Implementing an IRC Server Using an Object-Oriented Programming Model for Concurrency

Bachelor Thesis

Fabian Gremper
ETH Zürich
fgremper@student.ethz.ch

May 17, 2011 – September 25, 2011

Supervised by:
Scott West
Prof. Dr. Betrand Meyer

Table of Contents

1 Introduction

The Internet Relay Chat (IRC) is a form of real-time Internet text messaging. It was one of the most popular chat protocols in the early stages of the Internet. Most of the popular IRC servers (also called IRC daemon) were therefore not written with object-oriented principles in mind.

This bachelor thesis focuses on writing a new IRC server application from scratch, using SCOOP as a programming model for concurrency. The application is written in Eiffel and provides a working subset of the functionality documented in the IRC protocol specification in RFC1459.

The following functionality is provided:

- The server follows the RFC1459 protocol and works with common IRC clients.
- The server allows many users to connect simultaneously and interact with each other directly or using channels.
- Channels support channel modes +ismnt and allow setting topics.
- Users can gain voice and operator privileges in channels.
- Users with operator privileges can alter the privileges of users in a channel and kick users from a channel.
- Users can retrieve information about the available channels and other users on the server.
- The server pings its clients periodically in order to identify timed out clients.

The application has been developed and tested with EiffelStudio 6.8 and mIRC 7.19 on Windows XP (32-bit).

1.1 Related Work

Florian Besser has written a bachelor thesis about using SCOOP to implement an HTTP server. However, at the time of writing his thesis isn't available to me.

1.2 Outline

A quick introduction to SCOOP can be found in Chapter 2.

Chapter 3 deals with the implementation of the IRC server. The structure and classes of the application are presented and SCOOP related aspects of the implementation are highlighted. The development process is evaluated with respect to major milestones of the project.

A small user and developers guide is presented in Chapter 4. Unimplemented functionality of the IRC specification is listed and can be included in future work.

Chapter 5 contains a summary of my experience with SCOOP.

2 SCOOP

SCOOP (Simple Concurrent Object-Oriented Programming) [2] is a model and practical framework for building concurrent applications. It comes as a refinement of the Eiffel [1] programming language and is integrated into the latest version of EiffelStudio.

In the context of SCOOP, every object is handled by a *processor*. A processor is an abstract notion of an autonomous thread that is capable of the sequential execution of one or more objects.

SCOOP also introduces the notion of separateness. Declaring a type with the keyword *separate* results in the object being allocated to a new *processor*. This is called a *separate type*.

```
        my_type: separate SOME_TYPE
```

Calling a routine of a *separate type* is only possible using a *separate call*. A separate call wraps a feature call of a separate type in an enclosing routine. Execution of this enclosing routine is delayed until the processor handling the separate argument is available for exclusive access.

```
    calling_routine
        do
            enclosing_routine (my_separate_attribute)
        end

    enclosing_routine (a_arg: separate SOME_TYPE)
        do
            a_arg.some_feature
        end
```

The calling routine continues its execution and does not wait for the execution of the enclosing routine to finish.

3 Implementation

In this chapter we will look at the implementation of the IRC server.

3.1 IRC protocol

3.1.1 Structure

This section gives you a quick overview of how the IRC protocol works.

In a basic scenario there is one IRC server and many IRC clients that connect to it. Every client has a nickname which other clients use to refer to it. Clients can interact with each other directly or using channels.

Channels are the basic means of communicating to a group of users in an established IRC session. A channel has a channel name that is prefixed with "#" and a keeps list of users that are currently in the channel. Channels have can have a topic and several modes set. Users can join a channel using the *JOIN* command. A channel is created implicitly on *JOIN* if it does not exist, and deleted after the last user leaves.

3.1.2 Messages

The communication between IRC server and client is implemented using plain-text single-line messages. The format of the message is described in 'pseudo' BNF (Backus–Naur Form) in RFC1945 [3]:

```
<message>  ::= [':' <prefix> <SPACE> ] <command> <params> <crlf>
<prefix>   ::= <servername> | <nick> [ '!' <user> ] [ '@' <host> ]
<command>  ::= <letter> { <letter> } | <number> <number> <number>
<SPACE>    ::= ' ' { ' ' }
<params>   ::= <SPACE> [ ':' <trailing> | <middle> <params> ]

<middle>   ::= <Any *non-empty* sequence of octets not including SPACE
               or NUL or CR or LF, the first of which may not be ':'>
<trailing> ::= <Any, possibly *empty*, sequence of octets not including
               NUL or CR or LF>

<crlf>     ::= CR LF
```

The prefix indicates the origin of the message. This can be the server or another client. Prefixes are only included in server-to-client messages.

The command is either plain-text (such as *JOIN*) or a 3-digit number (e.g. 331, which represents *RPL_NOTOPIC*).

Parameters may not contain *NUL*, *CR* (carriage return), *LF* (line feed), space or be empty. The trailing parameter can be empty or contain spaces.

Example commands:

JOIN #bar	User wants to join channel #bar.
:foo JOIN #bar	User *foo* has joined channel *#bar*.
:foo TOPIC #bar :What's up?	User *foo* has changed the topic of channel *#bar* to "What's up?"

3.2 General Design

The `APPLICATION` opens up a listening socket on port 6667. Connecting clients will be accepted and a new socket is returned for every client. An object of class `IRC_USER` will be created for every client and handle further communication with the associated socket. Incoming messages from the client will be parsed and effective commands will be handled by a global object of class `CONTEXT`.

The following is an overview of all classes:

- **`APPLICATION`** – Creates objects `CONTEXT` and `TIMEOUT_CHECK`. Listens for connections on port 6667 and creates an `IRC_USER` object that handles the resulting socket for every client.

- **`CONTEXT`** – Keeps track of all users and channels. Responsible for manipulation of user and channel data and sending out messages and command responses to clients.

- **`IRC_CHANNEL`** – Represents an IRC channel. Stores data such as the channel name, creation time, topic, channel modes and the users in the channel and their privileges (op, voice).

- **`IRC_USER`** – Represents an IRC client. `IRC_USER` objects read incoming data on the associated socket and parse the messages. Commands that affect multiple clients are passed to the `CONTEXT` and handled there.

- **`NETWORK_STREAM_SOCKET_HACK`** – This is a simple modification of `NETWORK_STREAM_SOCKET` that allows recreation of a socket using the internal file descriptor.

- **`TIMEOUT_CHECK`** – Periodically calls the `CONTEXT` to "ping" all connected users in order to find and remove timed-out clients.

Let's see an example of how and where commands are processed. In the following scenario, user *foo* is already connected and registered with the server and wants to join channel *#bar*.

	Action	**Class**
1)	The IRC client sends the following message: *JOIN #bar*	
2)	Read the message from the socket.	`IRC_USER`
3)	Parse the message.	`IRC_USER`
4)	Invoke *join_channel* of the `CONTEXT` object using a separate call.	`IRC_USER`
5)	Look up the `IRC_CHANNEL` object by name from the corresponding `HASH_TABLE`. If it does not exist, create a new `IRC_CHANNEL` and add it to the hash table.	`CONTEXT`
6)	If the user is already in the list of users of the `IRC_CHANNEL` or the channel is invite only, return an error.	`CONTEXT`
7)	Add the user to the list of users on the `IRC_CHANNEL`.	`CONTEXT`
8)	Notify all users on the channel about the joining user by sending the following message to the respective sockets: *:foo JOIN #bar*	`CONTEXT`

3.3 SCOOP Related Aspects

This section deals with SCOOP related aspects of the implementation.

3.3.1 Separate objects

Dealing with separate objects has been one of the greatest challenges I had to overcome dealing with SCOOP.

Assume you want to listen for incoming connections and then create a new separate object for every connection to handle each socket individually.

The basic listening socket setup looks like this:

```
        local
            in: NETWORK_STREAM_SOCKET
        do
            -- create listening socket
            create in.make_server_by_port (6667)
            in.listen (5)

            -- block and accept socket
            in.accept
```

You can now access the newly accepted socket through *in.accepted*. Let's assume we have another class SOCKETHANDLER which looks like this:

```
create
    make

feature
    socket: separate NETWORK_STREAM_SOCKET

    make (s: SOCKET)
        do
            socket := s
        end

    run
        do
            -- code involving the socket
            -- (possibly infinite reading loop)
        end
```

After accepting a socket in the main class, we first create an object of class SOCKETHANDLER and pass the accepted socket to it. Then we call the method *run* using a separate call.

Now the following problem occurs: the *run* routine cannot use the socket and execution of *run* is delayed forever, because the socket belongs the processor of APPLICATION, which is currently executing and not available for exclusive access. If we were to pass the socket to the *run* routine instead of the *make* routine, the main APPLICATION would wait until the execution of *run* finishes.

We could avoid this problem if the NETWORK_STREAM_SOCKET was non-separate in our SOCKETHANDLER class.

The solution: Instead of passing the socket itself, it is possible to pass an INTEGER_32 representing the internal file descriptor of the socket to the SOCKETHANDLER. We can then recreate the socket as a non-separate object in the SOCKETHANDLER using this integer.
The NETWORK_STREAM_SOCKET class does not provide a public creation routine that takes an internal file descriptor. However, a private routine *create_from_descriptor* does exactly what we're looking for. As you can see in section 3.2, NETWORK_STREAM_SOCKET_HACK is a simple class that inherits NETWORK_STREAM_SOCKET and makes the functionality to recreate a socket using the internal file descriptor publicly available.

Before my supervisor pointed me to this elegant solution, I was experimenting with another idea. Instead of listening for connections in the main APPLICATION, the listening is done in the SOCKETHANDLER. Initially, one separate SOCKETHANDLER object is created and it is listening for incoming connections. As soon as a client connects, it stops listening and deals with the socket instead. The APPLICATION is notified that the SOCKETHANDLER is no longer listening. This is done via a separate object that can be accessed by both APPLICATION and SOCKETHANDLER objects. A new SOCKETHANDLER object is now created and starts listening for incoming connections.

In a running state there are always *n* SOCKETHANDLER objects handling their respective sockets and one SOCKETHANDLER listening for new connections, where *n* is a number greater or equal to zero.

This solution also resulted in the sockets being non-separate in the SOCKETHANDLER and therefore the aforementioned problems can be avoided. However, apart from being far less elegant, this would also result in a very short period of time in which no listening socket exists after a client connects.

You can consider the class IRC_USER in the actual implementation to be an improved version of SOCKETHANDLER in this example.

Another difficulty was dealing with separate strings. Passing strings to a separate processor will turn them into separate strings, which are pretty much unusable, since the Eiffel libraries do not support separate types (yet).

The following simple routine recreates non-separate strings from separate strings by copying them character by character:

```
make_local_string (s: separate STRING): STRING
    local
        i: INTEGER
    do
        create Result.make_empty
        from
            i := 1
        until
            i > s.count
        loop
            Result.extend (s.item (i))
            i := i + 1
        end
    end
```

This auxiliary routine is used countless times throughout the entire project.

3.3.2 Execution Order

In a concurrent system, problems such as race conditions can arise when different processors, processes or threads share resources.

In this implementation, shared information such as user and channel data is all handled by a single object of class CONTEXT. Requests to access or change this data is also handled by the same object. Because separate calls wait until the processor of an object is available for exclusive access, the routines of the CONTEXT object pose an atomic operation in respect to the data that is manipulated. This guarantees that no unwanted behaviour, such as race conditions, can occur. The order of execution of the calls to the `CONTEXT` object by `IRC_USER` objects is merely scheduled by SCOOP.

3.3.3 Separate calls

Because most of the IRC commands parsed by the `IRC_USER` class are eventually routed to the `CONTEXT` object, there have to be a lot of enclosing routines (see Chapter 2) in the `IRC_USER` class that merely contain a one line call to an object with identical parameters.

An example is:

```
context_kick_from_channel (c: separate CONTEXT;
        fd: INTEGER_32; ch: STRING; n: STRING; s: STRING)
    do
        c.kick_from_channel (fd, ch, n, s)
    end
```

Writing these enclosing routines (and updating them when you change parameters) is relatively time-consuming and really interrupt the flow of programming. Since in this case you only need to wait for the processor of one object, SCOOP should not require an enclosing routine or introduce a shorter syntax to achieve the same functionality.

3.4 Provided Functionality

A general overview of the functionality can be found in Chapter 1. The following section gives you a good overview of the functionality by listing the supported commands of the IRC server.

3.4.1 Commands

Here is a list of commands supported by this implementation:

- *NICK <nickname>*
  Set or change your nickname.

- *USER <username> <hostname> <servername> <realname>*
  Set additional user information.

- *JOIN <channel>*
  Join a channel.

- *PART <channel>*
  Leave a channel.

- *KICK <channel> <nickname> <reason>*
  Kick a user from a channel.

- *MODE <channel> {[-|+]i|s|m|n|t|o|p} [<user>{<space><user>}]*
  Set channel modes or change privileges of users in a channel.

- *TOPIC <channel> [<topic>]*
  Set or unset the channel topic.

- *NAMES <channel>*
  Request a list of users in a channel.

- *LIST*
  Request a list of all channels.

- *PRIVMSG <nickname> <message>*
  Message a user.

- *WHOIS <nickname>*
  Request user information.

3.5 Milestones

I developed the application step-by-step and tested my code thoroughly after every modification.

The development process can be described best by breaking it down into the following stages:

3.5.1 SCOOP Concurrency

The first thing I did was trying to get SCOOP functionality working and play around with it a bit in order to get a feel of how it works.

SCOOP is built in into EiffelStudio 6.8, but it has to be enabled first: Go to Project → Project Settings → Advanced → Concurrency and set it to SCOOP.

3.5.2 Echo Server

My first real goal was a server that several clients can connect to simultaneously (e.g. with telnet). The server will repeat every line of text it receives back to the client. My idea was to create separate objects to deal with each socket individually and concurrently. It took quite some time to figure out a good way to do this – section 3.1.1 focuses on this aspect.

3.5.3 Echo-to-all Server

The next step was to modify my echo server and broadcast any line of text it receives to all connected clients. However, up until this point, every `IRC_USER` was self-contained and did not know of the existence of other IRC users.

I introduced a global `CONTEXT` object which keeps a list of all connected users and is referenced in every `IRC_USER`. When an `IRC_USER` wants to broadcast a message, it invokes the `CONTEXT` object which then messages every user.

3.5.4 Interaction with mIRC

One of the major changes from the last milestone is the necessity for parsing messages. The parsing itself takes place inside the `IRC_USER`. Also, in order to address individual clients, the `CONTEXT` object needs to be able to associate nicknames with sockets; I use a `HASH_TABLE` for this.

In order to actually be recognized as an IRC server by mIRC very less is required. Upon connecting, mIRC sends the commands *NICK* and *USER* to the server. The server replies with *001* (*RPL_WELCOME*).

Now you can interact with other users on the IRC server. I implemented the command *PRIVMSG* to let users send messages, assuming they know their target's nickname.

3.5.5 "Complete" Functionality

Now it was time to implement the remainder of the functionality listed in section 3.4. This step was very time consuming as it required accurate implementation of the specification.

4 User and Developers Guide

4.1 User Guide

The IRC server can be accessed via Subversion under:

https://svn.origo.ethz.ch/scoop/examples/ircd

Open the project file (ircd3.ecf) in EiffelStudio 6.8 or later and compile and run the source code.
Running the project creates a local IRC server to which IRC clients can connect to on port 6667.
There is currently no configuration file. Changes to the IRC server have to be made directly in the source code.

4.2 Developers Guide

Chapter 3 explains the functions of the classes and the flow of events. Apart from this, the code is should be self-explanatory. Unfinished parts that were not part of the requirements and can be improved are highlighted by comments in the code.

Future work can include implementing the missing functionality described in section 4.2.1 and adding the ability to configure the IRC server using an external configuration file.

4.2.1 Missing Functionality

The following functionality is part of RFC1459 but has not been implemented yet:

- Origin prefixes are not ignored in client-to-server messages.
- Nicknames and channel names are not checked for validity (max. length, valid characters) and are not case-insensitive.
- There is no maximum message length. (should be 512 including CRLF)
- Not all error codes are returned if an error occurs.
- Properly handling disconnecting clients.
- No support for multiple users or channels in a single command. (e.g. JOIN #foo,#bar)

- WHOIS only supports users, not servers. (makes sense, since there is currently only one server in this implementation)
- MODE has no support for ban masks or setting a channel limit.
- LIST does not support parameters.
- The following commands:

SERVER	Connect to server.
OPER	Become IRC operator.
SQUIT	Server disconnect.
VERISON	Request server version.
STATS	Request statistics.
LINKS	Check for linked servers.
TIME	Request time.
CONNECT	Connect server to server.
TRACE	Find route to server.
ADMIN	Find name of server administrator.
INFO	Request server information.
WHO	Like WHOIS for several matches.
WHOWAS	Like WHOIS with history.
KILL	Kill a user.
ERROR	For the server to report serious errors.
REHASH	Re-read config file.
RESTART	Restart server.
SUMMON	Ask people to join IRC.
USERS	Similar to WHO/RUSERS/FINGER. (deprecated)
WALLOPS	Send message to all IRC operators.
USERHOST	Request userhosts for multiple users.
ISON	Requests online status for multiple users.

5 Conclusion

In my opinion, the concurrency model of SCOOP is very well suited for developing an IRC server. It is possible to maintain an object-oriented structure and still take advantage of multiple, concurrently active execution vehicles.

However, in its current stage of development, SCOOP is not ready yet to be used by a larger audience. The base library has no support for separate types – it's not even possible to print a separate *STRING* to the console easily. Only basic types, such as *CHARACTER* and *INTEGER*, can be accessed if they are on a separate processor. Separate objects have to be deconstructed to the level of basic types and reconstructed from ground up in order to be used. It seems like this can be avoided for the most part if libraries are extended with support for separate types.

The difficulties described in section 3.3.1 concerning the listening socket are a general problem. The general scenario is an object factory: The returned newly created object will always be on the same processor as the factory. This is undesirable in many cases. Since it is an 'untouched' object that has no affiliation with any processor, SCOOP should provide functionality to assign it to another processor than the current one.

Using a single object to hold data and offer routines to access and change it eliminates the problematic of race conditions. I did not have to deal with atomicity of operations, because all critical data is manipulated from a single object. This aspect is also discussed in section 3.3.2.

6 References

[1] EIFFEL website – http://www.eiffel.com/

[2] SCOOP website – http://www.origo.ethz.ch/

[3] RFC 1459, Internet Relay Chat Protocol – http://tools.ietf.org/html/rfc1459