Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) Major Qualifying Projects

April 2006

eBay Skype Web Services

Michael C. Leonardo
Worcester Polytechnic Institute

Nicholas E. Bonatsakis
Worcester Polytechnic Institute

Thomas Allen Schindler
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
Leonardo, M. C., Bonatsakis, N. E., & Schindler, T. A. (2006). *eBay Skype Web Services*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/1079

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

eBay Skype Web Services

MQP Report

WORCESTER POLYTECHNIC INSTITUTE

by

Thomas A. Schindler

Michael Leonardo

Nicholas Bonatsakis

Date: April 21, 2006

Professor David Finkel

Abstract

In this project we investigated the feasibility of implementing web services over a peer to peer infrastructure. Working within the eBay Research Labs team, we built several prototypes that leveraged a new feature of the Skype API which allows application to application messages to be exchanged between Skype clients. The prototypes were written as standalone java applications which connect to the Skype API and use java swing classes to provide a user interface.

Table of Contents

1.0 Introduction

The eBay Research Labs is a fairly new division of eBay. Their main purpose is to experiment with and investigate new ideas and technologies that could be applicable to eBay in either the near or distant future. After eBay's recent acquisition of Skype, the Labs were presented with an idea for a project that leveraged the large scale peer to peer network formed by Skype. A team led by the labs spent the later part of 2005 doing architectural design, simulation and persuading others that a prototype should be built. When we arrived, it was at just the right time to form the core team to do the detailed design and coding of the prototype.

The eBay Research Labs were interested in having us produce a prototype; a proof of concept that would show the managers at eBay that the idea was both possible and innovative enough to warrant funding full production.

To implement this project, we were to develop a prototype Skype plug-in. The plug-in would use the Skype API to implement a complex protocol over the already in place Skype peer-to-peer architecture

The project was broken into three sections. The first section consisted of a simple prototype that demonstrated the ability to write a plug-in for Skype, and some simple interactions with the Skype client. This section of the project lasted about 12-14 days. The second section was rapidly developed prototype of the Skype product. This section of the project lasted about 10 working days. After section 2, we had most of the features of the Skype program working to our satisfaction, but due to the short amount of time we spent working on the prototype, it was not as organized as we had hoped for. Section 3 is when we started cleaning up our code and re-writing the product in a more finalized form. This section lasted us the rest of our time at eBay, which was about 15 working days. The procedure section of this report is divided into sub-sections that correlate to the aforementioned parts.

2.0 Background

2.1 eBay History and Overview

In 1995 established computer programmer Pierre Omidyar had an idea for an Internet based marketplace system. He had little knowledge of auctions; in fact he had never even attended one in his life. Yet over a long weekend, he created the first version of what would eventually become eBay and introduced it to the public. "AuctionWeb" as it was initially known as, allowed users to auction off items that fell within specific categories, such as antiques, books, and electronics. AuctionWeb hosted thousands of items through the end of that year, and in 1996 Omidyar decided to see if users would accept small fees for listing their items. The fees were accepted and the user base grew considerably. Eventually, the website expanded into a company, was renamed to eBay and acquired a full staff. In 1998 eBay went public on the New York Stock Exchange and went on to become one of the most successful Internet ventures in history.

Today, eBay continues to thrive. With a staff of over 11,000 (2,500 at its company headquarters in San Jose CA), it has blossomed into the largest Internet person-to-person trading system in the world. With over 150 million registered users, it is clear that eBay has rooted itself within the structure of today's society and will continue to support the ever-expanding online trading community. [1,2]

2.2 Web Services API - SOAP

SOAP stands for simple object access protocol, and allows a very flexible system for communicating information in a decentralized environment. The SOAP messages have an XML format with three parts. From the draft W3C specification: SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. [5]

eBay's API makes use of the SOAP protocol, and has objects that model the basic components of eBay's website. These objects are:

- Item
- Category
- User
- Transaction
- Feedback

Each of these items has a large number of sub-items which further help describe the item. These sub-items can be a simple string, such as the item.title, which is simply the title of the item, or a complex set of data that has its own sub-items, such as item.shippingdetails. [6]

2.3 XML

XML stands for Extensible Markup Language. This language allows for the programmer to create a standardized set of structures to define a complex document. Unlike HTML files or other web formats, XML tags only define the data structure, these tags do not directly specify the way the data will be displayed. If the programmer has access to the XML schema file, which is a file specifying the syntax the XML document is formed in, a program can be written to parse the XML file based on this schema file, and produce user-friendly formatted results.[7] This language is commonly used in areas such as news feeds, invoice data feeds, stock market feeds and many other areas. A sample piece of an XML file looks like:

```
<note>
  <to>Bill Gates</to>
  <from>Steve Jobs</from>
  <heading>Advice</heading>
  <body>Use the internet wisely!</body>
</note>
```

This file defines a simple note; it fields that define who the note is to, who it is from, a heading for the note, and a body text. This one block could be repeated over and over again for multiple notes.

2.4 eBay API

eBay is one of the most widely used Internet auction sites in the world. Everyday thousands of items are listed and sold through their auction system. People traditionally list items by using the eBay website to enter all the required data about their item but power sellers upload items in bulk using the eBay API interface.

API stands for Application Program Interface and is a simple way for software to communicate with other software.[8] The API for eBay consists of over 100 SOAP functions with matching calls for Java, PHP, Perl, XML, and python. These calls do everything the eBay site can do and more. Examples include adding items on the eBay site, getting popular keywords, and checking high bidders.

Individuals using the eBay API have a limited number of "free" calls to the service, this number varies, but at the moment it is set at 10,000 per month. Users wishing to use more calls per month than the individual license allows can purchase a Commercial API License which allows them many more API calls, up to about 1.5 million per day.

To gain access to and use the eBay API one must create an eBay developer account which is available from http://developer.eBay.com. This ID must be tied to an active eBay username. A sandbox environment must also be created. Once that is complete, users can be created within that sandbox. The sandbox is more or less a small contained version of the eBay application. The user can practice making calls that add or modify items and users in their sandbox while not affecting the actual eBay systems.

The user must also create a set of tokens. These tokens are called your Application ID, Developer ID, and Certification ID. An authentication token is also needed for your calls to successfully connect to the eBay sandbox. All of these tokens can be created from the eBay developer website.

Once the application you have written is ready for "prime time", that is, being connected to the live production eBay service, it must be checked. EBay will not allow just any application to be getting and posting data directly from their servers, as that would be a major security hazard. EBay has implemented a certification process which allows you to self-certify your application by filling out information about your application, and submitting it to eBay for approval. Another method for certification is the standard application certification, which requires you to submit your application to eBay and get it approved. Once the application is certified, new production tokens must be used in order to run on the production eBay. [6]

Anyone can create an eBay developer account and experiment with the system themselves. There are many code samples and tutorials available on the eBay developer site. The samples are written in a number of different languages: ASP, VB, VB.NET, C#, PHP, PHP5, Python, Perl, and Java.

2.5 Skype

One of the most promising technologies in today’s market is Voice Over IP (VoIP). Voice over IP is viewed as the future of communication, likely to replace conventional phone systems altogether. While many companies are currently rushing to be players in the oncoming VoIP war, one company stands far above the rest. With over 200 million downloads, that company is Skype, the industry’s leader in VoIP.

Skype is a joint business venture between Niklas Zennstrom and Janus Friis started in August of 2003. Each founder has a reputation of starting successful cutting edge businesses. Both were cofounders of Kazaa, the most downloaded Internet software in the world, with over 370 million downloads. They both have also had their hands in several other successful companies such as Joltid, Altnet, and Bullguard. [10] Skype has been their most successful venture. Zennstrom is the CEO of the company, which has headquarters in Luxembourg with offices in London and Tallinn. They also have an office in Silicon Valley, but no other U.S. branch.

Skype holds a strong position in the mind of consumers when it comes to VoIP. The company is well known for its free unlimited voice calls (from Skype to Skype users), chat, and file sharing services. Along with the free calls, you can call a normal landline and have your own Skype number which landlines can call – both for a fee. [10] You can also pay for better quality of service. Skype has a total user based of over 75 million people and is growing by over 150,000 users a day.

Skype's technology is based on peer to peer principles and designed with security in mind. Instead of simply sending voice packets over the network and through untrustworthy nodes, Skype uses 256-bit AES encryption actively on the data stream to prevent ease droppers from reconstructing the voice stream. Skype also uses its own proprietary session establishment protocol; which prevents nodes on the internet from recording a login transmission and then pretending to be the sender by logging in with the same data. [9]

Skype has gone under much criticism over the past couple of years but has confronted any issues head on. Questions about its security have been met with detailed analyses of the behind-the-scenes encryption and observation of algorithms used by Skype. With hardware rushing to improve data rate and quality, Skype looks to hold a firm grip on the future of communication. With the careful planning and implementation of their software, they intend to tighten that grip with each passing day.

2.6 Skype API

Skype, much like eBay, has provided an API to allow programmers to write plug-ins that communicate with a Skype client. The API has access to everything you can do through the traditional user interface, as well as some features that are hidden to the user.

The API allows for both software and hardware interface, which has allowed for the development of USB phones that control Skype.

The API is a message based system, which means the communication between the Skype and the plug-in is done through the use of pure text strings. The API has objects for each of the following:

- USER – information about a Skype user
- PROFILE - information about the user's Skype profile
- CALL - an object to allow calling of other Skype users
- MESSAGE – an object that contains information about a single Skype message
- CHAT –contains information about Skype chat sessions
- CHATMESSAGE – contains information about individual chat messages
- VOICEMAIL - allows access to voicemail
- APPLICATION – allows plug-ins to talk to each other over Application to Application messaging
- GROUP – information about the groups on a Skype buddy list

All of the objects are accessible to the user except for the Application object. The application object allows plug-ins to talk to other plug-ins with the same application name over the Application to Application messaging system, which allows text messages to be sent without the user seeing them.

Due to the message-based nature of the Skype API, it can be used from almost any programming language. There is sample code on Skype's website for Java, and C/C++/C# applications.(http://share.skype.com/directory/)A sample conversation between two Skype clients over application to application messaging would go as follows:[11]

All lines indicated as => are outgoing messages to the Skype API.
All lines indicated as <= are incoming messages from the Skype API.

Refer to the lines starting with // for the flow of the sample.

```
// register application on both sides
[JIM] => CREATE APPLICATION SampleApp
[JIM] <= CREATE APPLICATION SampleApp
[JOE] => CREATE APPLICATION SampleApp
[JOE] <= CREATE APPLICATION SampleApp
// JIM initiates communication to JOE
[JIM] => ALTER APPLICATION SampleApp CONNECT joe
```

```
[JIM] <= ALTER APPLICATION SampleApp CONNECT joe
// connection establishing ...
[JIM] <= APPLICATION SampleApp CONNECTING joe
// .. and is successful
[JIM] <= APPLICATION SampleApp CONNECTING
// .. and creates one stream
[JIM] <= APPLICATION SampleApp STREAMS joe:1
// and JOE is notified by new stream
[JOE] <= APPLICATION SampleApp STREAMS jim:1
// JIM sends data over stream to JOE
[JIM] => ALTER APPLICATION SampleApp WRITE joe:1 Hello world!
[JIM] <= ALTER APPLICATION SampleApp WRITE joe:1
// stay tuned while data is transmitted...
[JIM] <= APPLICATION SampleApp SENDING joe:1
// .. and you are notified on delivery success
[JIM] <= APPLICATION SampleApp SENDING
// JOE receives notification about the incoming message
[JOE] <= APPLICATION SampleApp RECEIVED jim:1
// .. and reads data from stream
[JOE] => ALTER APPLICATION SampleApp READ jim:1
[JOE] <= ALTER APPLICATION SampleApp READ jim:1 Hello world!
// ... and is notified that stream is empty
[JOE] <= APPLICATION SampleApp RECEIVED
// JOE sends back acknowledgement of message
// A datagram is used because it is not so important to acknowledge
[JOE] => ALTER APPLICATION SampleApp DATAGRAM jim:1 Hello back!
[JOE] <= ALTER APPLICATION SampleApp DATAGRAM jim:1
// Now data is transmitted...
[JOE] <= APPLICATION SampleApp SENDING jim:1=11
// .. and notificed when it was sent (but delivery not assured)
[JOE] <= APPLICATION SampleApp SENDING
// JIM receives datagram notifcation
[JIM] <= APPLICATION SampleApp DATAGRAM joe:1 Hello back!
// JIM decides to end the communication
[JIM] => ALTER APPLICATION SampleApp DISCONNECT joe:1
[JIM] <= ALTER APPLICATION SampleApp DISCONNECT joe:1
// .. and when stream is closed it is notified
[JIM] <= APPLICATION SampleApp STREAMS
// Also JOE receives notification that stream was closed
[JOE] <= APPLICATION SampleApp STREAMS
// JIM unregisters applicaton
[JIM] => DELETE APPLICATION SampleApp
[JIM] <= DELETE APPLICATION SampleApp
// JOE unregisters applicaton
[JOE] => DELETE APPLICATION SampleApp
[JOE] <= DELETE APPLICATION SampleApp
```

3.0 Project Summary

. In November, 2005, Skype added an underlying messaging system to their network known as Application to Application communication. Applications running as Skype plug-ins can communicate with each other without the user seeing the interaction.

This project involved creating an application that would be used to form a large scale network using the underlying App2App protocol from Skype. In essence the application would rely completely on Skype for the P2P connections while managing exactly what to send and who to send data to. Such a network would rely on each Skype client node to perform the fundamental functions that would normally be performed over http to a central web server in a conventional design. The key advantage of a Skype based application is that it scales without needing to invest in a central web server.

4.0 Procedure

4.1 Phase One – Learning/About Me Program

4.1.1 Description

The overall scope of the prototype was massive. Along with building it from scratch and dealing with constant design changes along the way, we also had very little experience working with the actual technologies involved. The eBay Research Labs expected us to use SOAP, XML, and the Skype API, none of which we were proficient at. So instead of immediately starting the prototype, our advisor decided that our group's time was better spent first working on a mini-application that we could learn from. This project also provided a testing ground to see if certain ideas were possible, such as using the underlying application to application messaging service in the Skype API to implement the entire Skype peer to peer network. The mini-application was called *AboutMe.*

The *AboutMe* program performed a fairly simple task; it would show you who was on your Skype buddy list and also running *AboutMe* and would allow you to retrieve a small picture and biography from any of those people. The small Java application consisted of a user interface with a simple list of buddies and two small buttons; one labeled “Update Buddies” and the other labeled “Get Buddy Info”. Clicking “Update Buddies” would populate the list with all users running *AboutMe*, and clicking “Get Buddy Info” would retrieve a short biography of whoever was selected in the list.

4.1.2 Design

We started designing *AboutMe* by deciding on exactly how to split up the program into clean, individual tasks that also made for a good design. The three sections we decided were most separate and modular were the Skype connection, the buddy list, and the user interface.

The Skype connection was a key element in the success of both the *AboutMe* and the Skype prototype project. To keep the Skype connection separate and clean, everything pertaining to it was handled by one manager. We called this class the SkypeManager. The SkypeManager would send and receive all transmissions from all other *AboutMe* applications running. It would also be responsible for actually connecting to all other users at the initial start of the program.

The next section of the program we decided on was the buddy list. A buddy list manager would be created (called BuddyListManager) to specifically organize and maintain your list of buddies who were using *AboutMe*. The idea was that it could also store the addition information and picture as they were retrieved. This way all relevant information about all your buddies could be controlled by one manager. Whenever

SkypeManager needed to get information it could just pull it from the BuddyListManager. Also whenever a user asked for more information on a user, SkypeManager could retrieve from the network and then store it in BuddyListManager. In this way the BuddyListManager would also really be the storage component of the program.

The user interface would make up the last section. The interface was fairly straight forward but a lot of work. It would be the core to the program; responsible for starting up the SkypeManager and BuddyListManager along with having the graphical code to generate the list, buttons and various listeners.

4.1.3 Implementation

The implementation process of the first phase of the project inherently involved a great deal of decision making as problems arose and needed to be dealt with. Each of the three primary sections of *AboutMe* spawned unique issues, and dynamic changes to the design were inevitable.

The most difficult issues came from the SkypeManager. The first task was ironing out how to actually use the Skype API. At the time Skype had released an API which could be used to do exactly what was needed, but it was not written in Java. Luckily, at the same time, an individual released a package on the internet called jSkype. This JSkype project was a Java wrapper class that we could use to access the underlying Skype API. After installing and trying it out we found it couldn’t be used as an included standalone JAR library. Instead we were forced to import the entire JSkype source code, which cluttered our project. Since we really couldn’t find a better alternative, it was left in for this phase of the project.

After working with the Skype API, another problem with the Skype connection arose. To figure out who was also running *AboutMe*, the program needed to poll each person on the actual Skype buddy list and attempt to connect to them. When connecting one *AboutMe* application to another one (another instance of *AboutMe*), the Skype API would block the program from continuing until it received a response. If the other user *wasn’t* running *AboutMe* then the program would have to wait for a response until it timed out before it could continue connecting to other users. To deal with this problem we opened a new thread for each user on the Skype buddy list. This meant the program would simultaneously try to connect to everyone, allowing some connections to wait until they timed out but still allow those people who *were* running the application to connect. We decided to do this because it allowed connections to show up faster on the user interface then waiting for each user one by one, especially when the Skype buddy list was large.

Also embedded in the SkypeManager was the messaging. In addition to connecting to users, we needed to decide how to communicate with other instances of the program. In short we had to decide on a messaging standard. Our original design from eBay called for SOAP as the primary messaging protocol; however after a significant amount of trial and error, we determined that it would be more straight-forward to just use standard XML. SOAP was designed far more with remote procedure calls in mind. It has the ability to make direct calls to a given URL and looks for a response from the server. What we were actually looking to do, however, was just to send objects over

Skype as strings; SOAP was just not tailored for this purpose. We then implemented an object to XML converter class that would XML-encode our messages and produce a string that could subsequently be sent over Skype and be “decoded” back into Java objects on the other end.

Another problem that we encountered was the transmission of images. Specifically we ran into some issues when attempting to encode an image file as a string. When an image is encoded as a set of bytes, some bytes represent characters that cannot be interpreted by Skype (particularly the 0x00 character). When we attempted to send the faulty characters over the network, the reconstruction of the image on the other end was incorrect. To fix the problem we figured out that we could encode the image as a string in Base64. By converting all image byte sets into base 64, thus restricting the resulting characters to acceptable values, we effectively solved the problem.

Implementation of the BuddyListManager was fairly straightforward. It managed an internal ArrayList of ExtendedUser objects. An ExtendedUser object had three parts; a unique user id, the Stream object that represented the actual connection to a user, and a UserData object that represented the biography and picture information for that user. When a connection was made to a new buddy, BuddyListManager would add a new ExtendedUser to the list which included the new Stream and a user id. Once information had been requested on a specific user, the biography that was returned would be stored in the same ExtendedUser as the UserData. We decided on this implementation because it kept a single record for each user. This meant the program was easily able to retrieve a list of all connected users and, using a specific id, could obtain addition information on each user if they had previously requested it over the network.

Lastly was the implementation of the user interface. As described before, we kept it fairly simple: the list of friends, two buttons, and fields for buddy information to be displayed. We also included a product image (at the time the project's codename was

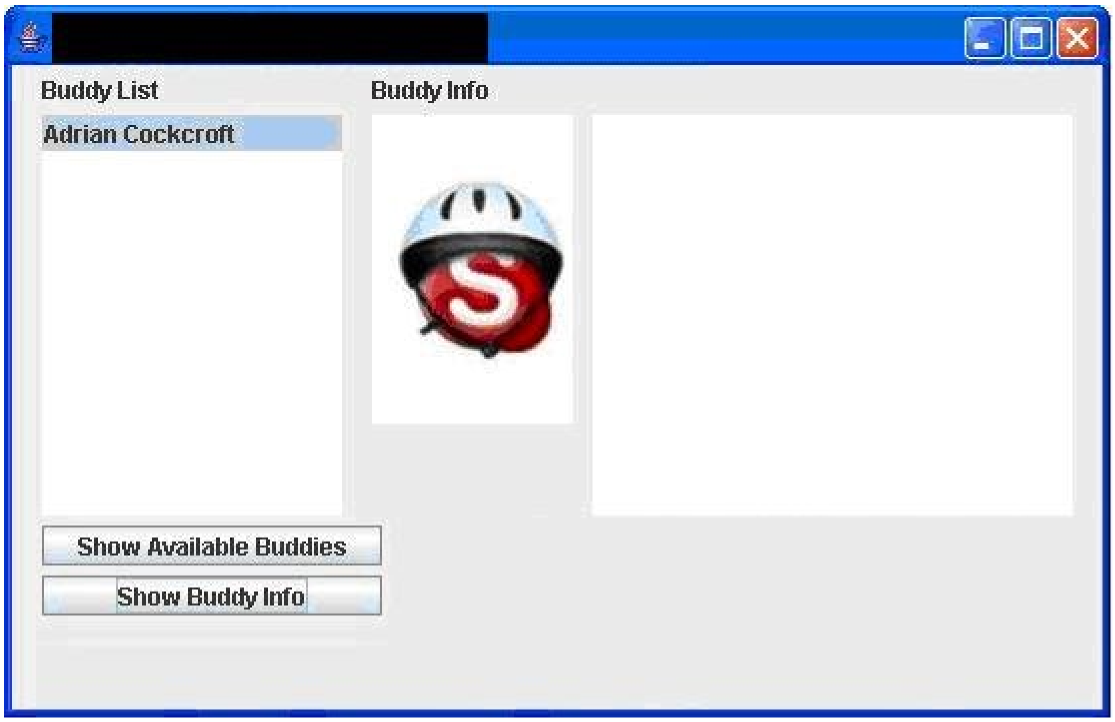

Figure 1

SkypeBay). The only major decisions we made here came from how to display the user's biography and picture. They were displayed using another type of JPanel called a JEditerPane. This type of panel allowed the use of HTML to format how things were displayed on it. This presented the ability to change the layout of the biography fairly easily, and the code could possibly be reused later for web applications. Figure 1 shows the user interface design.

4.2 Phase Two – First Prototype

The Skype prototype included many aspects that made it a very large project Along with our advisor we created several use cases that described different tasks a user could perform.

4.2.1 Design

4.2.1.1 Persistent Storage

Persistent storage for the Skype prototype was accomplished by encoding a class into XML and writing it to file. At startup, the file is read in to recover the previous state.

4.2.1.2 Packaging

Once we had a working prototype, it was important to be able to distribute it for both testing and user feedback on things like usability and features. The overall goal of creating the prototype was to expose the rest of the eBay and Skype developers to the idea in hopes that it would be well received and gain funding for production. An essential part of distributing the software was packaging it in such a way that they could install and run the software through a straight-forward installation process. Upon reviewing all the components needed to run the application, we realized that we needed to implement a standard Windows install program in order to distribute the software. There were several aspects about the way the software ran that led us to this conclusion.

Firstly, because we had been running it mainly from within Eclipse, the Skype prototype only ran as a set of compiled Java class files that referenced a class path file for included libraries. It would not be reasonable to expect every user to manually run compiled Java files through a console, nor would it be acceptable to distribute a package consisting of so many individual files.

Secondly, due to the fact that the software is designed for use on the Windows platform, it requires several external DLL files to be present in the Windows System32 directory. The required DLLs contain libraries that allow Windows to run Java SWT based applications, a feature not native to Windows. While we were in development, we simply added these files manually; however again, it would not be reasonable for an end user to perform this task.

Another reason to have an installer was to manage the content files of the program. The application eventually required a significant number of icon and image files in order to run. It also produced several XML documents used for persistent storage. It was required that these files reside within the proper directory structure with relation to the executable application in order for it to be able to locate them. Allowing the end user access to these files and delegating the responsibility of placing them in the correct directories would not be logical and could result in software malfunctions.

Finally, we decided that getting the distribution to behave like a standard Windows application would make it more user-friendly. It would be best if the installer could place the application files within the proper Windows Program location, create a Start Menu group for shortcuts, and add Desktop or Quick Launch shortcuts if desired.

4.2.1.3 Skype API Communications

The underlying communication with Skype was done through a single class called SkypeManager. This class was defined such that it would open up connections to each user on the Skype buddy list that was running the same plug-in, and keep this connection open. All communication between the two users would happen over this connection.

4.2.2 Implementation

4.2.2.1 Persistent Storage

Persistent storage was fairly straightforward. The system worked so that when a user shutdown the application, it would place all relevant information into the save model class and then convert that class into XML and write it to file. When the program was reactivated, it would attempt to load all items and settings back from the file using the same save model template; this method was used because XML encoding would not work on the main persistent storage class itself due to the more complex methods and overloaded certain getters and setters, which was not allowed in XML encoding/decoding.

4.2.2.2 Packaging

There were several major steps in implementing an installation package. The entire process involved first getting the application to run independently of Eclipse as a standalone JAR file, next converting it into a Windows EXE file, and finally packaging it as a single Windows setup executable. Each of these steps involved some important decisions and trade-offs.

Producing an executable JAR file was the most basic building block of the final distribution package. Eclipse features the ability to export a JAR archive and even to specify a manifest to indicate which class contains the intended Main method. Unfortunately, for our specific application, the Eclipse export feature produced JARs that would not run for some unknown reason. After attempting to remedy the problem, it was decided that it would be more efficient to find an alternative. Some research turned up an Eclipse plug-in called “Fat-JAR”. Fat-JAR was designed specifically to produce one JAR file out of any Eclipse project. Additionally, it includes all required libraries and class paths within that single JAR, resulting in very robust standalone JARs.

The second major step in creating the package was converting the now executable JAR file into a Windows executable (EXE) file. Some more research yielded a simple Java application called “JSmooth”. JSmooth simply takes in a JAR file and an image file (used for an icon). It then asks which class is meant to be run within the JAR file, and it produces the Windows executable.

The final step in producing the distribution package was to create the Windows setup application. The original intent was to use InstallShield as it is considered to be the industry standard for creating installers. It turns out, however, that InstallShield is licensed software. Instead of going through the red tape of obtaining a copy through eBay, we decided it would be easier to find a shareware or freeware version of a different program. We found some success with a program called “Install Stream”.   It offers a wizard-like interface for creating an installer step by step. Ultimately this method proved insufficient as the installer produced was incapable of placing files in specified directories, something that was absolutely required. More investigation led to the use of an application called “Innoo Setup”. Like Install Stream, it provides a wizard-like interface, the difference however, is that the program creates a script file which it follows to create the setup EXE. It is thus possible to manually edit the script in a way that offers a great deal of control. Inno Setup proved to be the best option as it produced a setup file that did everything we required of it.

4.2.2.3 Skype API communications

The communications between all Skype users was handled by the SkypeManager class. This class was originally designed to open up a connection to each user and keep that connection. We soon found out that the Skype API closes any inactive connection that had been open for more then 8 minutes, so we had to have keep-alive messages being sent. Every two minutes, we sent a message, which also serves the purpose of a keep-alive message, out on every connection, ensuring it did not lose connectivity to a user. We encountered some unexpected difficulty in sending and receiving large amounts of data. The Skype API documentation says the Application to Application messages are limited to 0xFFFF bytes, or 64 kilobytes. When we tried to send messages that were larger than 16 kilobytes, the application responded with a syntax error. Upon further investigation and discussions with Skype, the API documentation was found to be incorrect. We had to be sure that all of the messages we sent were under the 16kb limit.

When the Skype prototype is first attached to Skype, it would send out a connect message to everyone on the buddy list over an Application to Application message. App to App commuications uses a Skype communications method called a stream; a stream is communication protocol built into the Skype API which all Application to Application communication is sent over. If the user the message was sent to was running the application, the Skype API would notify the Skype prototype of a stream number on which to communicate. If the user was not running the application, no response would be sent.

Whenever the SkypeManager class needed to send data out or process incoming data, it would spawn a thread to do so. This was done to be sure the program did not block the thread that was listening for incoming data. Skype seemed to be able to handle the multiple threading for two or three users running the prototype, but began having issues when more people were online at the same time. When there were too many users, Skype would not respond and took up 85-95% of the computer's processing power. We believe it was due to the amount of data we were sending through the API. There appeared to be an issue with the API. So, because the Skype connections were so critical to the application, we created a stress testing program to help figure out this problem. The results of our testing can be found in the Results section of this paper.

4.3 Phase 3 - Skype Prototype Refactoring

4.3.1 Design: eBay API Integration

Since the inception of the Skype project, integration with the eBay website was a desired feature. Though the application is primarily intended to be independent of eBay itself some integration between the Skype and eBay APIs was desirable.

Each of these features requires access and calls to the eBay API. Our design for this section of the project called for another singleton manager we named "EBayAPIManager". This manager would handle any tasks relating to the eBay API.

Due to the fact that the prototype now had to make remote calls to an external server, we had to take into consideration how this could affect performance and stability. To handle this, we worked to minimize the number of calls we made, and ensured that redundant calls were avoided. Additionally, it was important to consider how call failures could affect the rest of the application. Thankfully the Java eBay API methods all produce detailed exceptions upon failure. The best way to handle these exceptions was to have any methods throw the exception up so it would surface at upper levels and could be displayed as an error message or logged, rather than catch the exception within the method.

The authentication feature mentioned above was originally planned to ensure that every running copy of the prototype would register itself with eBay.

4.3.2 Design: Messaging

In realizing some of the flaws of our phase 2 messaging system, we took a step back and decided to rework it for phase 3 of the project. We had a much more clear understanding of what was needed and how to make the system more efficient overall.

The first and most important change came in abstracting the message/payload relationship in such a way that each was independent of the other. Achieving the separation meant keeping our hierarchy of a generic parent message class with task specific sub classes, but also designing payload objects that would be carried within the messages. These payload classes would also be specific to their tasks. For example, a specific Message extends the SkypeMessage abstract class, and contains within it an instance of the actual payload object. In this way, the payload class can be changed in any way and not affect the messaging structure.

Another major change we decided upon was the use of enumerated integer values as commands instead of our earlier string commands. This restricts the messages to a task specific set of commands for each message class.   Because there is only a finite set of messages possible, it is less likely that a string parsing problem, or typo on either end of the application code would produce an error. These message commands would be defined within the sub classes, and actually carried within the super class.

4.3.3 Implementation: eBay API Integration

Implementing the code to handle the eBay API calls proved to be more complex than we had originally anticipated in some regards; the fact that the API is constantly being updated made it absolutely necessary to get in communication with other developers. In addition to this, we had to wade through a fair amount of documentation in order to achieve the desired calls and results.

The eBay API has gone through several major revisions since its creation. The most significant of these was a move from a manual SOAP interface in which SOAP-XML messages had to be constructed and sent, to an Apache Axis based system, where all API calls are represented as regular Java methods. Both systems are still supported, it was apparent to us however, that the latter of the two would be best because it handles everything behind the scenes and is much more straight-forward. If we had used the manual XML based system, we would have had to produce even more code to create and handle the XML SOAP envelopes. In addition, though the XML system is still technically supported by eBay, it is now considered legacy software and thus more support is available for the newer Axis system from both eBay and the developer community.

Aside from which version of the API to use, there were some other specific implementation issues to deal with. How to implement the authentication process was not very clear during the design phase, so it took some experimentation to arrive at a feasible solution. In the end, we used a hard wired eBay API Token that every application needs in order to access the API. This token is unique to the application, and by hard coding it, eBay would simply have to deactivate that specific token to render any version of our prototype using that token useless. Alternatively we could have gone with a username and password login authentication process, but this requires one single eBay account to be used as the authenticator for every version of the prototype. It was unclear to us whether or not it would even be possible to obtain a generic account through which every instance of the application would login. Also, the majority of example documentation for the API uses the token authentication method and thus it was much easier to troubleshoot authentication related problems.

5.0 Results/Analysis

5.1 Simulation

The peer to peer nature of this project lends itself to exponential growth. Every node on the network can send and receive search requests, information about items, or offers at any time. The scalability of this network is dependent on the bandwidth available at each node.

With there being possibly tens of thousands if not hundreds of thousands of nodes, there is no easy way to estimate what the bandwidth usage will be like. To figure out the answer to this problem our advisor decided to design a simulator to do just that.

The simulator was written in occam, which is a parallel programming language first created in 1983. The reason for choosing this language was its ability to operate very a large number of threads quickly in a small amount of memory. The simulator spawns a thread for each node to talk back and forth with each other and logs activity to a file in GraphML format – An XML dialect used to describe nodes and edges connecting nodes. With 100,000 nodes running the simulator took about 200 megabytes of memory. What the network would look like with 1000 nodes after making 500 total connections is shown in Figure 2, the graph was rendered from GraphML using the freely available yEd editor from yfiles.com.

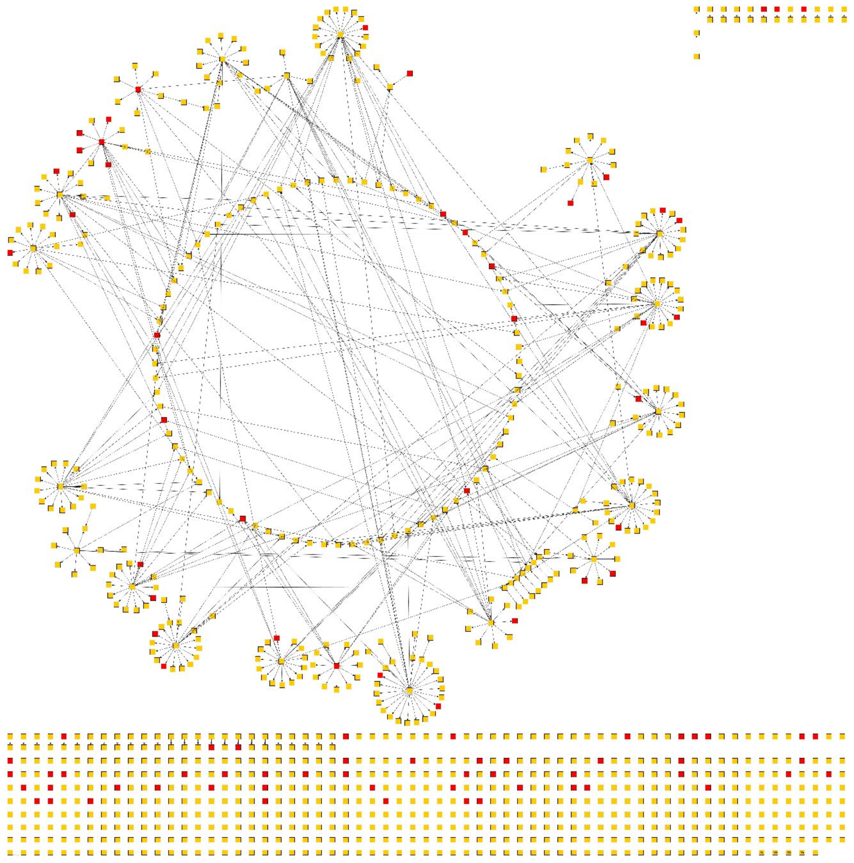

Figure 2

The protocol for the messages that needed to be sent was defined by using this simulator as a proof of concept. As you can see, this application has the capability of creating many connections and forming a lot of virtual social communities in which items can be traded. The occam simulator allowed us to work with a complex protocol that had already been proven to work with defined semantics and message exchange logic.

5.2 Stress Test

Skype's application to application messaging provides a reliable data stream for Skype plug-ins to talk to each other. The size of the messages you can write is limited to 16 kb, but there is a limit to how fast Skype can send them.   When we got to the point in our prototype where there were a lot of messages going back and forth between many different users, Skype started to have difficulties handling all of the data. The first issue we had was that the Skype client would disconnect from the Skype network, but still act like it was online to the user. We suspected that we were somehow overflowing Skype’s ability to handle all of the data.

In order to figure out where the underlying issue was, we wrote a stress testing program that would determine the limits Skype’s capabilities. This program opened a connection to another Skype user and started sending application to application data. We started by sending just one message, and having the receiving user echo that message back to the sender. This seemed to work well, so we started sending multiple messages at the same time. We were originally sending just the a short string under thirty characters, but we soon realized that our messages within the Skype prototype were much longer, so to get more relevant results we vastly increased the size of our string to the point where we were sending about 1kb of text.

Internally, Skype would take the text that all of the threads were sending and queue it up to send over the Application to Application stream. When the receiving Skype client got a message, it would echo it right back to the client who sent it. This provided us with a console message indicating that the transmission was a success.

We gradually ramped up how much text we were sending and how many times to decipher the limits of the program. When we got to the point where we were sending about 1 kb of text 20 times the program crashed on one side or the other.

After submitting a bug report to Skype, they released a new version of Skype, version 2.0.0.81. This version seemed to have fixed the issue in our initial tests, but upon further review it did not.   Instead of just disconnecting from the Skype network, the client program would close completely. After completing our project, Skype released a hotfix version that was shown to have fixed the problem, and the next version of Skype will include a full fix.

6.0 Conclusions

6.1.1 Results

From its inception, it was apparent that the project would require a great deal of creativity and experimentation in terms of both design and implementation. Several technologies that had previously never been used in conjunction, came together to produce an innovative system. The experimental nature of the project and its purpose as both a proof of concept and a learning tool meant that the development process involved a lot of trial and error, as well as approaching problems from different angles. The various decisions we made throughout the process each had an effect on the resulting product, some positive and some negative.

Firstly, the breakdown of the project into three distinct phases, though not completely planned from the start, ended up causing incremental growth and gave us a clearer picture of what was done and what needed to be done. Starting out with the *AboutMe* testing application turned out to be a good decision. Sending messages over Skype Ap2Ap was a new and poorly documented process. Sending structured messages that would be part of such a large system would be challenging. It was valuable to learn things about basic restrictions early on so that these restrictions wouldn’t bog us down later in development.

The deadline for the phase two iteration of the project was unexpectedly moved up on us by about three weeks. As a result, the phase two version was not as robust as we had hoped. Halting development after the first deadline to stop and redesign the entire system was an important decision. Separating the redesign to phase three gave us time to really sit down and rethink every aspect of the prototype and to openly discuss changes with the entire team. At this time, we considered both improvements and new features. The open forum “Work-out” method we followed for this re-design gave everyone on the team a say in every aspect of the newest planned build, this resulted in a much more efficient and practical design for phase three.

Finally, care was taken during the final days of the project center to ensure that the code base would be left in a condition such that future work could be done without us being present. Ensuring easy extensibility turned out to not be very difficult due to the fact that our phase three design was more consistent throughout the different modules of the project.

6.1.2 Accomplishments

Throughout our nine week period at the eBay Research Labs we strived to be as productive as possible. In short, we had the opportunity to develop a complex application from original inception to design and through implementation. We worked as part of a team of experienced engineers with the common goal of making the Skype proof of concept prototype a reality.

The first and most basic building block was the XML based Application to Application messaging protocol. The fact that the Skype Ap2Ap had just been released at the start of the project meant that the system we had to design was something that had never been done previously. It took significant time and experimentation to solidify a proper design and implementation for the needed protocol.

Once the messaging protocol had been developed, we had to make many decisions with the entire team about how to proceed. We were able to design a system that could accurately produce the desired environment. Interpreting the original business model for this project and producing a system that actually followed it was a significant portion of our work and represents a large accomplishment.

Lastly, throughout the project, the end goal was always to have an application that could be distributed and used in order to gain insight into its feasibility as a production application. We made many contributions to user interface design and into usability. Additionally, we went through all aspects of moving an application from a development environment into an end-user environment such as producing a standalone executable and creating an installation process that anyone would be able to use.

6.1.3 Future of the Project

This project is an ongoing endeavor and development will continue after our departure from the project center. While the prototype is essentially built, its ultimate outcome has not yet been determined and there are several possible options.

Currently it is planned that a smaller team within the eBay Research Labs will continue work on the project until it is stable and functional enough for wide distribution. This distribution may come in the form of an internal test, or could end up being a public beta. Once the prototype has been refined enough, it could be used as part of the launch of the eBay Research Labs division into eBay (as a demonstration of the purpose of the labs).

There are several other possibilities for the future of the project. One of these possibilities would be for the project to be released into the eBay developer community code-base. Introducing it to the public developer community would theoretically result in a much larger development team working on the project. Alternatively, it maybe possible that eBay's growing development team in China may find interest in the project and pick it up.

In any event, the end goal is for eBay upper-management to see the potential of the concept and provide full funding and approval to bring it into production. While the ultimate future of the application is out of our hands, we are anxious to see our prototype realized as a production level product. Whether eBay decides to fund the production development or not, the concept is one that we feel will not go away and will most likely surface one way or another. Our ten week implementation only begins to scratch the surface of its potential.

7.0 References

1: Internet Story, http://www.internet-story.com/ebay.htm
Accessed: 11/30/05

2: Brandeis University,
http://www.cs.brandeis.edu/~magnus/ief248a/eBay/history.html
Accessed: 11/21/05

3. eBay Investor Relations Website
http://investor.ebay.com

5: W3C, http://www.w3.org/TR/soap/
Accessd: 11/20/05

6: Ebay Codebase, http://developer.ebay.com/
Accessed: 11/15/05

7: http://sunra.lbl.gov/rtml3/dictionary.html
Accessed: 11/16/05

8: Kulnis And Co, http://www.kulnisandco.com/terms.htm
Accessed: 11/26/05

9: Skype Security Evaluation, http://www.skype.com/security/files/2005-031%20security%20evaluation.pdf
Accessed: 11/28/05

10: Skype, http://www.skype.com/
Accessed: 11/28/05

11: Skype API 2.0 Reference,
http://share.skype.com/sites/devzone/2006/02/skype_20_api_reference.html