Proposal for Thesis Research in Partial Fulfillment
of the Requirements for the Degree of
Master of Engineering

Title: An XML Messaging Protocol for Multimodal Oxygen Applications

Submitted by: Justin H. Kuo
129 Franklin Street #132
Cambridge, MA 02139

Thesis Advisors: James R. Glass and D. Scott Cyphers

Date of Submission: November 16, 2001

Expected Completion Date: June 2002

Laboratory: Laboratory for Computer Science
Spoken Language Systems Group
1 Introduction

Recent advances in computer technology have brought about increased computer usage among the general public. Computers are rapidly becoming integrated into our lives as they serve more practical and feasible purposes; they have grown beyond the role of their earliest predecessors as oversized tools that helped scientists and engineers solve equations.

The Oxygen project at MIT is an initiative to advance the capability of computers in everyday life [1]. It promotes the notion of pervasive computing, whereby various forms of computation are easily accessible to a user from anywhere at anytime. The goal is to build systems that will enable humans to interact with their environment freely and naturally, and have the environment respond and react to users’ commands and actions.

In addition, humans need not communicate with the environment using conventional devices such as keyboards and mice, but rather through natural forms of communication such as speech and vision. In other words, Oxygen aims to have people “accomplish more with less work.”

The vision of Oxygen also illustrates a multimodal aspect of computing. Analogous in nature to the term “multimedia”, multimodal stresses different modes of interaction with the system. Examples include the aforementioned vision and speech recognition systems, as well as the traditional keyboard, mouse, stylus, etc. Whereas before, each mode was modularized and required specific setup before use, the intended system integrates all the modes into one conglomeration whose integration details are hidden from the end-user. In this sense, the user need not conform to restricted types of input to the system, but rather, the system will conform to the forms of interaction the user prefers.

The general paradigm by which an Oxygen environment might be constructed consists of combining and/or integrating stand-alone systems. Integrating these components into a single environment, however, is not trivial.

Part of the difficulty arises from the fact that these systems operate independently of each other. In order for one system to efficiently communicate with another, each system may be required to know specific details about the other. However, this information may have been abstracted away by the other system, to hide details away from external programs. Having a system delve into the internals of another system defeats the purpose of having the abstraction barrier in the first place. Furthermore, were a system to store information about another system it tries to communicate with, the amount of data it needed to store would grow to an intractable size as the system communicates with more and more independent systems.

The goal of this project is to address the issue of integrating systems in Oxygen while preserving their independence from one another, and to define a protocol that will do this easily and efficiently. It will require evaluation of the current protocols used in this sort
of distributed computing, and identification of their key features and advantages suitable for the Oxygen environment. Whatever method used to integrate systems must do so with very little modification to the original system, thus adhering to the modular design of the Oxygen environment. Lastly, the culmination of this work will demonstrate the protocol in its application towards integration of a conversational interface into a sample pervasive computing environment.

2  Background

The technology and underlying concepts relevant to this work and its motivations are briefly described below.

2.1  Spoken Language Systems

A spoken language system is one that incorporates speech recognition, natural language understanding, discourse, and other technologies to provide a conversational interface. Users can use natural spoken language to communicate with computers and other machines that employ a spoken language system.

The Spoken Language Systems group (SLS) at the MIT Laboratory for Computer Science (LCS) recently developed the Galaxy-II architecture to manage the suite of human language technology servers that comprise a spoken language system [9, 10]. Using the Galaxy-II framework, SLS has had success in developing various domains for conversational interfaces, such as the Jupiter weather information system [12] and the Mercury flight reservation system [8]. Galaxy-II is of particular interest because it allows the development of a standalone conversational interface by assembling the necessary servers into its hub-based framework.

2.2  Remote Procedure Calls

Remote procedure calls (RPC) address the issue of communicating in a distributed computing environment. The RPC model helps by providing the mechanism that enables procedure calls that occur on a single computer to also occur across a communication network [4]. In effect, RPC opens up an interface by which client-server communication can be achieved and a program has access to procedures located in external systems.

Many RPC architectures utilize a technique called marshalling 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 unmarshalls the request into the original contents and proceeds to call 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 [11], CORBA, DCOM, Java RMI, and SOAP [7], all of which have advantages and disadvantages. In general, the high-level behavior of RPC (of facilitating function calls to other machines and returning data to the call’s origin) is highly attractive for use in integrating Project Oxygen systems.

On the other hand, not all aspects of the current RPC models are suitable for the Oxygen environment. In particular, some characteristics stand out as being problematic when trying to implement certain features of Oxygen:

- RPC follows a conventional client-server approach, while Oxygen systems could potentially operate in a decentralized peer-to-peer fashion. Any system may act as either (or both) a client or server, so communication with another system should not depend on its role as a client or a server.

- Some RPC protocols use binary formats that are transmitted across the network. 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.

- In accordance with the concept of pervasive computing, the protocol should allow for robust handling of dynamic resource locations. In cases where machines may dynamically change locations, it is not straightforward (if even possible) to convey these changes to an RPC protocol, which may or may not assume that the client and server are at fixed locations.

### 2.3 Messaging Protocol

A simpler alternative to RPC that can accomplish some of the same tasks as RPC without encountering the problems stated above is a message-passing system. Messages are simply formatted lines of text relayed from one location to another and dispatched by the receiving end. These messages are similar in nature to the messages sent in a typical windowing system (like X-windows or Microsoft Windows), representing events or commands. If implemented correctly, they can contain both ASCII and binary data, meaning messages can also act as carriers for binary data transfer between machines.

The advantage of using a message-passing system with respect to Oxygen is that it features a “push” model where messages containing relevant information are pushed from sender to recipient. Under RPC, when a remote procedure is called, the local machine waits for the response, possibly yielding to other threads in the application. By using messages, the local machine need not wait for the remote machine; it can simply continue with certain other tasks until the remote machine interrupts by pushing in a message. For this reason, this
project will use a form of message protocol instead of RPC to facilitate system-to-system communication across a network.

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) in its use of tags and attributes, but adds more power by letting programs that decode XML documents define what the tags mean. Moreover, XML provides a text formatting that is platform-independent, extensible, and even supports internationalization/localization.

The following example illustrates the well-formed, nested structure of XML as well as its use of freely-defined tags and variable number of key-attribute pairs:

```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>
```

The extensible nature of XML is particularly useful for the messaging protocol in question, since it leaves the definitions of tags open-ended. Furthermore, encoding messages in XML allows generic support of variable message formats (in case specifications or other unforeseeable changes occur during the project) without having to build debugging tools prior to when the full specification has been defined and implemented.

Also, XML is well-supported, and there are many tools (e.g., parsers) available, some open-source, that can be used during development and implementation.

3 Project Description

3.1 Overview

This thesis will focus on defining and developing an XML-based messaging protocol to facilitate the requirements of the Oxygen distributed computing environment. This protocol
will have the following characteristics:

- Messages may contain text and binary data, and the receiver must be able to easily distinguish text from binary.

- Messages will be independent of the transport protocol (e.g., TCP, UDP), though they may contain specific information that can affect future resource location and communication involving the host system, such as dynamic URIs, port numbers, etc.

- The mechanism for message processing must be simple yet flexible enough to be used “as is” in both single-threaded and multi-threaded applications.

- Message tags will be extensible in that systems using these messages can define and introduce unique tags specific to their domains.

The objective is to illustrate that the messaging protocol is a suitable method to integrate independent systems in Oxygen. This will involve a demonstration using Compaq iPAQ handheld PCs that interface with a system built with the SLS Galaxy architecture.

### 3.2 Preliminary Protocol Specification

The specification for messages follow these assumptions and design decisions:

- Messages are multi-lined: one line holds text, another holds binary, and the last line is blank to signal the end of the message.

- Messages must have a reasonable upper limit on length, where “reasonable” has yet to be defined.

- For simplicity, each message will contain at most one discrete piece of binary data.

Messages borrow the convention for Hypertext Transfer Protocol (HTTP) header fields [3] that each line in a message is delimited with a carriage return-linefeed pair \(\text{(<CR><LF>)}\). Every message will contain either two or three lines: the first line always holds text, the second line is optional and holds binary, and the last line is blank to signal the end of the messages. To distinguish text from binary, every line for text begins with $\$$ (including the end-of-message line) while every line for binary begins with $\#$. Thus, a typical message looks like this:

\[
\begin{align*}
\$$[\text{ASCII text}] & \text{<CR><LF>} \\
\$#[\text{Binary data}] & \text{<CR><LF>} \quad \text{optional} \\
\$$ & \text{<CR><LF>} \quad \text{end of message}
\end{align*}
\]

\(1\text{<CR><LF>}\) is used because ASCII text will not contain \text{<CR><LF>}, making it easy to distinguish
ASCII text contains the XML message tags. For now, the only requirement is that all lines of ASCII text begin and end with `<ENVELOPE>` and `</ENVELOPE>` tags. This tag, which is borrowed from SOAP, holds information about the sender and recipient of the corresponding message, along with any other information relating to message delivery but not pertinent to the actual message content. The rest of the message tags are nested within the `<ENVELOPE>` tag like in ordinary XML.

Binary data is transmitted as raw bytes, but prepended with a 4-byte field that represents the length (in bytes) of the data chunk.

In summary, the rough skeleton of a complete message is given below:

```
$$<ENVELOPE>[XML message tags]</ENVELOPE><CR><LF>
#$[4-byte length header][binary data]<CR><LF>
$$<CR><LF>
```

### 3.3 Software Infrastructure

The software package written for this project will construct and process messages that adhere to the final protocol specifications. All code is written in C++ under the Linux operating system, and should be portable across different architectures (e.g., Intel x86 and StrongARM).

There are essentially two components: a message writer (MWriter) and a message reader (MReader). Together, MWriter and MReader accomplish the entire set of tasks required for a messaging system. The intent is that MWriter writes messages and sends them to MReader, which parses and dispatches messages. Multiple instances of MReader and MWriter may exist on a system, all of which act independently from one another.

Since the messaging system is targeted specifically for network communication, the initial implementations of MWriter and MReader write to and read from UNIX file descriptors linked to TCP sockets. TCP is used because it is a reliable protocol and is a widely accepted and supported standard.

MWriter will be an interface to compose messages consisting of pre-defined tags. It will also be responsible for packaging any binary data with a text message. The MWriter component collects tags in a buffer (a queue) until some predetermined buffer length is achieved, after which it will construct the message and send it across the network.

MReader will be the interface to receive and parse messages, and dispatch accordingly. MReader will contain a parser to parse overall structure, and an XML parser to parse XML message tags. The interface will allow for registration of callback functions for specific tags that are parsed in the message (one function per unique tag name). There is an additional callback function to be registered that is called when MReader reads binary data in a message. Callback (also called handler) registration is illustrated in the following sample
pseudo-code for the XML example in Section 2.4.

```c
void person_cb(); // callback handler for "person" XML tag
void name_cb(); // ... same for "name" XML tag
void binary_cb(); // ... and for binary data
MsgReader mr; // MsgReader object
...
/* register callbacks to their associated tags */
mr.RegisterHandler("person", person_cb);
mr.RegisterHandler("name", name_cb);
/* register a handler for binary data, if any arrives */
mr.RegisterBinaryHandler(binary_cb);
...
mr.parse(); // parse incoming message
```

### 3.4 High-Level Design

This messaging system is first and foremost a mechanism by which to send events from one computer to another across a network. Through the use of events, this messaging system is a tool to achieve multimodality in the Oxygen environment. Distinct message tags tied to event handlers can capture keystrokes, speech, and potentially any other form of input (and output), given that their representation can be abstracted as an XML message tag. Some examples of what can be captured are presented in the table below.

<table>
<thead>
<tr>
<th>Input modes</th>
<th>Output modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>• mouse clicks</td>
<td>• audio (speech, music, etc.)</td>
</tr>
<tr>
<td>• mouse/stylus movement</td>
<td>• images</td>
</tr>
<tr>
<td>(e.g., drawing circles, lines, and other gestures)</td>
<td>• video</td>
</tr>
<tr>
<td>• audio</td>
<td>• text (i.e., typed from keyboard, written using stylus, etc.)</td>
</tr>
<tr>
<td>• images (i.e., for face recognition)</td>
<td>• commands</td>
</tr>
<tr>
<td>• video (i.e., video conferencing)</td>
<td>(from one computer to another)</td>
</tr>
</tbody>
</table>

The messaging system functions as another layer of abstraction, built on top of present systems, whose job is to mediate inter-machine communication. It effectively acts as a glue between independent systems, as Figure 1 illustrates for a hypothetical multimodal environment (e.g., Oxygen).
Figure 1: Diagram of a multimodal environment using the messaging system. Each modal component communicates with an application’s messaging interface (denoted “M”), which can also communicate with other systems (such as a Galaxy-based system) in the Oxygen environment via the same messaging protocol.

### 3.5 Timeline and Milestones

Preparatory work for the thesis began in the summer of 2001. This thesis has an expected completion date of June 2002.

Preliminary specifications for the protocol were defined during June and July of 2001. At the end of August, a simple MWriter and parser with limited functionality were written in C++ to test and validate the effectiveness and practicality of the current design. During September, refinements were made to the design, and additional functionality was implemented in the source code.

The next phase of development, to take place between October and December, will involve testing across a networking environment, cross-compiling for different architectures (such as x86 for the desktop computers and StrongARM for the Compaq iPAQ handheld PC’s), revision and refinement of a final protocol specification, and finishing code that will lead to the first version of a fully-operational messaging system.

Afterwards, development of a demonstration system involving a conversational interface and a Compaq iPAQ will take place. This involves creation of a GUI for the sample application, development of the conversational interface using the existing framework provided by SLS, and integration with the messaging software package. This will also help test the messaging system and discover whatever bugs or inadequacies that should be resolved.
By mid-April, the final messaging system will be finished and released. Afterwards, the demonstration will be ready and finished by mid-May. During April and May, time will be divided between these tasks and writing the thesis.

The end result that we hope to achieve is to build a message protocol package that others can use and successfully integrate into current systems (particularly in Oxygen) without much modification of their codebase. Systems such as the SLS Galaxy audio server and the SpeechBuilder [6] software suite are targets for integration that could benefit from the use of messaging. For example, the current Galaxy audio server used for Oxygen application implements Sun RPC, which cannot support the proposed Oxygen feature of dynamic locations (e.g., using INS, the Intentional Naming System [2]). Redesigning the audio server with the messaging protocol will allow integration with INS while maintaining backwards compatibility with current protocols until INS has been fully deployed. Messaging could also enhance SpeechBuilder by enabling users to quickly access, construct, and deploy conversational domains from a pervasive computing environment such as Oxygen.
References


