University of Glasgow | Department of Computing Science

# GIM - Team G Instant Messenger

Ewan Baird
Heather Hoaglund-Biron
Gordon Martin
James McMinn

Level 3 Project — 28 March 2011

## Abstract

The abstract goes here

## Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

| | | | |
|---|---|---|---|
| Name: | | Signature: | |
| Name: | | Signature: | |
| Name: | | Signature: | |
| Name: | | Signature: | |
| Name: | | Signature: | |
| Name: | | Signature: | |

# Contents

# Chapter 1

# Introduction

## 1.1 Introduction

Instant messengers are an easy, fast way to communicate over the Internet. Basic instant messengers allow users to type messages to each other on separate computers and have those messages immediately show on the screen, like instant email. More advanced systems have features like audio and video communication. These days instant messengers are becoming more popular, given the way the Internet is helping people from all over the world connect with one another. This has led to an increased demand for quick and easy communication.

These days, it's difficult to be original when developing an instant messenger. Programs like Skype, AIM and MSN Messenger are popular, especially with the younger generation. There are many kinds of instant messengers out there, the most prominent of which are always competing against each other to have the newest and most intriguing features.

This project is not about joining that competition. Instead, we are using this opportunity to explore what it's like to create an instant messenger from the ground up. As we are part of the younger generation, we use instant messengers almost daily, and it's interesting to be able to go behind the scenes and discover how they work for ourselves.

The instant messenger model that we decided to use consists of a server, any number of clients, and a set of rules defining how the server and clients talk to each other, called a protocol. A user of the program is essentially interacting with a client, and the clients interact with each other through the server. That communication is structured using the protocol.

Instead of using the protocol and server of an existing instant messenger and focusing on the client, we decided to create the entire system ourselves. This way we're able to learn more about the whole process of creating such a program. We have divided ourselves into two smaller groups: two of us to do the networking and make the server, and the other two to make the client, including the user interface, or UI.

This report is meant for the Computing Science professors at Glasgow University, fellow Computing Science students, and anyone with an interest in instant messaging or the process of completing a large-scale Computing Science project.

The structure of this report is as follows. In Chapter 1, we have this introduction, the problem definition, and an overview of the requirements. Chapter 2 covers the design of the protocol, client, server, and overall networking. In Chapter 3 is a discussion of our implementation of the client and server. Chapter 4 covers evaluation of our finished product, and Chapter 5 is a conclusion and reflective of what we learned.

## 1.2 Problem Definition

The following was our given problem definition[1]:

> Instant messaging systems such as Jabber, AIM, ICQ and IRC have become popular in the last few years. The aim of the project is to build a simple instant messaging system, written in Java, to develop your understanding of networked systems and programming.
>
> The group will be required to develop both an instant messaging client and server, and to design the network protocol they use to communicate. The server should accept connections from an arbitrary number of clients. Clients will have a graphical user interface, and should be able to accept and receive text-based messages, to indicate if the user is busy, available or idle, and to convey usernames and other details.
>
> The project involves network and user-interface programming. It would suit a group with an interest in low-level systems design and implementation issues.

Despite the fact the above problem definition says we have to develop the client, server, and protocol, we were told we could also construct the system using the protocol of another existing instant messenger, like IRC or MSN Messenger. In that case, we would focus more on the development of the client. Instead, we opted to create the whole thing ourselves, since it would give us more control and we would be able to learn more about the entire process of creating an instant messenger.

The problem definition also states that we should at least have text-based messages, statuses (online, offline, etc), and usernames. We decided to expand on this, including smileys (small pictures) in messages, a personal message and display picture for each user, and more.

It wasnt explicitly stated, but we had to decide how to structure the users: whether they should have "friends" or "contacts," whether it should be chatroom-style, and whether there should be one-on-one conversations, group conversations, or both. We chose to do a combination, in that users may have contacts and one-on-one conversations, but may also join, create, and invite other users to group chatrooms.

## 1.3 Requirements

Our first task was to establish what format our application would take. We had two main choices which would dictate how the project would procede:

[1]`http://fims.moodle.gla.ac.uk/file.php/129/1243.pdf`

1. One-to-One - A similar style to Windows Live Messenger or AIM where users have a contact list of friends and typical conversations take place between two clients.
2. Multicast - IRC is the most prominent example of multicast instant messenging. Users connect to a server then join a room within that server. Rooms can have many users and will persist regardless of active users.

Within these basic frameworks, we also had the option to merge aspects from each, for example basing a one-to-one application on the IRC protocol.

Ultimately it was decided that we would create a one-to-one messenger program, with a consideration towards group conversations.

### 1.3.1 Conception of Features

Once we had established that we were going to use the one-to-one paradigm, the nest task was to determine what we beleived to be the important features of an instant messenger, and what was acheivable within the scope of the project. This process involved several team meetings where we simply discussed our experience with a variety of programs and picked areas where we wished to draw from. Due to the popularity of instant messenger programs, they have undergone constant evolution, and continue to do so. Some of our work had already been done.

The constant iteration of instant messenger interfaces provides us with a solid foundation with which to base our client GUI. Our experience of these programs allowed us to highlight features which we felt were acheivable and, more importantly, useful to users. We conceived a feature set split into 4 categories of importance using the MoSCoW method.

**Must Have**

- Send Messages
- Graphical User Interface
- User Nicknames
- Contact list
- User Status

**Should Have**

- URL Parsing
- Display Pictures
- File Transfers
- Personal Messages
- Smilies

**Could Have**

- User Profile
- Custom Commands
- Themes
- Plug-in Support
- VoIP

**Would Like To Have**

- Contact List Grouping
- Offline Messaging
- Chat Logging
- Custom Fonts and Colours

This list was decided upon by taking into account what we belived to be each features' necessity, usefulness, and diffiulty of implementation. The requirements in the 'Must Have' category were taken from the initial problem specification, and the other categories were decided using the criteria described previously.

The 'Must Have' features generally contain the basic elements of an instant messenger, such as sending messages and a graphical user interface. Sending messages, a GUI, and user statuses were taken from the problem definition, however we felt it was crucial and within the scope of this project to include contact lists and user nicknames as must have features.

Should Have features are those which we felt were within the scope of the project and would significantly enhance the users' experience. URL parsing is the ability for user to select hyperlinks in the chat window. This was given high priority due to our experience of using other IM clients, which often involves sending contacts links to various websites. Display pictures are images that a user uses to represent themselves with to their contacts. While display pictures do not directly impact the functionality of the program, we felt that they would make the chatting experience more personal and users may expect to see what has been a standard feature of similar programs for some time. We considered the ability to send files between users to be a useful feature but were are that it would potentially be one of the most difficult items on the list to implement. Personal messages are one of the easier features on the list. A personal message is a small message a user sets on the interface that all other users can see, typically underneath the username and given less prominence. As this was considered to be simple to implement, it was assigned a relativly high priority. Smilies (also known as emoticons) are small icons used to represent emotions in chat. While they add visual appeal, smilies would not add significant functionality as text-based representations can be used.

Many of the could have features involve customisation of the interface. User profiles are pages in the interface which would contain details on that user which can be viewed by contacts. Beyond the basic concept we had not decided how these would function or what they would contain. Custom commands would act in a similar way to an IRC client; commands such as '/whois' or '/join'. As these were custom commands, the user would be able to set their own keywords for a set of possible commands. Themes are another feature to further customisation. A theme is a preset of colours used to alter the interfaces' look and feel. Two of the more demanding requirements we conceived were ”Plug-in Support” and VoIP. This is the ability for other developers to create additional features which can be slotted in to certian areas, for example support for other messenger protocols. VoIP (Voice over Internet Protocol) is a protocol designed to accomidate voice chat between clients. While this is a very desierable feature it was generally considered to be out with the scope of the project, but we decided to include it so that a decision could be made during the implementation phase depending on our progress.

Features which we would like to have are those which are low priority, but somewhat desirable. Contact list grouping is where the user is able to create subsets of their contacts to make managing large contact lists easier. This would likely be included in the form of displaying Online or Offline contacts, however this feature represents capability beyond that basic functionality, such as user-defined custom groups. Offline messaging allows users to send messages to each other regardless of their status. When a user logs into the system, they receive any messages that were sent to them while they were not logged in. This is useful for short disconnects to prevent messages being lost. Chat logging is the ability to store conversations between users, while this feature can be useful, it is generally not used often by users, hence it's low priority. Custom fonts and colours are related to themes, but would provide a deeper level of customisation. Themes were given a higher priority because they were considered easier to implement and would provide an adequate level of customisation for most users.

**Rejected Features**

We decided to aim the application at users who are comfortable with general computer use. As a result there were features which we agreed were not desierable or worth our time implementing.

- Nudges/Winks
- Audio Cues
- Spell Checking
- URL Warnings

Nudges are events which one user sends to another to force their chat window into focus. These were considered by many to be an irritation which is prone to abuse, hence our desicion to disclude them. Winks are a similar feature with more multi-media. Spell Checking was considered, but due to the typically informal nature of IM conversations, it was not included. URL warnings are quick pop-ups that appear whenever a user clicks on a hyperlink to warn them that content may be harmful. As we has made the assumption that users will have a basic level of competance with computers, this feature was not included.

# Chapter 2

# Design

## 2.1 Protocol Design

### 2.1.1 Protocol Overview

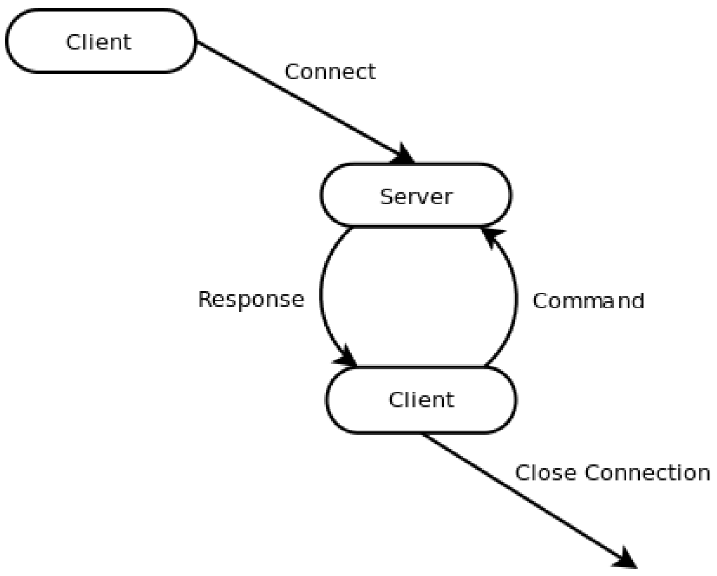

Figure 2.1: The exchange of messages between a client and the server

Gim uses a client-server architecture, where one computer (known as the Sever) acts as a central point to which other computers (the Clients) connect. The clients do not communicate directly with each other and all communication takes place between the clients and the server. If a client wishes to send a message to another client it must first got through the server.

In GIM, the Protocol is responsible for enabling communication between the clients and the server in a reliable and consistent manner. A protocol is a set of rules that determine the format and transmission of data between computers. The syntax (the structure, or format) and semantics (the meaning) of the Protocol are discussed in the next section.

At the highest level of abstraction, the GIM Protocol works in a very simple manner. A client connects to the sever and they exchange messages until the connection is closed, as shown in Figure 2.1.

In practice there are several clients connected to the server at once, however as the clients do not directly interact with each other and do not need to know about each other, the entire system can be described as above.

### 2.1.2 Protocol Specification

#### Syntax

The GIM Protocol uses a simple, text based syntax. Each full command begins with a colon followed by a command name and its arguments. A second colon marks the end of the headers and beginning of the data segment. A semi-colon marks the end of the data segment and the termination of the command. The basic structure of a full command is shown below:

```
:<COMMAND_NAME> <ARGUMENTS>:  <DATA>;
```

Commands names are predefined and `<COMMAND_NAME>` can be any 1 of the following 19 values:

| | | | |
|---|---|---|---|
| AUTH | FRIENDREQUEST | MESSAGE | ROOM |
| BROADCAST | GET | OKAY | SERVERSTATUS |
| ERROR | INFO | PING | SET |
| FRIEND | KILL | PONG | UPDATE |
| FRIENDLIST | LOGOUT | QUIT | |

`<ARGUMENTS>` is a set of zero or more predefined arguments which can be passed to the command in order to change its behaviour. Each command has its own set of arguments, some commands require arguments, some commands accept more than one argument and some commands have no arguments. The protocol specification defines over 60 arguments and a full listing for each command can be found in the *GIM Protocol Specification Document*.

Unlike the command and argument segments, the `<DATA>` segment does not have any predefined values and has a varying syntax across different commands. For example, the `ROOM` command has `JOIN` and `LEAVE` arguments which are used to join or leave a chat. The majority of the data is likely to have been provided by the user at some point, and as such the data segment is considered to be “unsafe”. This data is encoded (See *Encoding, Limits and Restrictions* for information about how the data is encoded). A full specification for the data segment of each command can be found in the *GIM Protocol Specification Document*.

For example, the following is an example of a command sent from the client to the server requesting the Nickname and Status of the users joe@example.com and bill@gmail.org:

```
:GET NICKNAME STATUS: joe\U+0040example.com bill\U+0040mail.org;
```

To which the server may reply:

```
:INFO NICKNAME STATUS: joe\U+0040example.com
Jeo
ONLINE
bill\U+0040mail.org
Billy The Kid
AWAY;
```

*(Please note that the data segments is the previous examples have been encoded defined below in Encoding, Limits and Restrictions.)*

## Roles and States

The protocol defines two separate roles: The Client role and the Server role. Each role is only able to send a specific subset of commands and receive all of the commands sent by the opposite role (i.e. the server must understand all of the commands which the client is able to send and vice-versa). This means that both the client and server must implement all commands.

Having clearly defined roles is an integral part of the GIM Protocol. This allows the semantics of a command to be used to generate a response in cases where both roles are able to send and receive a command but where the command has a different meaning depending on its source.

For example, the `AUTH` command can be sent by both the client and the server roles. When sent by the server it is used to indicate the current state of the client (Either Logged-in or Unauthorized) but when sent by the client it indicates that they wish to login or register a new account. This means that the same command must be implemented differently depending on the role of the sender.

As mentioned above, the client role has 2 states: Logged-in and Unauthorized. When the client first establishes a connection it is placed in the unauthorized state. This means that it has access to an even smaller subset of commands, only those essential to logging-in or registering a new account. Once the client has successfully logged-in it is granted access to the full set of client commands.

## Encoding, Limits and Restrictions

Any non letter or digit (anything not a-Z or 0-9) in the data segment of a command is encoded and replaced by `\U+` followed by a 4 digit number representing the UTF-16 code for the character. For example, the @ character in the previous example is encoded as `\U+0040`. The only exception to this rule a character which is used to format or separate different chunks of data. In the GIM Protocol this is always either a single space character (the equivalent of `\U+0020`) or a newline character (`\n`).

Display pictures, which are small images in JPEG format, cannot be transferred easily using the GIM Protocol without first converting them to a text based format. JPEG and any other binary data is first encoded using BASE64 before being sent using the protocol.

This makes the protocol very easy to parse as we can safely split the data segment into separate parts without worrying about how data from the user is formatted. This means that we can guarantee every occurrence of a white-space character in the data segment is safe to use for splitting the data.

Clients connected to the server are limited to sending 100 commands in rolling 5 second window. Normal usage should result in a much lower rate (less than 5 per second) and anything more than than this most likely indicates a malfunctioning or malicious client. In the event that a client exceeds this limit then the server closes the connection with the client.

The data segment of commands are limited to 8192 characters in length. This includes data that has been encoded. The exception to this rule is the `:SET DISPLAY_PIC:` command, which is used to set a users display picture. The limit for this command is 4x greater than that of a normal command (32,768 characters).

### 2.1.3 Protocol Evolution

As the project

### 2.1.4 Possible Future Changes

## 2.2 Client Design

This section will detail the process of designing the GIM client; the application used the communicate with the server.

### 2.2.1 GUI Structure

### 2.2.2 Design of Client Structure

[intro, with some context goes here (?)]

#### Approach

In designing the client, we had the high level aim of creating a modular system to allow team members to take responsibility for certain aspects of the system. From an early stage in the design process, we were aware that the client had a large set of responsibilities. We determined these responsibilities to be; to maintain a connection with the server, implement the GIM protocol, provide a user interface to the system and keep a record of up to date information about the users friends. Our first task was to design a set of interacting sub-components to handle these responsibilities.

It was agreed that the MVC (Model-Viewer-Controller) architectural pattern, widely used for applications involving a graphical user interface, was a useful model to base our discussions around. This model abstracts the UI (view) from the back end of the system. When a user performs an action, the controller updates a collection of data associated with the system, and updates the interface to reflect any changes the user needs to know about. This seemed appropriate to our needs of keeping and displaying an up to record about the users friend list and creating an interface, and would allow us to split responsibilities between the back and front end of the GUI.

However, we faced challenges in adapting an additional networking component, to implement the GIM protocol, into this framework. We had the choice to conceptually view it as either an additional interface, which viewed changes to the model, or indirectly (by way of the network) the controller component. [discussion about possible merits of both goes here] We decided it would be useful to treat the networking code as part of the controller, as it would be modifying the model based on the servers response to its calls, and modifying the GUI to reflect these changes.

This high level approach allowed us to draw high level boundaries between components, and assign the more detailed design of individual components to team members. We decided a logical split would be to assign two members to the networking aspects of the system (client and server), and two members to the GUI components of the system (the GUI and its model.) However, further collaboration was required as it became more evident what was required from each component, from other components, as will be discussed.

[Diagram goes here, reflecting the above discussion (???) ]

### Model-Viewer-Controller Structural Concerns

Controller

The controller was one of the more challenging parts of the design process. The controller would be called by both the networking component, and the viewer, implying that its operations must be thread safe. We were constrained by our use of the swing environment, which we had learned from previous experience, will freeze if the networking code is ran on the thread running the GUI. A further complexity was handling the boundary between the requirements of the network code, and the protocol that was being handled by the networking code. Naturally, the GUI architects should not need to concern themselves with the specifics of the protocol, while designing the GUI.

To deal with the threading complexity, we designed a scheme whereby there would be an intermediate buffer between the controller component, and the networking code. The controller thread would wait until there was a command from the network placed onto the buffer, and act accordingly on the model and GUI. Furthermore, when the controller component needed to call the network, it would place a command onto the buffer, to be interpreted by the networking code. We believed this scheme to be appropriate as controller could either call the network from internal code, or events from the GUI (possibly simultaneously).

To deal with the design complexity of the boundary, we decided that initially, an interface would be written by the networking architects, including javadoc, to describe the requirements of the GIM protocol of the client, from the controller. In the implementation, the controller would then implement this interface. This allowed the requirements from the client to be understood, without having to understand the inner workings of the protocol. Furthermore, the structure of the protocol could change, without any change to the interface or consultation with the GUI architects. The networking architect would write code to parse incoming commands to calls these methods described by the interface, and write outgoing commands from the GUI.

[there's a diagram on google docs - will ask someone how to put it here later] [note: Somebody will make a better, possibly UML diagram]

[NOTE: WRITE IN IMPLENTATION SECTION, THIS DIDNT HAPPEN. (okay, lower case time)... Instead, we put things onto swings event queue, and only used the buffer for outgoing traffic. We decided the parsing code was best placed in the network reading thread, to keep the code tidy. Furthermore, this hid the protocol code from the GUI more...]

[Picture of flow of information between buffer, controller, and network.]

Model

The design of the model involved determining what state information had to be held about the client. This was informed by the previous work of requirements analysis and the conception of features. The model was also informed by the design of the viewer.

Two must have of the system were the presence of a contact list, and support for user statuses. This implied that there should be data structures for the maintenance of a contact list, and the statuses of users. To deal with this requirement, we designed the model to have a data structure to maintain information about users, including their status, nickname and personal messages.

The viewer required that the users own state information (such as status, and nickname) were displayed. As these were evident in multiple windows in the viewer design (such as in chat windows, and on the buddy list), the model was designed to maintain a record of the users current status.

Viewer

[Reference GUI design ? -¿ that section comes next... ]

## 2.3 Server Design

blah blah blah

## 2.4 Networking Design

blah blah blah

# Chapter 3

# Implementation

## 3.1 Client

blah blah blah

## 3.2 Server

blah blah blah

## 3.3 Client-Server Communication

blah blah blah

## 3.4 Storage

blah blah blah

# Chapter 4

# Evaluation

## 4.1 Server Evaluation

blah blah blah

## 4.2 Client Evaluation

blah blah blah

## 4.3 User Evaluation

blah blah blah

# Chapter 5

# Conclusion

## 5.1 Contributions

blah blah blah

## 5.2 Reflective

blah blah blah

# Appendix A

# Appendices