            // call binary handler with 0 bytes to signal end
            if (_binHndlSet) {
                _binHndl(_userData, NULL, 0);
                _binHndlSet=false;
            }
            _state=LIMBO;
        }
        else {
            _bindat=new unsigned char _binLength];
            _binPos=0;
            _state=BINREADDATA;
        }
    }
break;
case BINREADDATA:
    if ((UINT)_readbufRemain+1) < ((UINT)_binLength-(UINT)_binPos) {
        memmove(_bindat+_binPos, &_readbuf[i], _readbufRemain+1);
        _binPos+=(_readbufRemain+1);
        _readbufRemain=0;
        quit=true;
    }
else {
    memmove(_bindat+_binPos, &_readbuf[i], _binLength-_binPos);
    _binPos=_binLength;
    i+=(_binLength-_binPos);
    // send binary data to handler and deallocate binary buffer
    if (_binHndlSet)
        _binHndl(_userData, _bindat, _binLength);
    else
        fprintf(stderr, "WARNING: no binary handler function set!\n");
    delete[] _bindat;
    _bindat=NULL;
    _state=LIMBO;
} 
break;

case EOM:
    // delete binary handler, if one exists
    if (_binHndlSet) _binHndlSet=false;
    // change state to LIMBO
    _state=LIMBO;
    i++;
    break;
}

if (_readbufRemain==0) quit=true;
}
flushbuf(parsebufLen, 0);
return 1;
}

void MsgReader::start_handler(void *data, const char *el,
    const char **attr) {
    /* process tag */
    map<string, TAGHANDLER> tagHndl=
        reinterpret_cast<MsgReader*>(data)->_tagHndl;
    map<string, TAGHANDLER>::iterator tgh=tagHndl.find(string(el));
    void *ud=reinterpret_cast<MsgReader*>(data)->_userData;
    void *bh=(void*)&(reinterpret_cast<MsgReader*>(data)->_binHndl);
    if (tgh == tagHndl.end()) {
        // run default handler
        if (reinterpret_cast<MsgReader*>(data)->_defHndlSet) {
            reinterpret_cast<MsgReader*>(data)->
                _defaultHndl(ud, bh, el, attr);
        } else fprintf(stderr, "No default handler set\n");
    } else {

TAGHANDLER th=(TAGHANDLER)tgh->second;
th(ud, bh, el, attr);
}

void MsgReader::end_handler(void *data, const char *el) {
    // currently does nothing
    // may be used for supporting XML end-tags
}

/* send _parsebuf to parser
   * parser object must be already initialized */
void MsgReader::flushbuf(int &len, int finished) {
    if (len == 0) return;

    if (!XML_Parse(_p, _parsebuf, len, finished)) {
        fprintf(stderr, "Parse error at line %d: \n\n",
                XML_GetCurrentLineNumber(_p),
                XML_ErrorString(XML_GetErrorCode(_p)));
        exit(-1);
    }
    len=0;
}
Bibliography


