EPCC-SS97-12

EPIC2000

Golestan Radwan

Abstract

The objective of this project was to re-implement the existing distance learning package **EPIC**, making use of the emerging networking and world-wide-web technologies, moving on to **EPIC 2000**. This document describes the old **EPIC** model, the new approach and the reasons for which it was chosen, as well as the details of the Java implementation of **EPIC 2000**.

Contents

1 Introduction **3**

2 Background **3**
2.1 What is **EPIC** and how does it work? 3
2.2 The old **EPIC** model 4
2.3 Weak-points of the old EPIC model 5

3 EPIC2000 **6**
3.1 EPIC2000 model 6
3.2 Why a Proxy Server? 6
3.3 Why Java? 7
3.4 The EPIC GUI 7

4 Implementation of EPIC 2000 **8**
4.1 The ProxyServer class 8
4.1.1 Design issues and problems 8
4.1.2 The code 8
4.2 The epic_com implementation 8
4.2.1 Problems encountered 9
4.3 The epic_master implementation 9
4.3.1 Problems encountered 11
4.4 The GUI implementation 11

5 Current EPIC 2000 state **12**

6 The Future **12**
6.1 What could have been done 12
6.2 Possible further extensions to EPIC 12

7 References **14**

1 Introduction

EPIC (**EP**cc **I**nteractive **C**ourseware) is a distance learning system developed at *EPCC*. The package was originally written by Martin Westhead in 1994, and has undergone many changes and improvements. In the meantime, more *EPCC* courses have been written and integrated into the package. **EPIC** is a useful aid in classroom teaching at *EPCC*. Furthermore, Summer Students and visiting researchers at the centre use **EPIC** as a self-learning aid for EPCC courses. Furthermore, being in the public domain, **EPIC** is accessible by anyone through the web, who wishes to download the courses and work on them. However, **EPIC** was written at a time where networking and web technologies were still evolving. Hence, it doesn't make use of the facilities available today. This project was proposed to help develop **EPIC** for the 21st century, moving on to **EPIC 2000**, by making use of the Java language features. The new **EPIC 2000** is platform-independent, self-contained, portable, and easier to install. Further scope of the project is to add some new functionalities into **EPIC**, trying to make it compete with other distance-learning systems in the world of today.

2 Background

2.1 What is EPIC and how does it work?

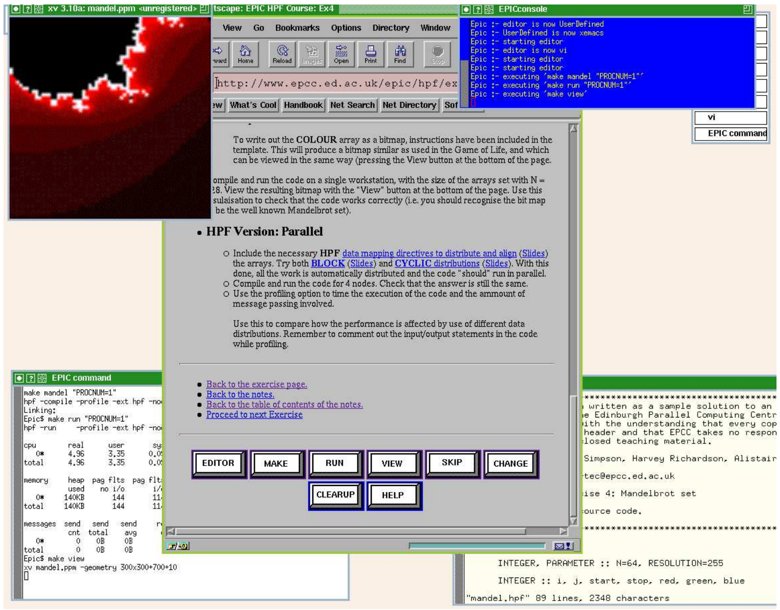

Figure 1: A typical **EPIC** environment

The figure above displays the typical **EPIC** environment. It consists of the actual exercise page, as well as the components listed below:

- The `EDITOR` button: starts an editor (lower right window in figure)with the current exercise file. If the file does not exist, an empty editor is started. If no specific editor has been selected by the user, a default editor is used.

- The MAKE button: compiles the current exercise code using the Unix *make* command.
- The RUN button: runs the compiled code.
- The VIEW button: displays any graphical output the program might have made. It makes use of an external graphics package, such as xv or AVS. The graphical output of the exercise shown above is the Mandelbrot set, displayed in the upper left window of the screen. If the current exercise does not have a graphical output, this button is not found on the exercise page.
- The SKIP button: gives the user the option of skipping the current exercise. They are allowed to view the model solution, or proceed right to the next exercise.
- The CHANGE button: allows the user to make some changes to the current configurations, such as the number of processes they are running, the editor to use, or the hosts the code is running on.
- The CLEARUP button: terminates the current exercise, and clears up all windows on the screen.
- The HELP button: displays a help page, providing the user with some tips on how to use the package.
- The Command window (lower left window in figure) executes the MAKE, RUN, and VIEW commands. It displays any system calls and the system's response to them.
- The Console window (upper right window in figure) is a system status indicator. It echoes back the command currently being executed, and any error messages.

2.2 The old EPIC model

EPIC is a set of *Perl* scripts, combined with various *Unix* applications. It consists of two main scripts, epic_com and epic_master. When the user installs **EPIC**, epic_master, is installed onto his local machine, called the *client* machine, along with the files of the courses they chose to download. In order for the installation to complete successfully, the user needs to have Unix, Perl and X11 installed on their machine. The old **EPIC** model relies on the *Helper Application/Remote Viewer* mode of use. This model operates with the browser sending an HTTP-'get' request to the desired web server, which responds with the requested document's HTTP-header. The HTTP-header is a document header, which is echoed back as part of a server's response to an HTTP request. It contains several pieces of information about the document, including the document's date, server, content-type, etc. The browser takes an action according to the document's content-type, also called *MIME*-type. *MIME* stands for *M*ultimedia *I*nternet *M*ail *E*xtensions. It describes the type of information that document contains (HTML code, sound, image, etc.). These are called *MIME*-types, and are part of a pre-defined set within the browser. Each type is associated with a respective action the browser is supposed to take, for example, a 'postscript' *MIME*-type would be associated with the application 'ghostview'. This means that, whenever a 'postscript' *MIME*-type is found in an HTTP-header, the browser would fire up 'ghostview' to view that document.
EPIC uses the same technique. When the user presses a button on an **EPIC** page, a URL is sent to the *EPCC* web server. URL stands for *Universal Resource Locator*, which is the "address" for any document on the web. The *EPCC* web server receives this URL and calls epic_com with

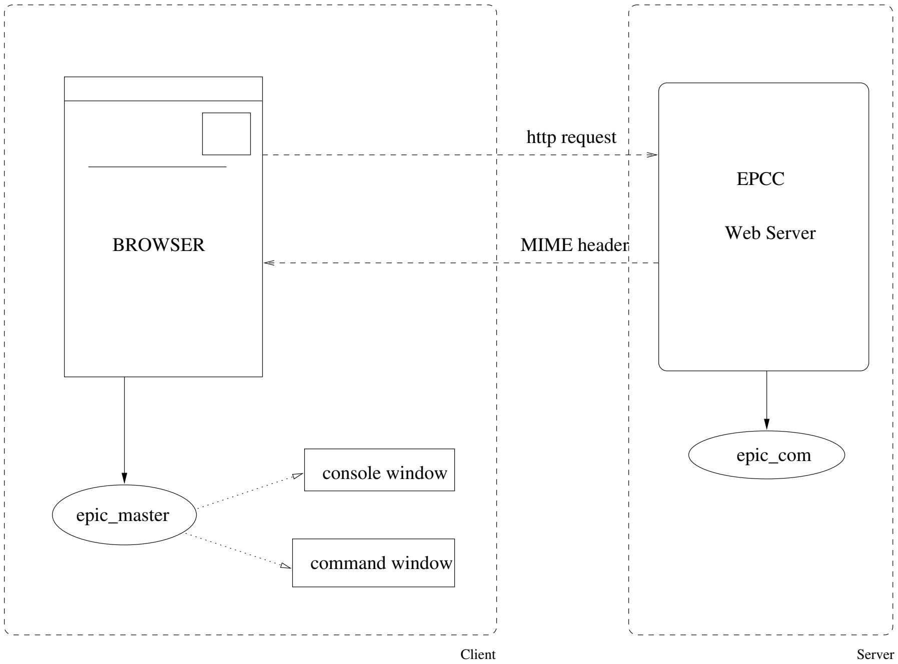

Figure 2: The old **EPIC** model. On the left, the client machine running the browser, and having `epic_master` installed in the local environment. The two small boxes represent the `console` and `command windows`, which are initiated by `epic_master`. On the right, the server machine at *EPCC*, with `epic_com` installed. The browser send an HTTP-get request to the *EPCC* web server, and receives an HTTP response.

the URL information, which interprets it and sends an HTTP-header back to the browser, along with the requested document. If the request is for an HTML page, the content-type `text/html` is echoed back and causes the browser to display the requested HTML page. If the request is for an **EPIC** command to be executed, the content-type `application/x-epic` is echoed back to the browser. The action taken in this case is to initiate `epic_master`, which executes the requested command. This script, `epic_master`, also handles the firing up of the different windows, and then dies after processing each request and has to be initiated again to process the next command.

2.3 Weak-points of the old EPIC model

Obviously, this model suffers from some drawbacks, which can be summed up by the following points:

1. Too much network traffic with connections to a potentially distant server to process each request.
2. Platform dependent, with the user required to have several packages on his local machine.
3. Time-consuming, with remote connections and re-initiation of `epic_master` for every new request.

3 EPIC2000

3.1 EPIC2000 model

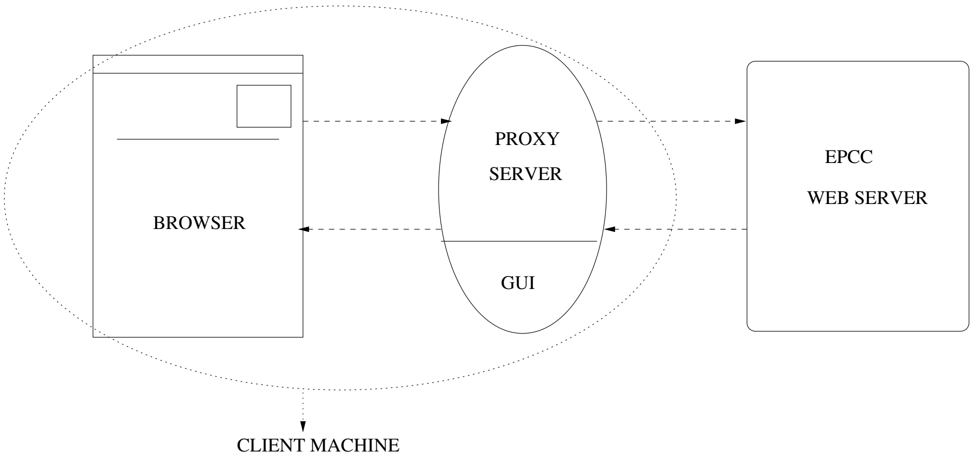

Figure 3: The new **EPIC 2000** model. On the left, the client machine running the browser, with the proxy server installed on it. The proxy server contains its own GUI, which initiated when the server is started, and idea when the server is killed. On the right, the server machine. Connections between the client and the *EPCC* web server are minimised, but still possible.

The new **EPIC** model proposes having a light-weight server on the client machine, which would handle the jobs of `epic_com` and the EPCC web server. Instead of connecting to the remote host each time to process a new request, the user just has to connect to their local host. Connections to the *EPCC* server are minimised, but not totally eliminated. You can still connect to *EPCC* to fetch files that are not available locally, or to download a new course.
Furthermore, **EPIC 2000** has to be backwards compatible, to enable the use of existing course material and HTML pages.

3.2 Why a Proxy Server?

Having a server on the local machine reduces the connection time remarkably. Furthermore, minimising the connections to the *EPCC* web server reduces the load on it, allowing potentially an unlimited number of users to run **EPIC** at the same time, without any latency concerns.
The proxy server makes **EPIC** more self-contained, in the sense that it does not depend on the state of another server any more. So if for example the *EPCC* server goes down, the user would still be able to run **EPIC**, whereas in the past, they had to wait for the server to come up again. An important advantage of having a proxy server is also the persistence of state. The server survives throughout the whole **EPIC** session and is not killed until the user quits the application, as opposed to `epic_master`, which had to be initiated every time a new request had to be processed.

3.3 Why Java?

Java was designed with a networking approach in mind. It provides a wide range of facilities for network-based applications, and is therefore becoming the natural choice for anyone who wants to write such an application. Another advantage of Java is its platform-independence. If **EPIC 2000** is to be platform-independent, then Java is the most realistic solution. Platform-independence is a very important feature, since it will increase the user-base of **EPIC** to include non-*Unix* users as well. Java is object-oriented. This makes **EPIC** portable and maintainable. If in the future new features should be added to it, it would be easy to write them separately and then integrate them into the package without having to make major changes. And finally, re-writing **EPIC** in Java would make it up-to-date and give it a future, as opposed to *Perl*, which is less suitable for web-based applications.

3.4 The EPIC GUI

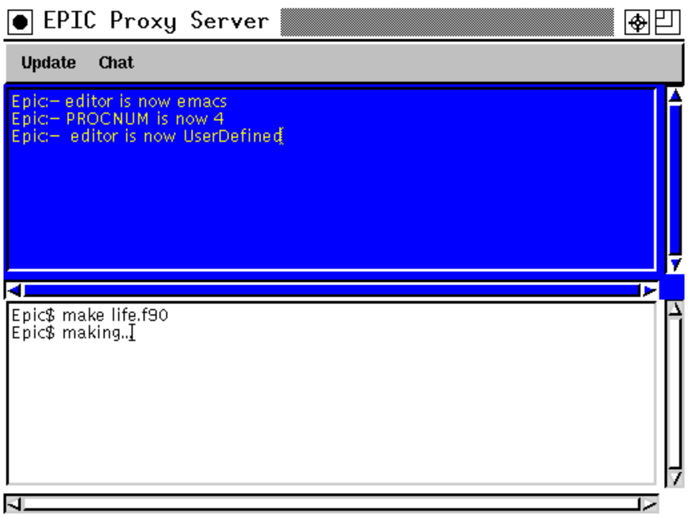

Figure 4: The new **EPIC 2000** GUI. The upper window is the console window, displaying the system state, the lower window is the command window, echoing **EPIC**s calls to the system, and the system's response to them. The menu bar on the top includes some of the existing options and can easily be expanded to absorb more.

EPIC 2000 has its own Graphical User Interface (GUI). This avoids the somewhat annoying feature of several windows popping up and dying from time to time. This also facilitates the implementation of the server, with no need for sockets or *X11* to run **EPIC**. The GUI is initiated when the server starts running, and dies when the server is killed, as opposed to `epic_master`, which had to be initiated every time to process a new request.
The GUI includes a `console window` on top and a `command window` underneath, which perform the exact same functions as the respective windows in the old version.

4 Implementation of EPIC 2000

4.1 The ProxyServer class

4.1.1 Design issues and problems

Some points had to be considered while preparing the design of the proxy server.
First of all, **EPIC 2000** had to be backwards compatible. As much as possible of the existing material should be usable in the new version. This meant a more or less similar structure to the existing `epic_com` and `epic_master` design. To preserve readability and understandability of the code, most of the old names and functions of routines had to be preserved. All these considerations put some restrictions on the design, which would not be existent if the proxy server was being designed from scratch.
To ensure the feasibility and functionality of the new implementation, the server had to be implemented step-by-step, with tests running before and after each stage was completed. This was very time-consuming, especially that Java is a slow language to compile and run, and that the machine used for the compilation (the ssp) is slow as well.

4.1.2 The code

The ProxyServer class is mainly an encapsulation of both the functionalities of `epic_com` and the *EPCC* web server. Currently, the server listens on port 60000 of the local host, but it can be easily modified to listen to another port. After the connection has been established between the client and the server (usually through the pressing of an **EPIC** button), the server receives the *CGI* request from the client. This request is searched for the string `/cgi-bin/epic/epic_com/`, which represents the **EPIC** command *path_info*. If this string is found, it means that an **EPIC** command is in progress. The string is parsed further to determine the first word after the *path_info* of the request. This will be the keyword which identifies which **EPIC** command is involved.

4.2 The epic_com implementation

The following methods are called depending on the keywords found in the request:

- `trans` and `install`: call the *trans* method. This method looks for the file specified in the request on the client machine. If it doesn't find it, it connects to the *EPCC* web server. When the file is found, it is transferred to the local host. It echoes back the content-type *application/x-epic*
- `skip_page`: calls the *skip_page* method. This method simply displays an HTML page, giving the user the options to skip the exercise, display the model solution, or proceed directly to the next exercise. It echoes back the content-type *text/html*.
- `configure`: calls the *configure_page* method. This method displays an HTML page about the changes a user is allowed to make to the configurations.
- `localchange`: calls the *change_page* method. This method creates an HTML page dynamically, i.e. at run-time, according to the configurations the user is allowed to change.

At some points, for example, the user is only allowed to change the editor, but not the number of processes, and so on. Related to this method are the methods that display the respective parts of the configuration page: *print_page_head*,*print_editor*,*print_hosts* and *print_nprocs*.

- The *header* method creates the HTTP-header and echoes it back to the caller method.
- The *get_rest* method takes the request, transforms it into a string, and parses that string to search for the substring `/cgi-bin/epic/epic_com/`. Then, it echoes back the first word after that substring, which is the actual **EPIC** command, followed by its argument(s).

4.2.1 Problems encountered

Problems encountered at this stage were basically in finding the corresponding Java-syntax to the existing *Perl* code. Two main problems were encountered:

1. Finding an appropriate Java equivalent to the *Perl* regular expressions. Java provides a wide variety of string-parsing utilities. However, it does not provide anything similar to a regular expression. Use of the *indexOf()* and *substring()* methods in the *String* class were of particular significance.
2. A problem with the *change_page* method was the fact that the number of argument to that method can only be determined at run-time. Hence, a variable-size array has to be created, which required further string parsing that was not necessary for the *Perl* implementation. A way around this was writing a whole method to parse the string and determine the number of arguments and then create the array, which was a rather tedious job. However, this method proved to be useful later on in the `epic_master` implementation.

4.3 The epic_master implementation

The first step towards the implementation of `epic_master` was the design of the **EPIC 2000** (GUI). Simple methods of the *java.awt* package were used to design a frame containing two windows and menu_bar.

The following is a listing of all `epic_master` routines, and their current status, with regards to whether or not they are implemented, deprecated or re-implemented in another way.

The following list displays the routines needed for **EPIC 2000** and their functions:

sysInit: performs some initialisations and variable declaration.

trans: Takes the file, uncompresses it, and puts it into the proper directory. Slightly modified to absorb the old *install* functionality. It deletes the old files and installs the new ones.

change: makes changes in the current configurations.

check: checks the path on the client machine for the executables needed to run **EPIC**.

epictest: simply displays messages to the console window.

edit: fires up an editor window.

Routine	Plan	EPIC 200 Java method	Current status
sysInit	needed	init	implemented
trans	needed	trans	implemented
install	not needed		
change	needed	change	needs minor corrections
check	needed	check	not implemented
clearup	not needed		
echo	not needed		
epccchimp	not needed		
epictest	needed	epictest	not implemented
edit	needed	edit	not implemented
make	needed	make	not implemented
mvExercise	needed	mvExercise	not implemented
tutor	needed	tutor	not implemented
getEPICFiles	needed	getEPICFiles	not implemented
skip	needed	skip	not implemented
fullname	needed	fullname	not implemented
readconfig	needed	readconfig	implemented
writeconfig	needed	writeconfig	implemented
consolewin	not needed		
active	not needed		
activenum	not needed		
sendcom	not needed		
transmit	not needed		
flush	not needed		
printflush	not needed		
CommsMain	not needed		
createSocket	not needed		
readData	not needed		
readSocket	not needed		
executeCommand	not needed		
checkpointCommand	needed	checkpointCommand	not implemented

make: reads and executes the makefile of the current code.

mvExercise: moves the current exercise to the tutor's directory.

tutor: sends an e-mail to the current course's tutor.

getEPICFiles: returns the names of the files which should be sent to the tutor.

skip: copies the current exercise in to a *.skipped file, and eventually displays the model solution to the current exercise.

fullname: returns the full path to a certain file.

readconfig: reads the configuration file and writes it to a hash table. If no configuration file exists, it creates one with some pre-defined default values.

writeconfig: writes the current changes to the configuration file.

The following lists the deprecated `epic_master` routines, and the reasons for their deprecation:

install: most of the features it contains deal with sockets and xterms, and are not needed in the current implementation. The rest of the functionality is absorbed into the *trans* method.

clearup: The console and command windows will be contained in the GUI. Hence, there is no need to kill the windows, since they will be killed automatically when the server is killed.

echo: It simply echoes back some arguments, which is an unneeded feature in the current implementation.

epccchimp: This routine was originally designed to incorporate the *CHIMP* MPI interface at *EPCC* and is not needed any more.

consolewin: creates a socket interface to an xterm which runs the console window. This window is enclosed in the GUI now, so there is no need to create an xterm any more.

active: write the *.active* file, which stores process IDs. It was needed because *epic_master* needed to be re-initiated after processing each request. Since this is not the case any more, the file is not needed.

activenum: deals with the *.active* file, which is not implemented any more.

sendcom, *transmit*, *flush*, *printflush*, *CommsMain*, *createSocket*, *readData*, *readSocket*: perform socket handling. With the new GUI, there is no need for sockets any more, since it is self-contained and persistent throughout the lifetime of the proxy server. *executeCommand*: executes a command. Can be encapsulated into another method. No need for a separate method to do it.

4.3.1 Problems encountered

The main problem was the inconsistency of the old version. Some routines had some bugs, others were modified and enhanced more than once. There was not enough documentation and the code was not very readable.

A second problem was the Java system interactions. *Perl*, being a *Unix*-based language, supports many system interactions and facilitated *Unix* system calls. Java, on the other hand, being platform-independent, does not support that many system interactions. There are many differences between the system handling in JDK 1.0 and 1.1, many of which have caused confusion and instability in the code. A typical example is the *exec()* method in *java.lang.system*. In JDK 1.0, it requires the whole path to the *Unix* command, which may very well vary from one machine to another. This was one of the main reasons the whole project had to be written using JDK 1.1, which supports better system interactions.

4.4 The GUI implementation

The purpose of having a separate GUI for the server is to eliminate the need for xterms and sockets, which had to be used in the old version. With that approach in mind, the GUI was designed to contain both the console and command windows, making them perform the same functions as they used to in the old version, and so preserving the backwards compatibility and maintaining the same 'look and feel' of **EPIC** for people who are already used to it.

During the execution of the application, output is constantly sent to the console window upper window) for immediate system state indication. Once a system call is made, the call and the system's response to it are sent to the command window (lower window). At the top, there is a menu bar, which includes some functionalities, such as 'quit' or 'update'. Most of these are not implemented yet, but it also has a large space for possible future extensions.

5 Current EPIC 2000 state

Currently, **EPIC 2000** is almost completed. The proxy server performs all the `epic_com` functionalities, and is compatible with the old `epic_master` version. Basically, the ProxyServer class can be installed onto any machine that uses the existing **EPIC** model, and replace epic_com. The user would have to connect to their local host instead of the remote host at EPCC. This would require minor changes in the HTML pages, but should be achievable. Some of the `epic_master` functionality has also been incorporated into the proxy server. As indicated above, however, `epic_master` has undergone radical changes, as opposed to the minor changes made to `epic_com`. It would be advisable of course to install the two scripts together, instead of only one, once they're finished.

6 The Future

6.1 What could have been done

The `epic_master` functionality is incomplete. The tutor button functionality still has to be implemented. This is a set of methods, allowing the user to send an e-mail to the tutor, explaining their problem. The e-mail would automatically include a tar file of the exercise the user has been working on. This file would then be automatically un-tarred and installed into a temporary directory on the instructor's machine, allowing them to see exactly where the problem is and find the appropriate solution for it. Some other enhancements could have been made to the GUI, such as resizing the windows, being able to kill one of them and keep the other, etc. Finally, another important feature would be to provide the facility for the user to connect to the remote server at EPCC to fetch the **EPIC** home page and see the most recent changes and decide, whether or not an update is needed.

6.2 Possible further extensions to EPIC

- A pure Java editor. This would increase the platform-independence of **EPIC**, and make it even more self-contained.
- A PC-compatible **EPIC** version, so as to allow PC users to use **EPIC** as well.
- More distance-learning features, such as chat-rooms or White-boards. A further extension would be providing audio transmission as well, so as to create virtual classrooms.

- Some EPCC staff members have raised the point of **EPIC** depending on several external packages, especially for graphical display and image processing, some of which are very memory and system-resource consuming. A solution would be integrating some of these packages' functionalities into **EPIC**, to eliminate the need for any external software.

7 References

References

[1] Schwartz, R.,
Learning Perl, O'Reilly & Associates, Inc., 1993.

[2] Wall, L., and Schwartz, R.,
Programming Perl, O'Reilly & Associates, Inc., 1991.

[3] Gilly, D., and The staff of O'Reilly & Associates, Inc.,
Unix in A Nutshell, O'Reilly & Associates, Inc., 1992.

[4] Naughton, P.,
The Java Handbook, McGraw-Hill, 1996.

[5] Westhead, M., Antonioletti, M., Ewing, A., and Yau, H.,
EPIC Users Guide

[6] Westhead, M., Ewing, A., and Yau, H.,
EPIC Planning Document

[7] Westhead, M., Ewing, A., Yau, H.,
`"The EPIC source code"`

[8] Harold, E. R.,
Java Network Programming, O'Reilly & Associates, Inc., February 1997 (First Edition).

I study Computer Engineering at Cairo University in Egypt, now about to start my 3rd year out of five. So far, I've done a year of general engineering studies and a year of Computer Engineering. Many aspects of computing interest me, among which is networking. Since I arrived in Edinburgh, I have developed an interest in High Performance Computing as well, and I hope to have more chance of exploring that field some time. In my free time, I enjoy reading, listening to music and going out. Sports include basketball, tennis, and, very recently, badminton. I also like travelling and exchanging e-mail with my friends!

At the end, I would like to thank my supervisors in this project for being most helpful and encouraging all the time. SSP has been a whole new experience for me and they really made it worthwhile. Many thanks to:

Mario Antonioletti
Dr Stephen Breuer
Martin Westhead
and last, but most certainly not least,
Dr Hon Yau