Samariteen AIM Hotline

An Interactive Qualifying Project
submitted to the faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
degree of Bachelor of Science

by
Elliot Pennington

Date:
12 October 2009

Report Submitted to:
Gary F. Pollice
Worcester Polytechnic Institute

Abstract

The Samaritans of Boston are an organization which provides anonymous telephone council for suicidal individuals. The Samaritans organization have been seeking a way to expand their hot-line services from only a telephone hot-line to a service that would feature a computer instant messaging hot-line in addition to the telephone service.

The Samariteen IQP project aimed to build the tool that would provide hot-line functionality for instant messaging over the Internet. The project was generally a success, though not all of the desired features were implemented, the basic hot-line technology is functional.

Contents

1 Executive Summary

The first step of building a software product is always to analyze the existing requirements very carefully. For this project, not all of the requirements were initially clear.

Originally, The Samaritans were unsure whether they wanted a system to handle text messaging from cell phones, or if they wanted a system to handle instant messaging from computers over the Internet. My team consulted an IQP that had been completed the previous year investigating which of those two options would be the most effective. That project showed that a product built around instant messaging would be more popular than would a product for cell phone text messaging.[1]

With the main idea of the project established, we needed to collect the particulars. We had a meeting with some representatives from the Samaritans organization who were able to help us understand exactly what it was that they wanted the tool to be able to do.

The basic structure of the product required a way for multiple Samaritan representatives to have multiple independent conversations with clients behind the same screen name. All of the conversations needed to be able to be recorded into a history database. In addition, none of the real incoming client screen names could be visible to the Samaritan representatives; it needed to be completely anonymous.

[1]Nguyen, Tuong-Vi (2007). *Samaritans' Teen Line.* Unpublished Interactive Qualifying Project. Worcester Polytechnic Institute.

With the requirements mostly clear, we started to prioritize. The major backbone of the project is the basic technology for an AIM hot-line. The additional features, such as history and anonymity, could be built in later, so long as the backbone provided a flexible infrastructure for expansion.

We realized that we actually needed two separate applications for this product. One is a server application that connects to the AIM network and accepts incoming instant messages from the Internet. One is a client application that the Samaritan representatives can run on their computers to connect to the server application. We chose to build the applications with the Python programming language. There existed some very simple AIM libraries for Python that provided functionality for connecting to AIM and sending and receiving instant messages.

Since the client application would be useless unless it could connect to the server application, I took on the task of building the server first. It needed a fairly large amount of concurrent functionality. I implemented the concurrency with built in Python threads (rather than OS level threads), which are not the best for performance, but they do provide the necessary functionality.

The server application needed to perform a number of tasks simultaneously. It needed to wait for incoming AIM messages, and decide what to do with them. It needed to listen for connecting Samaritan representatives. It needed to route messages from connected Samaritan representatives to the correct clients. It needed to have a flexible enough infrastructure so as to allow expansion for screen name encryption and history. The server also needed to be able to support centralized settings for specific users. Not all users (Samaritan representatives) will have the same permissions on the system.

The client application also needed a lot of concurrent functionality, though it was initially clear that this was the case. We needed to be able to allow the users to log into the server such that they would be registered to receive messages from clients. The client application needed to be able to carry on an arbitrary number of conversations each in their own window, and also be able to listen for new conversations. It turned out that a lot of threading was needed even for just a single conversation window, largely because of the way that the graphics work.

We did encounter a major setback during the construction of the basic technology. One of the reasons that we chose to use Python was due to it's multiplatform portability. Python works on Windows, Linux, Solaris, and Mac OS. Unfortunately, we got stuck on one of the few unavoidable portability issues. All of the initial testing of the client application graphics was done on an Apple computer running Mac OS. When we moved the software to Windows for testing, the interface wouldn't work. It took extensive debugging to diagnose the issue. We in denial for a long time; we did not want it to be a portability bug. It would have been easier if it was a problem in our code that we could have fixed. We wound up having to create a new solution in the form of a tabbed window for multiple conversations.

We did succeed in our quest to build the basic hot-line technology. Multiple users can log into the servers and have multiple independent conversations with clients. Unfortunately, due to our setbacks with the client application, we did not finish all of the additional requested features.

2 Introduction

Saving lives was the fundamental purpose of the Samariteen project. The Samaritans organization has provided anonymous telephone counseling services since 1974. Their telephone hot-line runs twenty four hours a day, seven days a week. They also run a special hot-line for teenage callers called the Samariteen help line. This is a special line staffed by teenagers. The Samaritans were seeking a means by which to popularize their Samariteen line. A previous IQP did research that indicated that a change in technology would yield better results. That IQP presented results to show that a hot-line to support computer instant messaging over the Internet would probably see the greatest increase in usage among teenagers. [2]

The Samariteen IQP project handled, along with the members of the Samariteen MQP, the construction of the software tool to support an instant messaging hot-line service. This report will detail how and we built the tool the way we did, along with the problems we encountered during the process, and how we solved them.

[2]Nguyen, Tuong-Vi (2007). *Samaritans' Teen Line.* Unpublished Interactive Qualifying Project. Worcester Polytechnic Institute.

3 Methodology

The methodology of software engineering projects has been a hotly debated topic for a long time. We attempted to be somewhat agile in our approach to the project, in that we developed our product in such a way as to accept and welcome changes in the requirements during the development process. Sometimes, people will build a product with methodologies that are very inflexible, and when requirements change, large quantities of the code have to be rewritten to accommodate the alterations necessary to fulfill the new requirements. We wanted to avoid having that sort of problem. We were somewhat successful.

3.1 Requirements Analysis

Requirements analysis is a very necessary stage of any software project. If a team gets the requirements wrong, the customer will not be pleased. We met with some people from the Samaritans of Boston and they explained to us what they wanted. For certain aspects of the requirements, they were somewhat unsure of what they wanted. It took quite a while for them to decide on how accessible they wanted the chat transcripts to be, for example.

We eventually did flush out exactly what our tool needed to do. We broke the features down into specific tasks. This is not always a trivial process. It takes a fair amount of experience to be able to look at a requirement from a user and break that into specific technical tasks that need to be operational for that feature to work.

For example, they had wanted transcripts saved for each chat. This required a database to be set up with the fields to store all the information, connected to the central server application. It also required a graphical front end for users so that they could actually view the history that had been stored. It required a message passing system between the client and server so that the server can parse commands separately from outgoing messages. It required a query builder to parse the commands from the client application into queries for the database, and a means to return the results to the user application. So, one feature requirement can easily mean a large number of tasks.

Our team set up a database of the requirements for our project so that we could keep track of everything. We used the WPI Sourceforge tool to help us organize everything. Once we had all this figured out, it was time to start delving into the specifics.

3.2 Order Of Operations

When a computer receives messages over the Internet, it doesn't automatically know what to do with them or how to display the data. It has to run a bunch of checks on the received message to figure out what it is and where it goes. The order in which these checks are performed were particularly relevant for our project, because if something was checked in the wrong place, the whole thing might not work.

When our AIM server receives an instant message, it just receives a big string of characters. From that string, we need to parse out specific information, most notably: the screen name that sent the message, and the message itself. Our system then needs to encrypt the incoming screen name immediately, and check in some software table to see if any connected Samaritan representative is having a conversation with the client associated with the encrypted screen name of the incoming message. If so, that message needs to be routed directly to that Samaritan representative across the network. If not, it must be a new message, and a separate algorithm must be run to determine to whom of the connected Samaritan representatives to send the new message. A default message must be sent if no Samaritan representatives are connected to the server. All of these cases must be checked for each message received by the system.

A similar process occurs in the client application. The client application receives messages from the server, but it needs to check to see if the message is for a new conversation, in which case a new window needs to be spawned, or if it is for an existing conversation, in which case the message needs to be appended to the appropriate window.

3.3 Unexpected Problems

Unexpected problems are something that anyone writing software learns to expect. The trick is to anticipate things to break, and write code in a way that allows for easy modification, so that the problem can be fixed quickly.

We did encounter one major unexpected problem during our project. It set us back for a long time, not so much because we were unable to fix it, but because it took a long time to determine the root cause of the issue.

The client application naturally uses a graphical toolkit to display the conversations and to allow the users to enter text to send to their clients. Originally, we had each conversation running in a separate window. All of the original AIM clients from AOL used this strategy as well. This functionality, with a new window appearing for each conversation, was built primarily on my personal computer, which incidentally is an Apple computer running the Macintosh Operating System. The graphical toolkit we used to display the windows is a cross-platform package called Tk. Tk is a mature product and has been thoroughly tested on all of the major systems: Linux, Windows, Mac, Solaris. So we knew that any Tk code that worked on my Macintosh would also work on any Windows computer.

And this turned out to be true. Tk is portable. Unfortunately, the way processes are controlled in the Windows operating system makes it impossible to listen for events on multiple graphical windows simultaneously. The amount of debugging required to figure this out sapped a catastrophic amount of time from our project. In addition to this problem, there were assorted synchronization problems that had to be fixed before the portability issue could even be considered as a possible bug. Dealing with thread synchronization with infinite loops is famously difficult, and sometimes impossible. Debugging software does not handle thread scheduling well, which adds further difficulty to the process.

Eventually, once the synchronization issues were solved, and the program still did not work on Windows, we were left with few places to turn for a solution. A few more steps through the debugger led us to realize that the true problem came from differences in the way the operating systems implement their process control, or, at the very least, in the way the two different versions of Tk (Windows vs. Mac) are programmed to work with the operating system process control.

Luckily, we knew our threading system still worked, so we were able to come up with a fix. We simply implemented tabbed windows. This way, each new conversation would open in a new tab in the same window as opposed to a completely different window. Tk does not support tabbed windows, but it was possible to hack up an implementation with radio buttons. With this change, we put our same old threading and messaging mechanism underneath the graphical layer for a functional tabbed interface. Because of our flexible design, we were able to make a significant change to the program without needing to change much code.

4 Results

By the end of the project period we had developed a fairly robust instant messaging hot-line system. The server program could receive instant messages using the AIM protocol, distribute them to a new logged in Samaritan representative the appropriate Samaritan representative (in the case of an existing conversation). The server application has database connectivity functionality built in, along with infrastructure for clean implementation of database query commands. However, no real database functionality has been tested with our system. The server accepts login requests from Samaritan representatives running our client application, and serves them messages. The server can also send out messages received from the client application.

The client program can receive messages from the server and either display them as part of an existing conversation or display them in a tab for a new conversation. The client program also accepts typed messages from the user and sends them to the appropriate screen name through the server.

5 Conclusions

By following good software engineering practices, we were able to develop a functional tool with the basic required technology. Strong requirements analysis prevented us from wasting time implementing features incorrectly, or even implementing incorrect features. We correctly implemented the technology with careful attention to the specific operations that needed to occur, and by building flexible software with interchangeable components. Our robust framework allowed us to solve the portability issue, though we were detained by that quagmire for a long time.

Additional features do need to be implemented before the system can be put into use by the Samaritans of Boston. A proper database needs to be installed, and an engine and an interface need to be built for both searching the database and for changing system settings and user permissions. Once those items are complete, the system will be functional as required.