



Swansea University
Prifysgol Abertawe



Swansea University E-Theses

JACIE - A scripting language for Internet-based multimedia collaborative applications.

Haji-Ismail, Abdul Samad

How to cite:

Haji-Ismail, Abdul Samad (2001) *JACIE - A scripting language for Internet-based multimedia collaborative applications..* thesis, Swansea University.

<http://cronfa.swan.ac.uk/Record/cronfa42685>

Use policy:

This item is brought to you by Swansea University. Any person downloading material is agreeing to abide by the terms of the repository licence: copies of full text items may be used or reproduced in any format or medium, without prior permission for personal research or study, educational or non-commercial purposes only. The copyright for any work remains with the original author unless otherwise specified. The full-text must not be sold in any format or medium without the formal permission of the copyright holder. Permission for multiple reproductions should be obtained from the original author.

Authors are personally responsible for adhering to copyright and publisher restrictions when uploading content to the repository.

Please link to the metadata record in the Swansea University repository, Cronfa (link given in the citation reference above.)

<http://www.swansea.ac.uk/library/researchsupport/ris-support/>

JACIE – A Scripting Language for Internet-based Multimedia Collaborative Applications

by

Abdul Samad Haji-Ismail, BSc, MSc

Thesis submitted to the University of Wales
in candidature for the degree of Doctor of Philosophy

December 2001

Department of Computer Science
University of Wales, Swansea

ProQuest Number: 10807454

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10807454

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

To my family.



DECLARATION

This is to certify that neither the worked described in this thesis, nor any part of it, has already been accepted in substance or being concurrently submitted in candidature for any degree.

Candidate:
.....

Date: | MARCH 2002

STATEMENT 1

This is to certify that except where specific reference to another investigator is made, the work described in this thesis is the result of the investigation of the candidate.

Candidate:

Date:

Director of Studies:

Date:

Director of Studies:

Date:

STATEMENT 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Candidate:

Date:

ACKNOWLEDGMENTS

I am indebted to my supervisors, Professor Min Chen and Dr. Phil W. Grant, for their guidance and friendship throughout the course of my doctoral research. Their patience, insight, trust and assistance beyond basic responsibility have been integral to the success of this work.

My appreciation also goes to the secretarial and technical staff of the Department of Computer Science, University of Wales Swansea who ensured that everything was in place for a conducive working environment.

I would like to express my gratitude to the administration of the University of Technology Malaysia for the financial support throughout the course. Special thanks to my colleagues who understood my situation during the extension period of the course.

I owe thanks to Abdul Manan and Salbiah who had patiently gone through the pages of this thesis and given me invaluable comments.

Finally, I am deeply indebted to my wife Noorlhuda and my children Irfan, Asyraf, Nabilah and Athirah for their love, understanding, support, prayers and patience, and without these, this work would have never been completed. I am also forever grateful to my parents who have brought me up with love, wisdom and sacrifice. May the blessings of God be upon you.

SUMMARY

The Internet has opened the opportunity for geographically dispersed computer users to concurrently interact, collaborate and socialise in a virtual group environment. It has shifted the nature of Internet users from “individualistic net-surfers” to “active collaborative teamworkers”. But developing Internet-based collaborative applications is very laborious, tedious and time consuming. Besides patience, knowledge and skill in low-level network programming are required especially for managing interactions and communications.

This thesis presents a research on the construction of a development tool for collaborative multimedia applications. The tool, named JACIE (Java-based Authoring language for Collaborative Interactive Environments), is a script language which has been developed to support rapid implementation of a wide range of network-based interactive and collaborative applications. In particular, it facilitates the management of interaction and communication through simple communication primitives such as channels and interaction protocols, hence hiding much network programming from programmers. JACIE also features a template-based programming style, a single program for both client and server, and platform-independence by using Java as the target language. A compiler prototype has been developed that translates JACIE codes to Java. Several sample applications have been implemented in JACIE and are discussed in the thesis.

The major research contribution is a high-level abstraction language for collaborative multimedia applications that simplifies many programming tasks. JACIE can be a useful multimedia software engineering tool well-suited for a wide range of collaborative applications, be they stand-alone client/server applications or Web-based client/server applets.

Contents

1	Introduction	1
1.1	The Internet and Global Collaborations	2
1.2	Project Overview	4
1.2.1	Objectives	4
1.2.2	Principles	5
1.3	Thesis Outline	6
2	Related Works	8
2.1	Computer-Supported Cooperative Works	8
2.1.1	Definition	8
2.1.2	Advantages	10
2.1.3	Applications	12
2.2	Collaborative Virtual Environments	15
2.2.1	Definition	15
2.2.2	Related Research	16
2.2.3	Virtual Realism Versus Effective Multimedia, Interaction and Communication	18
2.3	Distributed Environments and Collaborative Tools	18
2.3.1	Distributed-processing Environments	18
2.3.2	Java for Developing Collaborative Applications	20
2.3.3	Java-based Collaborative Frameworks	21

2.4	Scripting Languages	22
2.4.1	Perl, Tcl and Python	24
2.4.2	JavaScript and VBScript	24
2.4.3	Other Scripting Languages	26
2.4.4	Scripting Language for Interactive and Collaborative Applications	26
2.5	Summary	26
3	Design Considerations	28
3.1	Typical Applications	29
3.1.1	Server-based Interactive Applications	30
3.1.2	Server-mediated Interactive Applications	32
3.1.3	Group Collaborative Applications	33
3.2	Scripting language for Interactive and Collaborative Applications	34
3.2.1	Scripting Versus Other Alternative Programming Approaches	35
3.2.2	The JACIE Scripting Language	36
3.3	Template-based Programming Style	37
3.3.1	What is a template-based programming style and how does it simplify programming tasks	37
3.3.2	JACIE's Template-based Programming Style	38
3.3.3	The Template Structure of the JACIE Language	38
3.4	Protocol Handling	41
3.4.1	Overview of Protocols	41
3.4.2	JACIE's Protocol Handling	42
3.4.3	Floor Management Protocol	42
3.4.4	Group Management Protocol	44
3.4.5	Interaction Protocols: A Case Study	45
3.5	Event-driven Programming	47
3.5.1	Overview of Event-driven Programming	47
3.5.2	JACIE's Event-driven Programming	47

3.6	Communication Channels	49
3.6.1	Overview of Communication Channels	49
3.6.2	JACIE's Communication Channels	49
3.7	Support for Graphics	54
3.8	Common User Access	55
3.9	Summary	58
4	Language Specifications	59
4.1	Template-based Scripting Language	60
4.1.1	System Configuration Component	61
4.1.2	Client Body Component	61
4.1.3	Server Body Component	61
4.2	Data Types, Operators and Expressions	62
4.2.1	Data Types	62
4.2.2	Operators and Expressions	63
4.3	Grammar	63
4.3.1	Main Constructs	64
4.3.2	System Configuration Statements	65
4.3.3	Client Implementation and Server Implementation Constructs	70
4.3.4	Variable Declaration Statements	71
4.3.5	Method Declarations	74
4.3.6	Basic Statements	75
4.3.7	Input-Output Statements	77
4.3.8	Graphics Statements	80
4.3.9	Event Control Statements	85
4.3.10	Communication Statements	87
4.4	Interfacing to Java Language	89
4.5	Summary	90

5	Compiler Implementation	91
5.1	Application Frameworks	91
5.1.1	Session Management Protocol and Delivery Management Protocol	91
5.1.2	Socket Programming and JACIE's Collaborative Architecture ...	94
5.1.3	Multithreading	97
5.1.4	JACIE-aware Interactivities	99
5.2	Compiler Tools	105
5.2.1	Java-based Compiler Tools	105
5.2.2	JFlex – Java Fast Lexical Analyser	106
5.2.3	CUP – Constructor of Useful Parsers	106
5.3	JACIE Compiler Environment	108
5.3.1	Lexical Analysis	108
5.3.2	Syntax Parsing	110
5.3.3	Semantic Analysis and Translation	112
5.3.4	JACIE-generated Java Programs	115
5.4	Software Components of JACIE-generated Java Programs	116
5.4.1	Types of JACIE-generated Java Programs	117
5.4.2	Software Components of JACIE-generated Client Programs	119
5.4.3	Software Components of JACIE-generated Server Programs	123
5.5	Running the JACIE Compiler	124
5.5.1	Configuring the JACIE Compiler	124
5.5.2	Compiling JACIE Script	126
5.5.3	Running JACIE-generated Client and Server Programs	127
5.6	Summary	130
6	Example Applications	131
6.1	A Server-based Interactive Application: Networked Knock Knock Jokes	132
6.1.1	Java Implementation	133
6.1.2	JACIE Implementation	137

6.1.3	Compilation and Execution	143
6.1.4	Summary	145
6.2	A Server-mediated Interactive Application: Multi-user Network	
	Troubleshooting	146
6.2.1	Implementation	148
6.2.2	Compilation and Execution	165
6.2.3	Summary	170
6.3	A Group Collaborative Application: Collaborative Scrabble	171
6.3.1	Implementation	173
6.3.2	Compilation and Execution	184
6.3.3	Summary	185
6.4	Result Analysis	185
6.5	Chapter Summary	187
7	Conclusion	188
7.1	Work Summary	188
7.2	Results and Contributions	189
7.3	Future Work	195
7.3.1	A Full Implementation of a Fully Reliable Compiler	195
7.3.2	Exploring the Possible Benefits of Visual JACIE	196
7.3.3	Extending the Web Server Feature with Collaborative Engine	196
	Appendix — The JACIE Language Specifications	197
	Glossary	205
	Bibliography	212

List of Figures

3.1	Framework of cooperative work	29
3.2	Client-server interactions in server-based interactive applications	31
3.3	Multi-user interactions in server-mediated interactive applications	32
3.4	Interactions in group collaborative applications	34
3.5	Transition state diagrams for server process and client process	39
3.6	Networked noughts and crosses game	45
3.7	A chat channel	50
3.8	A whiteboard channel	51
3.9	A voice channel volume adjuster	51
3.10	A video channel with receiving and transmitting live videos	52
3.11	A sample application employing a shared workspace	53
3.12	A sample application employing input/output messages	54
3.13	JACIE's standard user interface components	56
3.14	Variations of JACIE's menu bar to accommodate (a) tapping protocol or (b) reservation protocol	57
4.1	A set of standard JACIE components	60
4.2	A sample code showing JACIE's main construct	65
4.3	Standard user interface for connection/disconnection in JACIE-generated programs	66
4.4	Standard buttons for communication channels in JACIE-generated programs	67
4.5	A sample code showing JACIE's configuration statements	69
4.6	A sample code showing JACIE's variable declaration statements	72

4.7	A sample code showing JACIE's method declaration	75
4.8	A sample JACIE code with flow control statements	78
4.9	A sample JACIE program utilising input-output message bars	79
4.10	The running Echo2 client program	79
4.11	Standard canvas area for JACIE-generated programs with its coordinate system	80
4.12	A sample JACIE code with some graphics statements	84
4.13	A sample JACIE code showing some event control statements	87
4.14	A sample JACIE code showing communication statements	89
4.15	A sample Java code segment in a JACIE program	90
5.1	JACIE message in relation to other network layers	93
5.2	General model for network connections between clients and server	94
5.3	Extended model for networked interactive and collaborative applications	96
5.4	Server-based interactivities	100
5.5	Server-mediated interactivities	102
5.6	Group collaboration interactivities	104
5.7	Software architecture of JACIE	107
5.8	The user interface for JACIE-generated applications and applets	117
5.9	Nested layout diagram of JACIE user interface	120
5.10	A class diagram of JACIE-generated client programs (1)	121
5.11	A class diagram of JACIE-generated client programs (2)	121
5.12	A class diagram of JACIE-generated server programs	123
5.13	Output of running the JFlex tool on JACIE specifications	125
5.14	Output of running the CUP tool on JACIE specifications	125
5.15	Output of compiling Puzzle2.jacie program	127
5.16	The running process of Puzzle2Server	129
5.17	The Puzzle2 applet running in Microsoft Internet Explorer	129

6.1	The running KnockKnock client program as implemented by Campione and Walrath	132
6.2	The output of compiling the KnockKnock.jacie script	144
6.3	The KnockKnockServer program running on asahipc2	145
6.4	The running KnockKnock client program	145
6.5	The global view of Network Troubleshooting course	147
6.6	The network troubleshooting course in local view for Room 1 with a chat channel window	166
6.7	The network troubleshooting course in local view for Room 2 with a chat channel window	167
6.8	The network troubleshooting course in local view for Room 3 with a chat channel window	167
6.9	Transcript of the conversation for problem 1	168
6.10	Transcript of the conversation for problem 2	169
6.11	The collaborative scrabble game in action	172

List of Tables

2.1	CSCW space-time matrix	9
2.2	The role of communication media in different collaborative applications	15
3.1	Types of interactive and collaborative applications	30
3.2	Different versions of noughts and crosses in the case study for interaction protocols	46
3.3	JACIE-supported events	51
4.1	JACIE's primitive data types	71
4.2	JACIE's operators	72
4.3	Meta-symbols of BNF	73
4.4	JACIE's main events	80
4.5	JACIE-handled events	94
5.1	Functions of Managers and Assistants in interactive and collaborative applications	97
5.2	JACIE's system-defined message identifiers	105
5.3	JACIE compiler files	113
5.4	Naming conventions of JACIE-generated files	115
6.1	The NTS command instructions for network configurations	153
6.2	The Java classes for the client program of the Network Troubleshooting Application	165

6.3	The Java classes for the server program of Network Troubleshooting Application	165
6.4	The Java classes for the client program of the Collaborative Scrabble Game	184
6.5	The Java classes for the server program of the Collaborative Scrabble Game	185
6.6	The Java constructs required for developing example applications	186
7.1	The advantages and the disadvantages of available software tools for the development of interactive and collaborative applications	194

Chapter 1

Introduction

It is widely accepted that the Internet is the current information highway. As communication technologies develop, the Internet will turn into an information superhighway, or, even later, into an information hyperhighway. The Internet contains unimaginable amount of hypermedia information maintained by millions of hosts throughout the world. In fact, the Internet is more than just a highway and more than just a huge storehouse of hypermedia information; it is a medium for a global virtual community where social interactions take place among real human beings. Never before, in the history of mankind, have strangers around the globe been able to interact or collaborate freely with one another. It changes the normal rules for social interaction.

Collaboration is a deliberate activity. People collaborate for the enjoyment of the joint activity or furthering relationship. People collaborate by sharing each other's views, knowledge and experience in a process of effective learning and working.

This research is about collaborative activities over the Internet. More specifically, a research tool is proposed for developing interactive and collaborative applications using the Internet as the communication infrastructure. This authoring tool, named JACIE (a Java-based Authoring language for Collaborative and Interactive Environments), allows rapid prototyping of net-centric, multimedia and collaborative applications.

1.1 The Internet and Global Collaboration

Collaborative activities over the Internet commenced in the early history of the Internet. Electronic mail, or e-mail, has been used for research and education purposes from the very beginning of the Internet [84]. As more and more people came together to discuss and learn about matters of common interest, LISTSERV or automated mailing lists were developed [69]. E-mail and LISTSERV were seen as ideal tools for collaboration [133]. They have the speed of a telephone call with the ability to store an ongoing “conversation”. This means that e-mail, together with LISTSERV, are superior than the letter, fax and telephone in the support of collaborative working. Another Internet service similar to LISTSERV, but enhances group discussion, is USENET or newsgroup [69, 143]. It is an asynchronous threaded discussion forum similar to a bulletin board system.

Talk [106] is known to be the first Internet service that allows synchronous interaction for two users. First present in Unix, Talk allows two parties at remote locations to communicate through keyboarding in real-time. Currently the VoIP (Voice over IP) [122], which allows sending of voice in digital form over the Internet rather than in analog form over the public switched telephone network, overshadows the text-based Talk. Extending Talk, the Internet Relay Chat (IRC) [113] allows real-time electronic conversations between hundreds of remote users. They can chat about or discuss all kinds of topics. Many academic institutions maintain local newsgroups and IRC channels to allow their students and staff, whether remote or local, to converse on specific topics — be they academic discussions, campus events or employee announcements. One such example, NetFace [154], has been developed at Monash University in Gippsland, Australia. NetFace simplifies user interaction and integrates these functions. This menu-driven environment was tailored for their requirements and conceptualised as a virtual university. NetFace presents the students with “unreal places to go (classes, cafes, forums, the Union, the Chapel) and real things to do in those places, such as working in study groups, circulating research proposals for comment, bringing in guest speakers” [154].

Telnet [147] is another application that can be used for multi-user and collaborative work. One category of this type of applications is called MUD (Multi-user Dungeon) named after the original role-playing adventure game written by Roy Trubshaw and Richard Bartle, then students at Essex University [48]. Now MUDs, also known as Multi-user Dimensions or Multi-user Dialogues, are normally referred to as text-based networked virtual reality [36]. MUD applications provide an environment for synchronous, computer-mediated communication among many people. Each user takes control of a computerised avatar or character and can walk around, chat with other characters, explore dangerous monster-infested areas and solve puzzles [134]. There are many variations of MUDs: MUSH, MOO, MUSE and MUCK [see 21, 37, 36, 93, 136]. The most popular of all, which is being used for educational purposes, is MOO (MUD Object-Oriented). With its built-in object-oriented language, which is additional to normal MUD capabilities, users can also create new objects in virtual space like building a home, creating pets and programming new items that add up to the initial predefined objects. A well-known distance learning institution, the Virtual Online University, has used MOO since 1994 for Liberal Arts degrees [76].

A new generation of MUDs can support multimedia and virtual reality. This is as a result of integrating World Wide Web [18] and Virtual Reality Modelling Language (VRML) [75, 40] in MUDs [e.g. 98]. With the introduction of Java, a new breed of applications for collaborative environment has emerged. Sun Microsystems described Java as “a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, and dynamic language” [138]. Besides most of its features are being equally comparable to other programming languages, its platform-independent and Internet-oriented features make it stand out and have great popularity. It simplifies the current practice of network programming.

Recently, more networked multi-user applications have been developed for many different fields that incorporate some real-time collaborative features. There are also many electronic collaboration tools — a new medium to facilitate human-human collaboration.

Other than the ones developed for intra-departmental networks, all of the collaborative applications and electronic collaboration tools run over the Internet.

1.2 Project Overview

Despite the fact that the Internet is the main infrastructure for global electronic collaboration, initial research study showed that developing real-time Internet-based collaborative applications is still a very laborious and tedious task. Moreover, patience, knowledge and skill in low-level programming are generally required. Only after the introduction of the Java language with its built-in network support, has multi-user network programming been made simpler. However, it is not as simple as it first seems. It may be simpler for the hard-core system programmers but for non-specialist programmers this type of programming is still extremely difficult. It is even more difficult when the application to be developed requires some support of graphics or multimedia.

The current tools for developing real-time Internet-based collaborative applications are deficient. This research proposes a tool for average programmers to develop these kinds of applications in an easier manner.

1.2.1 Objectives

The objectives of this research are as follows:

- To identify issues surrounding real-time collaborative environments, e.g., the people (representing multiple collaborative users), the place (structuring shared worlds and visualisation), the activity (communication protocols between remote users within shared virtual worlds), and the development (programming these components to meet the collaboration objectives);

- To critically evaluate existing models, mechanisms or tools and their applicability in the development of collaborative environment applications;
- To extend and demonstrate the suitability of the Java programming language as a base for developing collaborative environment applications;
- To propose a scripting language that acts as a programming interface, as well as preprocessor to the Java interpreter, that simplifies the programming process of collaborative applications;
- To design and develop a language interpreter that converts programs in the scripting language to Java and
- To demonstrate the usability of the language through the development of some practical applications.

1.2.2 Principles

The aim of this research project is to develop a new software technology for the rapid prototyping of Internet-based collaborative environment applications. The tool, a scripting (or authoring) language called JACIE, is built on top of the Java language to make use of the Java's native support of the Internet. Two design principles of this language are as follows:

- **Special Purpose** — JACIE targets a collection of applications for which the existing programming tools would incur expensive development costs. These include groupware (e.g., collaborative management applications), courseware (e.g., educational teamwork courseware) and games (e.g. board and card games). These applications commonly feature real-time collaborative activities, a shared canvas or workspace, controlled access domains and structured communication. Requirements

for graphics, multimedia, communication media and communication protocols are also considered.

- **Programming Efficiency** — A scripting language is considered. This Java-based language has add-on features but is much simpler than Java and at the same time does not sacrifice Java's strength. This language hides the complexities of multi-user client/server programming by introducing high-level abstractions of various types of communication and interaction protocols. It uses a "single program" to specify both server and client. Higher-level grid-based graphic operations are provided in addition to basic graphic primitives. By simple specifications, it creates normal applications as well as applets that can be embedded in a Web page. Concentration is on collaborative communication protocols e.g., floor management, scheduling and synchronisation of shared objects and activities. Common user access is also considered for programming simplicity and for a familiar universal "look and feel" of all JACIE-generated applications

As a special-purpose language featuring efficient programming techniques, JACIE is anticipated to be an effective software development tool that enables networked interactive and collaborative applications to be developed at a very low development cost, within a short development period and by possibly inexperienced programmers.

1.3 Thesis Outline

This chapter provides the research background — the problems which need to be solved, objectives to be achieved and principles to be adhered to. It also spells out the outline of this thesis.

In Chapter 2, some related terms are introduced and the research related to the existing tools and implementations of collaborative environments are discussed. The discussion

also highlights the significance of collaborative works, especially over the Internet, and the need for an alternative development tool for interactive and collaborative applications.

In Chapter 3, issues concerning synchronous networked interactive and collaborative applications are discussed. These include the JACIE's classifications of interactive and collaborative applications, the design principles of the JACIE language and the language features in comparison to other alternative approaches.

Chapter 4 focuses on the formal specifications of the JACIE scripting language. This includes a general overview of the language, data types and expressions and the formal grammar. At the same time it shows how the issues discussed in the earlier chapters have been tackled and incorporated into the language.

Chapter 5 describes the development methodology and the implementation of the JACIE compiler. It also explains the software architecture of JACIE and software components of JACIE-generated Java programs for the client and the server.

In Chapter 6, three example case studies are described to demonstrate the viability of the proposed scripting language. Comparisons are made to show the simplicity and advantages of JACIE over Java. This chapter also shows the usefulness of the compiler.

The final chapter, Chapter 7, summarises the research work and the research results. Current contributions are described. Suggestions for enhancements are also discussed for future research work.

Chapter 2

Related Works

Throughout this thesis the word *cooperate* is used interchangeably with the word *collaborate*. This is due to the fact that most references in this research area used both words to describe a similar action. In Collins English Dictionary and Thesaurus [30], the word *cooperate* is defined as “work together” while the word *collaborate* is defined as “work with another on a project”. A sociologist cum educationist, Argyle [7] defined *cooperate* as “acting together, in a coordinated way at work, or in social relationships, in the pursuit of shared goals, the enjoyment of the joint activity, or simply furthering the relationship”. He further described that there are at least three reasons why people cooperate: for external rewards, to form and further relationships and to share the activities they are involved in.

2.1 Computer-Supported Cooperative Works

2.1.1 Definition

The term *Computer-Supported Cooperative Work*, or CSCW for short, was first used by Paul Cashman and Irene Grief in 1984 [67]. They organised a workshop attended by people from various disciplines sharing an interest in how people work with an eye to

understanding how technology could support them. It is said that CSCW “started as an effort by technologists to learn from economists, social psychologists, anthropologists, organizational theorists, educators and anyone else who can shed light on group activity” [67]. By being multidisciplinary in nature, there is no agreed upon definition of CSCW. But, in general CSCW refers to “the field concerned with the design of computer-based systems to support and improve the work of groups of users engaged on common tasks or objectives, and the understanding of the effect of using such systems” [53]. CSCW facilities are by means of computer-mediated communication (CMC); its tools or applications are often referred to as groupware; and its activities are referred to as electronic collaboration [50, 19]. With increasing acceptance of the Internet and the World Wide Web [18], these two technologies are becoming the main infrastructure for CSCW (e.g. BSCW [12], Futplex [77] and Mushroom [93])

A space-time matrix [86] is often used to denote the two principle characteristics of CSCW. They are

- the form of interaction (synchronous versus asynchronous),
- the geographical nature of users (remote versus co-located).

Table 2.1 shows the matrix. This characterisation may be influenced by the broader work of DeSanctis and Gallupe [38].

	Same Time	Different Time
Same Place	Face-to-face interaction	Asynchronous interaction
Different Places	Synchronous distributed interaction	Asynchronous distributed interaction

Table 2.1 *CSCW space-time matrix*

In short, collaborations can occur synchronously or asynchronously. Referring to the space-time matrix, a synchronous collaboration occurs in real-time (or “near” real-time if considering network latency) either at the same place or different places. If it occurs at the same place, a face-to-face collaboration can be easily conducted and if the participants are at different places, telephones, audio-visual conferencing or synchronous groupware (e.g. chat, shared workspace) are required. On the other hand, an asynchronous collaboration does not require participants to be available at the same time. If they are at the same place, collaboration is done through serial working but if they are at different places, letters or asynchronous groupware (e.g. e-mails, World Wide Web, threaded discussion forum) are required.

As this research focuses on Internet-based synchronous collaborations, unless mentioned otherwise, the term *collaboration* throughout the following chapters is referred to as remote (or networked) real-time (or synchronous) collaborations.

2.1.2 Advantages

Collaborations based on clear mutual goals and through proper implementations, either naturally face-to-face or electronically through information and communication technology (ICT), are recognised to be effective and efficient ways to achieve vital objectives. In management, many complex work processes require a variety of skills and experience to accomplish the objectives successfully. They require a team approach — individuals or groups of individuals collaborating on common objectives. Productivity increases and cost decreases when knowledge and expertise required in one area of the team is available to the entire team. With ICT working environment is no longer confined to physical office space — telecommuting is possible and CSCW makes it both efficient and effective.

In education, cooperative learning is a well-known approach for effective learning [99]. It encourages students to be active learners rather than passive observers. It involves

working together on some tasks or issues in a way that promotes individual learning through processes of group collaboration. It also contrasts with the traditional educational system which encourages competition on a *zero-sum* basis — whatever one person wins, the other loses. On the other hand, in cooperative learning or *non zero-sum* learning situations, all learners can do equally well if they cooperate. A study by Johnson and Johnson [87] proved that cooperative learning methods lead to higher achievement than competitive and individualistic ones. Currently there is a growing acceptance for integrating Computer-Supported Cooperative Learning (CSCL) [96] in coursework enrichment activities as it promotes enjoyment of learning through interactive multimedia and collaboration. In distance learning, cooperative learning provides the necessary interaction and fosters a sense of community for those isolated by geography or other constraints [128]. The Internet is seen as the most potential technology to promote collaborative distance learning. In cyberspace, students as well as teachers can visualise and rehearse in real-time the activities of CSCL. Solving simulated or real world problems over the Internet will encourage group learning even though the participants are far apart. Related computer jargons like “virtual university”, “virtual campus” and “cyber campus” are now becoming common terms to many people.

Currently the Internet is also being exploited for collaborative games and leisure activities. A growing number of Web sites are hosting Web-based multi-player games, chat rooms and all forms of collaborative entertainments. One example is KasparovChess.com [90] which is designed by World Chess Champion Gary Kasparov where, in addition to playing online chess, members can have their own @KasparovChess.com webmail addresses, a place to chat live, message boards, and interactive events where they can compete for prizes. Other examples are Yahoo! Games [155], MSN Gaming Zone [104] and PLAYSITE [118] that offer many different kinds of card games, board games and action games. There are also desktop computer games that support multi-players over LAN or the Internet (e.g. ChessMaster [102] and FIFA2000 [44]). With Internet-enabled features the whole world can be collaborative partners as well as rival “enemies”.

2.1.3 Applications

Collaborative applications have developed over a period of time. Early applications can be traced back as early as in the '60s. Douglas Engelbart [47] demonstrated NLS (after on-Line System) which contained what we now consider to be standard groupware applications: e-mail, shared annotations, shared screens, telepointers and audio/video conferencing. In the 70's there was a widespread deployment of asynchronous groupware, such as e-mail over the Arpanet (later the Internet) and threaded text conversations through conferencing systems and bulletin boards. In the 80's and 90's groupware became commercialised when products like Lotus Notes (and Domino, its Internet-age successor) [152], Novell GroupWise [110] and Netscape Collabra [107] (which is currently bundled with SuiteSpot) were launched. These groupware feature asynchronous collaboration facilities which, in addition to e-mail and threaded discussion system, are tools for development and deployment of document management system, workflow applications, group calendaring and scheduling. Support for collaboration of these products is very limited. Update propagation occurs according to predetermined schedules, precluding synchronous working.

There were also many Internet-based public domain and commercial groupware introduced during that period. One example is Cornell University's CU-SeeMe [42] (now licensed by White Pine Software) which boasts not only of a whiteboard, a file-transfer utility and a text-based chat but also multipoint audio-videoconferencing. ICQ [80] is another example which features, in addition to basic collaborative tools, an automatic searching facility for on-line friends or associates on the Net. Other examples are Netscape Conference [108], Vocaltec IC Pro [74] and Intel ProShare [81]. When Microsoft flooded the market with their free NetMeeting [137] with features include chat, whiteboard, file transfer, audio-video conferencing, application sharing and remote-desktop sharing many people were exposed to the power of Internet collaboration. But all of the above groupware are general-purpose collaborative tools — they may help remote users to collaborate in one way or another but the “real” collaborative applications that directly support group's collaborative work need to be designed and developed by professional

programmers with special development tools and specialised working knowledge of the field.

There are many domain-specific networked collaborative applications developed to support diverse synchronous collaboration activities. The examples are enormous. Major applications can be found in the field of education. Groupware have been used to support group learning (including distance learning) delivery and management. Some examples are EDS [105], TurboTurtle [28], CoVis [62] and Global Change [56].

EDS Collaboration [105] shows how the integration of various common collaborative technologies leads to an effective and cost-saving alternative to face-to-face instructions. The tools used are e-mail, audio-videoconferencing, IRC [113], NetMeeting [137], Virtual Places [51] and WorldsAway [155].

TurboTurtle [28] is a good example of how collaborative application is used to explore science subjects or more specifically Newtonian physics. Students experiment in a simulation environment with concepts such as gravity, friction, force, velocity, etc. and see how changes in their value affect the moving object. Another notable example is CoVis (Collaborative Visualization) Project [62] where participating students study atmospheric and environmental sciences through inquiry-based activities in collaboration with remote students, teachers, and scientists. Using state-of-the-art scientific visualization software, specially modified for an appropriate learning environment, students can have access to the same research tools and data sets used by leading-edge scientists in the field.

Collaborative application is also used to promote the investigation of global warming concept within a fully immersive, 3-D, virtual reality based model of Seattle [56]. As expected, most students thoroughly enjoy their experiences with virtual global change phenomenon while enhancing their knowledge about the real environment they live in.

There are also many CSCW systems developed for the office environment. Examples of such systems are TeamWorkStation [82], GroupDesk [55], TeamRooms [124] and wOrlds

[51]. There is also an application of a Group Support System (GSS) for cooperative policymaking [148].

Examples of different types of collaborative applications are collaborative Web browsing and group authoring, e.g. GroupWeb [63]. Last, but not least, there are also tools for collaborative video analysis, e.g. CEVA [27]. In short, the list of collaborative applications is only limited to human imagination.

Our study [71] showed that most of these collaborative applications fall into the following six groups:

- (a) collaborative work environment (such as engineering design, visualisation, documentation);
- (b) meetings, seminars and conferences over the Internet;
- (c) simulation of face-to-face contacts where visual quality is critical (e.g. recruitment interviews)
- (d) distance learning environments (course materials, tutorials, team projects);
- (e) networked computer games;
- (f) leisure and entertainment.

Each group typically requires different communication media through which remote collaboration can take place.

Table 2.2 summarises the different roles of communication media for different groups of applications. The first four media types are reasonably well supported by general-purpose groupware. In contrast, implementation of a shared interactive canvas or workspace is generally application-specific and usually requires considerable knowledge of system and network programming as well as knowledge in the subject field itself. This is reflected by the fact that applications in groups (a), (d) and (e) are poorly supported by the currently available development tools. As can be seen in Table 2.2, although the implementation of 3D virtual space requires a range of modelling and programming skills in addition to

media	(a)	(b)	(c)	(d)	(e)	(f)
text-based online chat	✓✓✓	✓	✓✓	✓✓✓	✓	✓
voice conferencing	✓✓	✓✓✓	✓✓✓	✓✓	✓	✓
video conferencing	✓	✓✓✓	✓✓✓	✓✓	✓	✓
shared applications	✓✓✓	✓✓✓	✓✓	✓✓✓	✓	✓
shared interactive canvas	✓✓✓	✓	✓	✓✓✓	✓✓✓	✓✓
3D virtual space and embodiments	✓	✓	✓	✓	✓✓	✓✓✓

✓✓✓ : necessary ✓✓ : desirable ✓ : occasionally useful

Table 2.2 *The role of communication media in different collaborative applications*

powerful graphics hardware, there is limited practical interest in 3D virtual space except for applications in groups (e) and (f).

2.2 Collaborative Virtual Environments

2.2.1 Definition

The term virtual environment is directly related to virtual reality — a computer-based application that provides a human computer interface so that the computer and its devices create a sensory environment called the virtual world [89]. This sensory environment is dynamically controlled by actions of the individual in a way that the virtual environment appears real to the user. A network-based virtual reality allows virtual worlds in the server to be controlled and visualised by the remote client workstation.

Collaborative virtual environment (CVE) applications allow multiple users to interact over a network in real-time and participate in the shared activities in a virtual world [127]. CVE offers the sense of group awareness or telepresence through avatars: synthetic bodies that populate a 3D world. Each one views the virtual world through his or her own “eyes”.

2.2.2 Related Research

The progress of CVE is directly related to the available technologies at its point of time. As briefly mentioned in Chapter 1, the early approach was through MUDs (originally stood for Multi-user Dungeon, but now popularly known as Multi-user Dimensions or Multi-user Dialogues) [48, 36, 134]. MUDs, together with their offspring MUD Object-Oriented (MOO) [49], are still being used particularly for educational purposes (e.g. Virtual Online University [76]). Integrating with World Wide Web, MOOs were then transformed to WOOs (Web-based MOO) [see 39, 128]. One notable example is INSTRUCT [128] which stands for *Implementing the NCTM School Teaching Recommendations Using Collaborative Telecommunications*. It blends synchronous and asynchronous technologies like World Wide Web, MUDs and e-mails for distance teacher training in the area of mathematics education. With this, the teachers can connect to on-line hypermedia educational resources, attend synchronous “meetings” with other INSTRUCT users and join in asynchronous discussions. Early attempts for graphical CVEs were implemented through the X-Window systems (e.g. XTV [2]). There are other systems, like WTV [3], that replaced X-Window with Microsoft Windows but it limits the system to run only on one platform.

In Europe, probably the most acknowledged research project in CVEs is DIVE (Distributed Interactive Virtual Environment) [24, 68]. DIVE is an Internet-based multi-user VR system which allows participants to navigate in 3D space to see, meet and interact with other users and applications. Developed by the Swedish Institute of Computer Science (SICS), DIVE supports the development of virtual environments, user interfaces and applications based on shared 3D synthetic environments. The first version of DIVE appeared in 1991 and is still being referred to by many researchers.

There are many ongoing joint European projects in this area. COVEN (Collaborative Virtual Environments) [145] is a project of the European Union Advanced Communications Technologies and Services Programme. The target issues of the research are collaboration support within virtual environments (including awareness,

communication, group interaction, etc.), corresponding requirements at network and platform level (involving concern for scalability and continuous media support), and aspects related to Human Factors. Another project, DEVRL (Distributed Extensible Virtual Reality Laboratory) [131] is funded by EPSRC (ROPA) with the aim of developing a shared and extensible virtual laboratory. Lancaster University, being one of the members, is developing a virtual physics laboratory that can be used for educational purposes as well as demonstrating physical principals of science in CVEs. Other significant projects are COMIC (Computer-based Mechanisms of Interaction in Cooperative Work) [123], funded by ESPRIT III Basic Research Action for developing general techniques for CVEs, and UK's VirtuOsi (Virtual Organisation) [10] project funded under the EPSRC/DTI CSCW Programme focusing on the industrial application of CVEs. Other notable projects are MASSIVE (Model, Architecture and System for Spatial Interaction in Virtual Environment) [64, 65]. Developed by University of Nottingham and funded by BT/JISC, these projects are the outcome of many pieces of research on CVEs.

DIVE is also one of the earliest tools designed for developing CVE applications. Being an early tool, it is platform specific and its latest version is currently available for SGI, Sun and Windows NT. Lancaster University has developed various tools and techniques for CVE. Some of the tools and techniques are AC3D [29] (a 3D modeller for developing virtual environments and objects), COLA [144] (A platform for sharing Cooperating Objects in Lightweight Activities), DNP [146] (Designer's Notepad: a tool for visualisation and arrangement of ideas) and SOL [132] (Shared Interface Object Layer: a shared object toolkit for cooperative interfaces).

Yarn Web [153] is one of the synchronous collaboration tools for World Wide Web. This Web-integrated electronic meeting system uses a separate X-Window for collaborative activities. Broll [20], on the other hand, proposed an extension to VRML to enable the support of collaborative virtual environments. Since VRML integrates nicely with Web browsers with proper plug-ins, a Web-based interactive multi-user virtual reality is proven real.

2.2.3 Virtual Realism Versus Effective Multimedia, Interaction and Communication

From the discussion presented above we have seen that enormous efforts are focused on 3D virtual environments and other issues related to virtual environments, such as awareness, scalability and human factors. From the standpoint of this research, the fundamental objective of most collaborative applications is to facilitate effective interaction and communication among users who are engaged in joint activities. The most important elements of such applications are their multimedia contents, users' interaction with computers and their communication with one another. This research recognises that the effort for making the virtual world to look real must not undermine the effectiveness of the elements of multimedia, interaction and communication.

2.3 Distributed Environments and Collaborative Tools

Developing synchronous collaborative environment applications can be extremely difficult. Implementing even the simplest system is a lengthy and tedious process. Among others, every application must deal with creating and managing socket connections, parsing and dispatching inter-process communication, locating other users on a network and connecting to them, and keeping shared resources consistent between users [125]. By using conventional programming tools a lot of low-level code must be written before getting to the logic-specifics of the application.

2.3.1 Distributed-processing Environments

Programming distributed-object systems that take into account heterogeneous machine architectures at physically distant locations is undeniably complicated. A number of distributed processing environments have been introduced to reduce the complexity of these tasks. Some leading ones are the OSF's DCE [114], the OMG's CORBA [129] and

Microsoft DCOM [46]. These *de facto* standards are in addition to the ISO and ITU-T standard known as Reference Model for Open Distributed Processing (RM-ODP) [83].

The Open Software Foundation's (OSF) Distributed Computing Environment (DCE) [114] provides a series of services for distributed programming. Despite the fact that some parts of its design and implementation are superior, its lack of support for object-oriented programming makes it unpopular.

The OMG's Common Object Request Broker Architecture (CORBA) [129] is more widely accepted. Despite its immaturity in some areas, their adoptions of a fully object-oriented architecture ensure their dominance over DCE [8].

Microsoft's Distributed Component Object Model (DCOM) [46] is a popular choice in Microsoft products environment. DCOM is among several software components that use COM technologies to provide interoperability with other types of COM components and services. Microsoft dominance over the "standard" slows down the adoption of this specification by other vendors. With the wide availability of Microsoft products (e.g. Windows, Internet Explorer, Office, NetMeeting and all Microsoft software development tools), the Component Object Model (COM) and its related COM-based technologies of DCOM, COM+, MTS and ActiveX comprise probably the most widely used component software model in the world. To promote COM-based technologies, Microsoft is assisting Metrowerks in porting COM to the Apple Macintosh, and the company is also working with Bristol and MainSoft to port COM to Unix [103]. However, at present, using COM objects (including DCOM) limits the platforms on which the application will run.

The ISO - ITU-T Reference Model for Open Distributed Processing (RM-ODP) [83] is an open standard. Like many formal standards the acceptance is driven by industry support. Unfortunately, none of the distributed-processing environments above implements all the specifications described by RM-ODP. Each one is a closed environment and it allows interoperation with other objects from the same environment.

Nevertheless, the many standards discussed above are only useful if supported by the development tools used by the programmers.

2.3.2 Java for Developing Collaborative Applications

The first alpha version of Java was published by Sun Microsystems on the Internet in May 1995 [22]. Since then it has received one of the most enthusiastic responses of any programming language in the history of computing. Much of the enthusiasm arose because Java was introduced when the then-emerging World Wide Web were overloaded (and still are) with all kinds of plug-ins and helper applications that need to be searched, downloaded and installed before the visitor can actually see any of the information on that site. Java solved this burden by allowing the Web publishers to publish both information and the application and by doing so the application can be delivered automatically to the visitor's site regardless of what platform the visitor was using. For that reason Java is known as "the programming language for the Internet" [13]. Also its "write once, run anywhere" [119] proved to be one of the biggest strengths of Java.

Java networking is based on the open standard protocol suite of the Internet known as TCP/IP (Transmission Control Protocol/Internet Protocol). Java includes several networking interfaces for Web-based and socket-based network communications. Java network programming can also exploit higher-level method calls like its native Remote Method Invocation (RMI) and CORBA distributed-object technologies. With RMI it is possible to invoke methods on objects in the remote virtual machine just as if they were normal local objects. Java's support for CORBA, the industry standard for distributed objects, makes it possible to interoperate with other CORBA objects developed in other languages.

As the only high-level language with cross-platform, secure, object-oriented, network-centric and native Web browser support, Java is the best language for developing network

applets or applications. Likewise, developing Internet-based synchronous collaborative applications in Java is the most practical choice.

2.3.3 Java-based Collaborative Frameworks

Java-based collaborative frameworks sit on top of Java networking interfaces. Many current collaborative frameworks are developed around the Java language which make them platform independence and nicely integrated with the World Wide Web. Abdel-Wahab [1] introduced mechanisms to intercept, distribute and recreate user events that allow single-user Java programs to be shared among collaborative participants. Calling their prototype JCE, after Java Collaborative Environment, the architecture is based on the “replicated architecture” of application sharing where a copy of the shared applications runs locally at each site and input events to each application are distributed to all sites. Replacing the standard `java.awt` toolkit by their `collawt` toolkit, in addition to the standard methods, the new component can handle communication issues, conference management issues and floor control issues. As each event by a participating user is triggered to all other remote users, this approach may introduce network storms.

Another similar tool is NCSA Habanero [25]. It is known as an object-sharing framework which can be used by developers to transform single-user applications into multi-user shared applications. This framework has been used for developing collaborative applications like shared whiteboarding, text and audio chat, collaborative text editing, voting tool and collaborative games. One limitation of Habanero is that integration with the Web is through helper applications — not directly as applets embedded on the Web page. The Habanero project has taken its course when the DARPA grant that funded the effort ended.

Sun Microsystems introduced comparable collaborative framework called Java Shared Data Toolkit (JSDT) [54]. Although it was developed by the creator of Java, JSDT is a toolkit, independent of Java extension. The JSDT’s strength lies in the monitoring and

controlling of multiple clients as they join and leave shared objects. It also has the ability to link participants intending to do collaborative works. This is unlike RMI which enables connecting two participants with method calls while letting the participants find each other. Confidence in JSDT among developers is fading due to the problem of support and licensing from the Sun Microsystems (refer to JSDT mailing list archive [139]).

Other Java-based collaborative frameworks which adopt a different approach from JCE, Habanero and JSDT are TANGO [9] and Promondia [57]. Sponsored by US Air Force Rome Laboratory, TANGO is extensive in features as its support asynchronous and synchronous collaborative communication. It is also extensible for C/C++ and Javascript. Promondia, on the other hand, is an early attempt to exploit the elegance and portability of Java code. Its focus is on group conferencing using a shared whiteboard, video and chat systems. There are also commercial Java collaborative tools available but their features are comparatively inferior to JCE, Habanero TANGO and JSDT. They are Eventware by Collaborative Systems Research [35] and Hesse by Praxis Technical Group [121].

The Java-based collaborative frameworks discussed above can minimise programming difficulties in a limited way. Knowledge of the programming language where these collaborative frameworks would be embedded is still required. An alternative programming technology must be studied to make further simplification possible.

2.4 Scripting Languages

Coding in scripting languages is known to be a “quick and dirty” way of doing programming. Nevertheless scripting has gained its popularity. It is predicted that scripting languages will handle many of the programming tasks in the 21st century in much better ways than system programming languages [116]. One of the factors that boosts the acceptance of scripting languages is the Internet.

Comparing the “traditional” programming languages with scripting languages, the creator of the scripting language Perl, Larry Wall [43], described the former as being about *orthogonality* or *minimalism* (express in the least number of features everything one might want to do), whereas the latter are about shortcuts and shortcuts are about going at an angle or *diagonal*, in metaphorical terms. The fundamental concept of a scripting language is *simple, direct to the point* — the primitive operations have greater functionality than the conventional ones. Scripting languages are normally interpreted [117] and many emphasise “gluing” external applications written in other languages. Thus, scripting languages are sometimes referred to as *glue languages* or *system integration languages* [116].

Ever since the Unix shell scripts [95], there have been many scripting languages introduced. One famous scripting language is HyperTalk, a scripting language of Apple Macintosh’s HyperCard environment [6]. This fairly easy-to-learn scripting language became popular due to the fact that it was bundled free with every Macintosh sold and it appeared during the time when people were enthusiastic about hypertext. Even though the designer of HyperCard, Bill Atkinson, admitted that it was not really a hypertext product from the beginning, HyperCard was said to be the most famous hypertext product in the world in the late 1980s [109]. Due to its popularity, the expectation that scripting languages simplify programming tasks became more apparent.

Currently the “big three” [97] scripting languages are Perl, Tcl and Python. Newer languages are VBScript and JavaScript. In the Internet environment some scripting languages are used extensively *within the Web server programming interface* and *as back-end gateway to other services*. There are also scripting languages used *as a user front-end* and *within the Web client-user interface* [91]. With scripting languages, Web pages are no longer simply static HTML but also active scripted applications. This facility allows more processing at the client (and the server) and, thus, reduces wasteful use of network bandwidth. Other lesser-known scripting languages include Rexx, Scheme, Guile and Icon.

2.4.1 Perl, Tcl and Python

Perl (Practical Extraction and Report Language) [149] is freely available for many operating systems. Perl's powerful text-manipulation functions makes it suitable for many system command languages. Recent popularity for programming dynamic World Wide Web with on the fly HTML page creation promotes it to be "the" language for the Common Gateway Interface (CGI) — a standard for external gateway programs to interface with information servers such as HTTP [15] servers.

Tcl (Tool Command Language) [115] is another strong scripting language. Tcl features are represented by commands. Commands can be of built-in (statements, expressions, control-structure) or of created command procedures written in C or C++. There are also numerous extension packages that can be incorporated into any Tcl application. One of the best known extensions is Tk (hence also known as Tcl/Tk), a programming environment toolkit for creating graphical user interfaces under X-Window, Windows and MacOS. Due to the fact that Tcl and Tk are easy to learn, yet powerful, and contain many sophisticated features, they have dramatically reduced development time for much GUI programming.

Python [150] is another scripting language which is gaining popularity. Unlike Perl and Tcl, Python has a strong model of object-oriented programming and is said to be most suitable for "programming in the large" [97]. It includes many modern programming language features together with many useful standard packages. Programmers may extend Python to interface to other arbitrary software components. Python can be used for CGI scripts, system administration, code generation, graphical user interfaces, file-format conversions and for general software engineering and product development.

2.4.2 JavaScript and VBScript

JavaScript [52], originally called LiveScript, was developed by Netscape Corporation for use in its browser Netscape Navigator. Currently Microsoft Internet Explorer also supports

the feature. The syntax of this scripting language is based on Java but is made more concise and simpler. The source code written in JavaScript is placed into a Web page and, since the interpreter is built into a Web browser, the source code can run without being compiled. JavaScript uses *objects* but it is not an object-oriented programming language. There are standard objects that enable the programmer to access features of the browser (e.g. windows, documents, frames, forms, links, and anchors) directly. Some common uses of JavaScript are embedding dynamic information, validating forms for CGI processing and making pages interactive. This scripting language also poses some security concerns which, among others, information on client system can be sent to the Web server. JavaScript is suitable for small applications such as to make the static HTML [16] “alive”.

JScript [85] is Microsoft’s implementation of JavaScript. JavaScript and JScript served as the basis for ECMAScript [45] — the only standard scripting language on the Web. The ECMAScript specification outlines an object-oriented programming language for performing computations and manipulating objects within a host environment, such as the browser.

Microsoft Visual Basic Scripting Edition (VBScript) [130] is a subset of Microsoft Visual Basic for Application (VBA). Its lightweight interpreter is used in Microsoft Internet Explorer and other applications that use Microsoft ActiveX controls (formerly called OLE controls) [26]. Like JavaScript, the VBScript interpreter processes source code embedded directly in HTML. Unlike JavaScript, VBScript can also be used for Web server scripting. VBScript talks to host applications using ActiveX Scripting. Technically, ActiveX controls are among Microsoft’s Component Object Model (COM) technologies that provide interoperability with other types of COM components and services [103]. Together with DCOM (refer 2.3.1), VBScript can be an important part of client/server programming. One major concern about VBScript is that it is only being supported by Microsoft Internet Explorer (through MSIE Object Model) at the client side and Microsoft Internet Information Service (through Active Server Object Model) at the server side. Its platform-dependence makes it somehow unpopular.

2.4.3 Other Scripting Languages

Other examples of scripting languages are Rexx, Scheme, Guile and Icon. Rexx's [59] strength is its historical base in IBM commercial systems. It may not be adequate for modern problems. Scheme [135] and Guile [61] are in the LISP family and are quite popular in GNU projects. It may only be suitable for those who are familiar with LISP. Icon [66] is a good language for text processing but its popularity is overshadowed by Perl.

2.4.4 Scripting Language for Interactive and Collaborative Applications

All scripting languages mentioned above are general-purpose languages. Other than as "glue languages", they are not well-suited for developing real-time, complex multithreaded shared-memory or large applications. They are meant to complement system programming languages with which software components are built. While Python can be extended to support distributed objects [8], a system programming language like Java is currently still the best. Java with its "object-oriented, distributed, robust, secure, architecture neutral, portable, high performance, multithreaded, and dynamic" [138] features is the best language for creating collaborative components. These components can be glued with a "simple, direct to the point" scripting language. A new language which employs the design philosophy of scripting language and built on top of the strength of system programming language Java is seen favourable and promising.

2.5 Summary

This chapter has shown that collaboration is an effective way to share views and to synergise efforts among a group of people in the pursuit of a common goal. With electronic collaboration, temporally and geographically dispersed users can carry out collaborative activities beyond regular practices. The Internet together with various

collaborative frameworks have given rise to the development and the deployment of many collaborative applications. If integrated with the Web, participants don't just surf for resources on the Internet but actively and interactively contribute to exploring innovative ideas and building relationship globally.

While collaborative frameworks reduce programming difficulties, scripting languages are the way forward in programming. Programming in scripting language, which is high level in nature, requires less code, effort and development time. A special-purpose scripting language for developing networked interactive and collaborative applications can also be an effective tool for the casual programmer in the era of the Internet. However, scripting languages alone cannot solve all the computer-supported problems. As "glue" languages, the scripting languages have to be used (or built) on top of or side by side with other system programming languages. Java is currently the best language for Internet-based client/server programming. Java collaborative framework components can be integrated or developed on the fly by the scripting language, thus hiding from the programmer the details of real-time multi-user network programming.

Chapter 3

Design Considerations

The dynamic nature of networked interactive and collaborative applications promises a potentially powerful facility for human-computer interactions and human-human collaborations. With the Internet as the infrastructure, interactions can occur either synchronously or asynchronously beyond physical boundaries and normal practices. The dynamic nature of such applications also leads to a potentially chaotic environment for the remote end users as well as the active server. Without a careful and comprehensive study of the design and implementation of this type of applications, as well as the underlying means that drive the interactions and collaborations, effectiveness and efficiency can hardly be achieved. Furthermore, the most important question is how to provide these facilities so that they can be within the reach of many casual programmers.

This chapter first looks at typical interactive and collaborative applications so as to understand their features and requirements. It then elaborates on the design principles of the JACIE language, which are *special purpose* and *programming efficiency*. It also discusses some design issues and how they promote towards the design principles. The design issues are:

- (a) Why do we resort to scripting language for developing interactive and collaborative applications? How does it simplify the task?
- (b) Why do we use template-based structure and what are its essential components?

- (c) What are the interaction protocols and how do we handle various protocols efficiently?
- (d) What are the required communication channels and how can JACIE overcome any inherent weakness?
- (e) What are the optimum graphic supports that balance out between powerful features and the need for simplicity?
- (f) What are other considerations that can promote simplicity but not at the expense of user and programmer requirements?

3.1 Typical Applications

The understanding of the nature of typical interactive and collaborative applications has provided the sound basis for this research. It gives an idea of various modes of interactions, special feature requirements and the right tools for the development.

While most common basis for the categorisation of collaborative applications is by where and when the interaction takes place (refer [41,67,86] and the space-time matrix discussed in Section 2.1), this research has categorised the applications according to their mode of interactions [see 71]. Partly influenced by the Cooperative Work Framework (Figure 3.1) of Dix et al. [41], this study classified the interactive and collaborative applications into three categories — *server-based interaction*, *server-mediated interaction* and *group collaboration*.

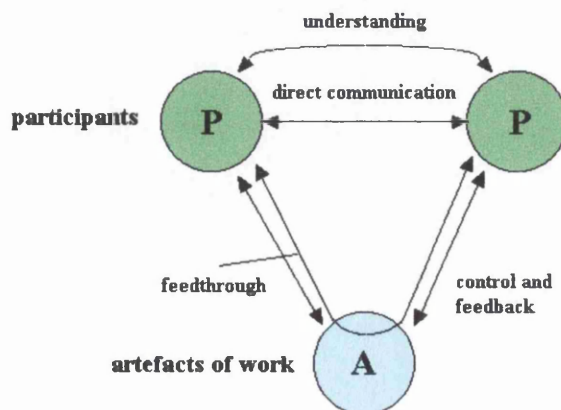


Figure 3.1 Framework of cooperative work

Modes and Features	Typical Applications
Server-based Interaction: comm. sync.: not essential display sync.: not essential	On-demand or time-scheduled presentation (slide, audio and/or video) Server-based interactive courseware, and online testing Online services, such as shopping, banking, and server-based computer games Accessing and modifying shared information (e.g. group calendar, distributed databases, online voting system, etc.) Audio/video on demand
Server-mediated Interaction: comm. sync.: desirable display sync.: necessary	Text, voice and video conferencing Multi-user slide presentation Unstructured shared space, such as public chat place, shared whiteboard Unstructured multi-player games (such as distributed architecture walks) Group decision support systems
Group Collaboration: comm. sync.: essential display sync.: essential	Teamwork courseware Collaborative authoring (writing, programming, etc.) Structured multi-player games (such as remote card games) Other collaborative applications with structured shared space or multiple inter-related displays

Table 3.1 *Types of interactive and collaborative applications*

Table 3.1 summarises the features of each category together with some example applications. The term *communication synchronisation* refers to the requirement for handling inter-process cooperation [142] among remote clients (and the server), whereas *display synchronisation* refers the requirement for maintaining *what you see is what I see* (WYSIWIS) [41] principle also among remote clients.

3.1.1 Server-based Interactive Applications

The server-based interactive mode is a type of interactivity where information or an application is placed on a server to be accessed and executed by any user on the network. Even though many users can access and execute the application at the same time, each user is treated individually and interactions only occur between the users and the server. In this mode, the server role is minimum. Since each active client application is independent of the others, neither communication synchronisation nor display synchronisation is

needed. Theoretically, any number of users can be connected simultaneously. The only restrictions are the server hardware and network resources. For that reason it is probably wise for the programmer to limit the number of active users.

As shown in Table 3.1, some of the applications in this category are: on-demand or time-scheduled networked presentations; interactive server-based courseware; online testing; single user networked games and other network-based services like online banking, shopping, group calendaring and voting. For the last example, even though the remote users are accessing and updating a common database, there is no direct interaction between users.

Figure 3.2 shows graphically how the interactions between the client users and the server take place. There are four clients currently interacting with the server but none of them is interacting with each other.

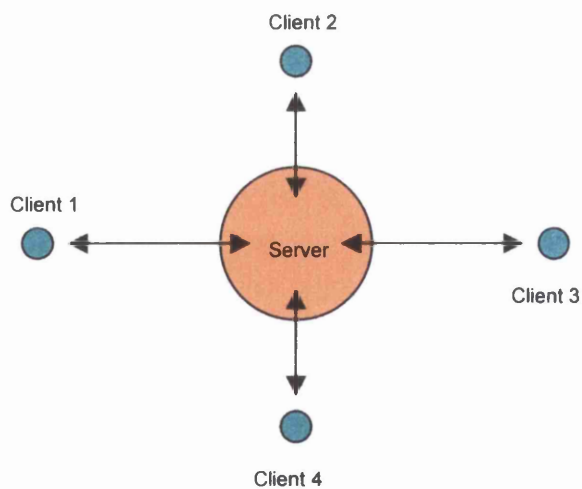


Figure 3.2 *Client-server interactions in server-based interactive applications*

3.1.2 Server-mediated Interactive Applications

An interaction is said to be *server-mediated* when the server plays an additional active role as a mediator to a number of interacting online users. Applications of this type normally are a little more complex than the first category as the server needs not only to maintain the state and behaviour of individual connections but also the state and behaviour of some common resources and operations. It becomes necessary for the server to maintain the consistency of the information viewed by all concurrent users. In most cases of such applications, it is desirable (if not necessary) to manage the communications among users in an orderly manner. A real-time turn control may be required to ensure coordination and synchronisation.

Table 3.1 shows some examples of server-mediated interactive applications which include: teleconferencing (text, voice or video conferencing); unstructured shared space; such as public chat place; shared whiteboard; multi-player games; multi-user coursewares; decision support systems, etc.

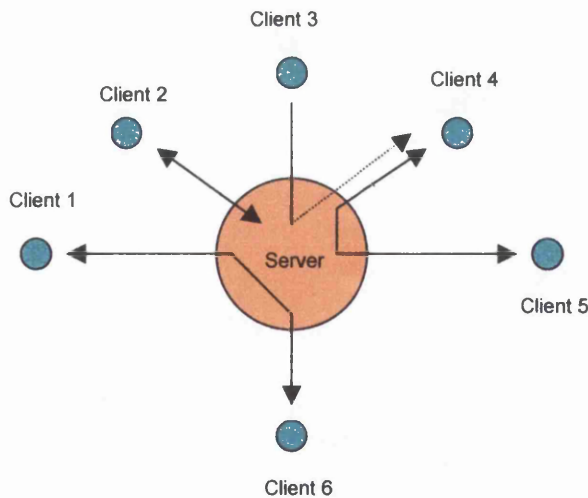


Figure 3.3 Multi-user interactions in server-mediated interactive applications

Figure 3.3 shows how interactions between remote clients take place with the server which acts as the mediator. In this five active client situation, *Client 1* is interacting with *Client 6*, *Client 3* is trying to interact with *Client 4* but *Client 4* is interacting with *Client 5* and *Client 2* is interacting with the server. Another possible scenario not shown in Figure 3.3 is where one client can broadcast to all the other clients.

3.1.3 Group Collaborative Applications

The final category of networked interactive applications is of type *group collaboration* or teamwork type. In this category the server is responsible for coordinating inter-group communications as well as intra-group communication and coordination. This extra role is in addition to maintaining individual connection and real-time turn control, if required by the application. Normally for this kind of applications, the application logics are divided into three — the client specific, the ones common to the group in which the client belongs and the ones common to all.

Some examples of this type of interactive applications are teamwork courseware, group collaborative authoring (writing, programming, etc.), group multi-player games and other collaborative applications with group support with or without private and public communication channels.

Figure 3.4 shows some typical interactions within the framework of group collaborations. The example shows eight active clients, divided into four groups with two members in each group. *Client 1* of *Group 1* is interacting with *Client 2* of *Group 2*. *Client 3* is interacting with all members in *Group 3*, namely *Client 4* and *Client 5*. *Client 6* is interacting with its own *group 4* member, *Client 7*. *Client 8* is interacting with the server. Also, not shown, it is possible that any one client can broadcast to all other clients.

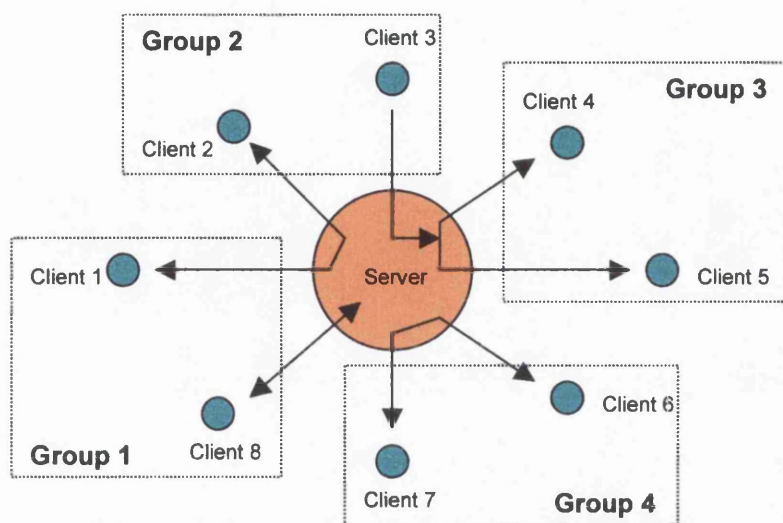


Figure 3.4 Interactions in group collaborative applications

The three categories of interactive and collaborative applications mentioned above are the ones targeted by the JACIE language. Some of the related issues need to be considered are the number of on-line users supported, the nature of their interactivity — whether they are independently interacting users (Category I), or they are interacting with each other (Category II), or they are grouped so that there are inter-group and intra-group interactions (Category III). For categories II and III, the protocols of interactions that define the rules that govern the means of interactions between users are required. Other issues are the communication channels to be used, the ways to handle shared resources and the application development.

3.2 Scripting Language for Interactive and Collaborative Applications

JACIE is designed to be a development tool for the above type of applications. While there are many programming approaches that can be adopted, JACIE was chosen to be a scripting language. Currently it is the only scripting language, among many scripting languages available, that is designed for this specific purpose.

3.2.1 Scripting Versus Other Alternative Programming Approaches

The most common approach for developing this kind of application is to use conventional programming languages. They are rich in features, robust, flexible and tailorable to strict user requirements. But, programming these network applications is tedious, time-consuming and requires an in-depth knowledge in low-level programming. Even Java, which is designed with the network support in mind, is not within the reach of many when it comes to network programming.

Another programming approach that simplifies coding is by introducing another layer to the programming language or by providing a software API library that is ready to be integrated into the code. Commonly known as called a software framework, this approach still requires the knowledge of programming techniques. The currently available collaborative frameworks (refer to Chapter 2) are still designed and adopted by researchers and hard-core programmers.

Another possible approach is a preprocessor. But none in the market adopts this programming style for developing collaborative applications. One possible reason is that the knowledge of the programming language, in which the preprocessor is embedded, still requires much effort.

Programming through scripting may be seen as the most appropriate approach if we were to allow more people to develop this kind of application. Even though, there is no specific scripting language designed for collaborative applications, the acceptance of general scripting languages available (e.g., JavaScript and VBScript) has been overwhelming. As scripting is *simple and direct to the point*, the primitive operations of a scripting language have greater functionality than the conventional ones. Many common features required by such applications — like handling and managing multiple users, managing interactivity through a predefined protocol, managing shared resources, managing channels through

which the communicating parties interact — can be hidden from the programmers but accessible through simple scripting specifications.

However, there is also an alternative technology that uses a visual programming style that reduces the burden of coding through graphical representations and manipulations. While the currently available visual programming tools are of general purpose and not suitable for developing networked applications, they are also normally built on top of programming languages. The knowledge of coding in a scripting language can be a good introduction to future visual programming technology for networked interactive and collaborative technology.

3.2.2 The JACIE Scripting Language

The decision on whether JACIE should be a scripting language or another programming approach has been made based on the following factors:

- (a) A software library/framework will not solve entirely the problems faced by the developers. The knowledge on the programming language, where a software library/framework would be embedded, will still hold the key to the effort and cost incurred.
- (b) Currently, the many scripting languages available (including the ones with Web-based support) have demonstrated their effectiveness and popularity. In most cases, the development of a scripting language involves a software library/framework in the target language to support the compilation of common functions. It is easy for a scripting language to provide a software library/framework as a side product, but not vice versa.
- (c) Visual programming for collaborative applications will be a natural progress from a scripting language. The key features of the scripting language can be made available through, possibly, graphical representations for easy manipulations.

In short, coding in the JACIE language suits the programmer of the above type of applications who may appreciate the short learning cycle of a scripting language. It promotes programming efficiency by reducing the complexity and inflexibility in function calls and parameter passing with traditional software libraries or frameworks.

3.3 Template-based Programming Style

3.3.1 What is a template-based programming style and how does it simplify programming tasks?

A template-based style divides each program into a set of standard components. While the recent programming approaches do not adopt template-based style programming, many programmers practice “template-based” in their programming work. Many programmers use, modify and expand the available codes without realising how the operations take place. The “free-form” style programming has become template-based in a semantic sense. Therefore, a carefully designed template-based language with common components related directly to the needs of the application developed should provide a better alternative to programming. For networked interactive and collaborative applications, a “single program” for the client and the server is possible if the template provides a proper placing for the client code and the server code within the same program. The coordination in programming codes for the client process and the server process will become simpler. In the “free-form” style programming, separate programs are required for the server and the client. Protocols (to be discussed in greater detail later) are required for coordination of interactivity. Separate programs for the server and the client, where each one has many classes or components, make the program more complex even for a simple client/server application.

3.3.2 JACIE's Template-based Programming Style

JACIE employs a template-based style in its scripting language. Adopting this technique, a coding task is just a matter of “fill in the form”. A template-based style also allows a networked client/server program to be written as a “single program” to specify both server processes and client processes (as opposed to normal client/server programming). The three main components of the template are the application configuration, the client implementation and the server implementation. The application configuration is for statements that specify basic networking and interaction parameters. The client implementation and the server implementation, on the other hand, are for statements that define the logic of the application. The three components are further subdivided into various constructs that simplify the coding tasks. A detailed discussion of these components, constructs and statements is presented in Chapter 4.

3.3.3 The Template Structure of the JACIE Language

The template structure of the JACIE language has been designed in consideration of the transition states through which a server and a client go in typical client/server interactions. The transition state diagram is shown in Figure 3.5.

The transition states for the server are as follows:

- i. **Starting state** — This is the initial state for the server process. In this state some initial housekeeping matters may be performed.
- ii. **Waiting state** — This is the state in which the server waits and listens to the socket for clients to make connection.

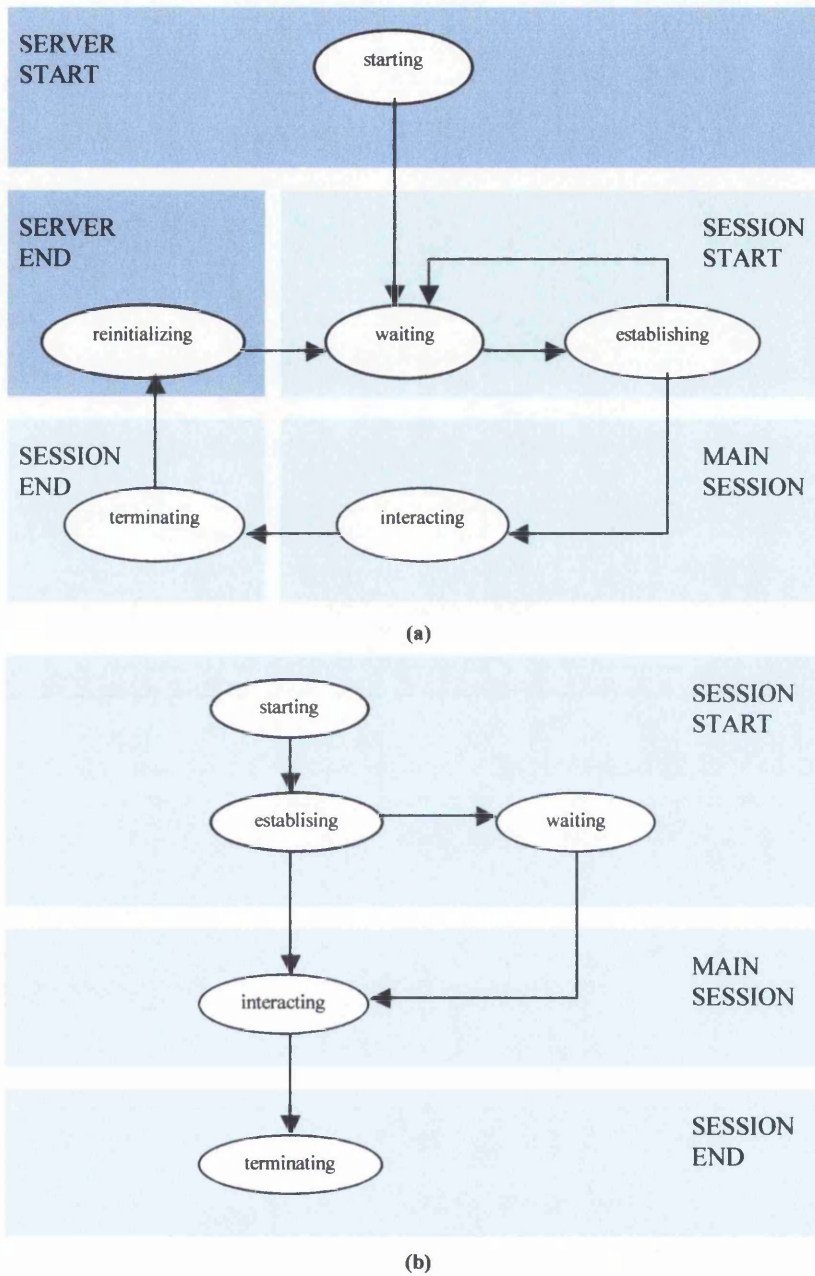


Figure 3.5 Transition state diagrams for (a) server process and (b) client process

- iii. **Establishing state** — When a client starts the communication, the server process goes into this state. If more clients are expected it will go back to the waiting state or otherwise it will proceed to the next state.

- iv. **Interacting state** — This is the heart of the transition state. In this state messages are exchanged during the active session. The connection remains in this state until either the client or the server wants to terminate the session.
- v. **Terminating state** — When the connection is in this state, some house cleaning for the specific client may be performed before closing the link.
- vi. **Reinitialising state** — This is the state where all client connections are gracefully terminated. Some house cleaning matters may be performed before the server goes back to the waiting state and ready for the next action.

The transition states for the client are similar, and directly related, to the transition states for the server.

- i. **Starting state** — This is the initial state for the client process. In this state some initial housekeeping matters may be performed.
- ii. **Establishing state** — This is the state in which the client requesting a connection to the known server. Upon acceptance, depending on the state of the server made known to the client, it will go to the waiting state or to the interacting state.
- iii. **Waiting state** — This is the state in which the client process is put on hold while waiting for a signal from the server to move to the next state.
- iv. **Interacting state** — Similar to the interacting state of the server, this is the heart of the transition state for the client. In this state messages are exchanged during the active sessions. The connection remains in this state until either the client or the server wants to terminate the session.
- iv. **Terminating state** — In this state, some house cleaning may be performed before closing the link. Several messages may be exchanged before the connection is gracefully terminated and the application is properly closed.

The transition states demonstrate that program codes which represent processes of interactive and collaborative environment applications can also be divided into the following program codes:

- i. program codes when the server process starts,
- ii. program codes when the client first establishes its connection,
- iii. program codes for the main client-server interaction during the active session,
- iv. program codes when the client leaves the session,
- v. program codes when all the clients leave the session and reinitialise the server process.

The template-based programming suits the target applications which share a set of common features. The above divisions reflect the template structure of the JACIE language. The five subdivisions are considered in the server implementation component. Since the first and the last items are for the server process, the client implementation only includes the second to the fourth. With template-based programming, coding in JACIE is just a matter of identifying and specifying which tasks are to be implemented in which states. As mentioned earlier, to a certain degree, coding in JACIE becomes form-filling. Its “single program” style eliminates the need for constructing server and client programs separately. This also facilitates better correlation between server and client functionality and easier software maintenance.

3.4 Protocol Handling

3.4.1 Overview of Protocols

A protocol, in general, is an agreed upon set of rules by which computers exchange information. An interaction protocol defines the rules that govern the means of interactions between user-user and user-server in a collaborative environment [71]. Without protocols, interactions and manipulations of shared resources can be chaotic, leading to unachievable collaboration objectives. Protocols promote coordinated actions in a multi-party environment. Except for low-level protocols, none of the available development tools has

a built-in support for high-level protocols. Each programmer has to design his or her own high-level protocol for each networked application he or she develops.

3.4.2 JACIE's Protocol Handling

In the context of JACIE, while low-level protocols are dealt with in the background, many high-level protocols are introduced. In general these protocols are either used by the server to execute its functions or are mechanisms for coordinating user-user interactions. This research categorises these protocols based on the functions which are *session management protocol*, *delivery management protocol*, *floor management protocol* and *group management protocol*. JACIE adopts and defines “standard” protocols so as to hide the complexities and potential errors in programming. Also, as one form of a high level abstraction, the “standard” protocols promote *simple* and *direct to the point* operations. The session management protocol and the delivery protocol form the software architecture of JACIE-generated programs and are discussed in Chapter 5.

3.4.3 Floor Management Protocol

In any situation where there are many people working collaboratively on some common tasks, unless there is an agreed upon rule that restrict each other's actions, it is likely that they may end up in a chaotic situation. Floor control is about restricting actions or, in terms of collaborative applications, restricting access to shared objects or any other common resources so that the collaborative process is coordinated and synchronised. JACIE employed a moderated style floor control where the `floor control manager` decides who has control access of the resources at any point in time. In other words, based on the predetermined floor control protocol, the `floor control manager` coordinates users' turns. By doing so this facility provides some kind of mediated access that can prevent unnecessary mistakes, unauthorised access and conflicting changes. Considering that there are situations when coordination is not required or situations whereby users are

to be involved in some kind of competitive activities, JACIE also allows the absence of floor control manager.

Various natural interaction and collaboration techniques have been studied [70] and made available as features of floor control protocol in JACIE language. The study observed that the most common collaborative technique practised is *in turn* or also known as *token passing* or *round-robin*. As each person gets his or her turn sequentially, this guarantees a fair chance to everybody. The drawback of this technique is when the person who is having the turn does not pass it to the next person. This leads to another technique, a *timed token*. This way the person is allocated a fixed time for him or her to act. If the time is over, whether or not he or she uses it, the turn will be passed on to the next person. These two techniques are also unfavourable in some situations because the time allocated may be wasted when the person who is having the turn has nothing to contribute to the group.

Another practical technique is *reservation*. This technique is customarily used in classroom (by the teacher) or during a question and answer session in conference (by the chairperson). By this protocol whoever wants to take part must raise his or her hand as a mean of attracting attention from the moderator (the teacher or the chairperson). The moderator will then reserve their turns in a queue and allow them to participate on a first-come-first-served basis. The only drawback of this technique is that passive participants will not be motivated to get involved in the activity.

There are also other collaboration techniques commonly practiced. One such technique could be referred to as *tapping*. This technique is normally used in game-like activities where the person who has a turn is expected to decide who gets the next turn. Another technique is *randomly-directed* (*random* for short) where one central figure randomly assigned the turn to any participant without any order. The last in the category of commonly used collaboration techniques is *contention*. In this style of interaction, as its name implies, there is no coordination at all. This technique is commonly used in competition-style activities. As expected, this uncoordinated technique can sometimes lead to chaos.

The JACIE language supports all the above mentioned natural interaction and collaboration techniques which are named as *round-robin*, *timed token*, *reservation*, *tapping*, *random* and *contention*. It is up to the programmer to choose any one of the techniques suitable for the specific application to be developed. As far as the JACIE scripting is concerned, besides just specifying the floor control protocol to be adopted, there is no other code required. The JACIE's `floor control manager` which coordinates floor control is generated accordingly by the compiler and is ready to support the server process on execution.

3.4.4 Group Management Protocol

Teamwork or group collaborative applications require protocols for grouping as well as protocols for intra-group and inter-group interactions. Naturally groups may be formed through mutual agreement among participants where a leader for each group is appointed and group members are determined through the consultation of these group leaders. Another technique for grouping is through time of arrival. In this way participants are grouped in alternate order of his or her arrival; e.g., if six participants are to be divided into three groups, the first person to arrive will be in group 1, second in group 2, third in group 3, fourth in group 1, fifth in group 2 and sixth in group 3. Another grouping technique is random assignment where a mediator will assign participants into different groups by a random order.

JACIE supports all the three grouping protocols under the name *user-defined*, *alternate* and *random* respectively. The same floor control protocols described above are also applicable to inter-group interactions. For intra-group protocol, in JACIE, the group members are allowed to decide among themselves which floor control protocol is to be adopted. Negotiation is expected to be made through private communication channels. Since the intra-group floor control protocol is not automated when the group receives its turn, any member of the group may act on behalf of the group. Also, considering that

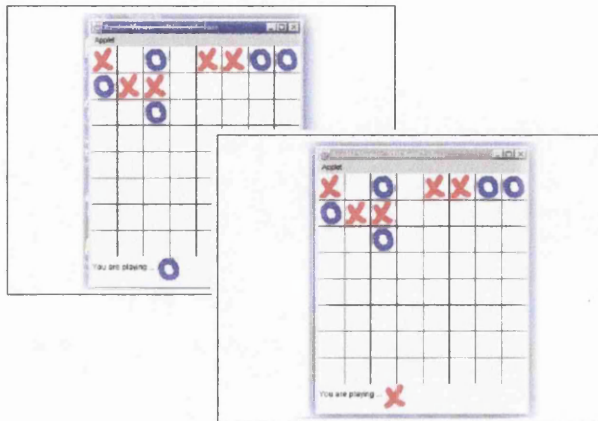


Figure 3.6 *Networked noughts and crosses game*

multiple steps may be required to be executed by the group during its turn, a pass turn facility is provided as a flag that the group has completed its turn. Upon receiving this flag, the floor control manager will then assign a turn to another group. In addition to the private communication channels made available for intra-group communications, the public communication channels can also be used for inter-group communications. Section 3.6 describes in more detailed various communication channels implemented in JACIE.

3.4.5 Interaction Protocols: A Case Study

The identification of interaction protocols discussed above is the result of a case study carried out at the early stage of the research work. A collection of networked noughts and crosses (tic-tac-toe) games were implemented in Java prior to the development of JACIE. Figure 3.6 shows the screen shots of the applets.

All games were designed to run across the Internet and users on different computers were interacting through Web browsers. Each game implemented a different rule that simulated a possible floor management protocol. Table 3.2 lists the main features of a selection of these games in this case study.

Name	Players	Board	Turn Control	Cell Control	Winning
Traditional Game	2 players	3x3 cells	Each player places his/her symbols in turn.	One symbol in an empty cell each time.	The first one getting 3 of his/her symbols in a line wins the game.
Generalised Game	2 players	8x8 cells	Each player places his/her symbols in turn.	One symbol in an empty cell each time.	The first one getting 3 of his/her symbols in a line wins the game.
Speed Fight	2 players	8x8 cells	Players place their symbols as fast as they can. No turn control.	One symbol in an empty cell.	The first one getting 5 of his/her symbols in a line wins the game
Vicious Fight	2 players	8x8 cells	Players place their symbols in any cells as fast as they can. No turn control.	An existing symbol can be replaced.	The first one getting 8 of his/her symbols in a line wins the game.
Gentlemen's Fight	2 players	8x8 cells	Each player signals the other player if he would like to play, and cannot place his symbols until he receives the permission from the other player (or the server).	One symbol in an empty cell each time.	The first one getting 5 of his symbols in a line wins the game
Dictator's Entertainment	2 players	8x8 cells	Each player must receive a signal from the dictator (the server) before placing a symbol, the dictator randomly selects a player each time using a random number generator	One symbol in an empty cell each time.	The first one getting 5 of his/her symbols in a line wins the game
Group Game	2 groups, 2 players each group	8x8 cells	Each player places his/her symbols in turn.	One symbol in an empty cell each time.	the first group getting 3 of his/her symbols in a line wins the game.

Table 3.2 Different versions of noughts and crosses in the case study for interaction protocols

The interaction method used in *traditional* and *generalised* versions of the game, where each player makes his or her move in turn, is an example of a round-robin (or token passing) protocol. For *speed fight* and *vicious fight* versions, the players make their moves as fast as they can (or practically as fast as the system allows) and without the restriction of turn control. The interaction method is what we call contention. The rule adopted by the *gentlemen's fight* version is an example of a tapping protocol. The *dictator's entertainment* version, where the server dictates who gets the turn, is an example of a random protocol. The *group game* version differs from the rest as players are divided into two groups. Both the inter-group and intra-group interaction rules adopt round-robin protocol. As mentioned earlier this particular study has helped the formulation of JACIE's built-in interaction protocols.

Briefly, protocol handling in Java is a unique and innovative feature that well suits the target applications and covers a range of commonly used interaction protocols. This also

reduces substantially the complexity and potential programming errors in specifying and implementing interaction protocols using low-level language constructs.

3.5 Event-driven Programming

3.5.1 Overview of Event-driven Programming

Many modern development tools support event-driven programming. The event process spends much of its time idling, waiting for an event to occur and when an event occurs, an event handler responds to the event [60]. With event-driven programming, the programmers are expected to sense the events and manage the event accordingly upon activation. Many of the common events supported by many development tools are user-interface events, but none considers having interactive and collaborative events as standard features. A number of programming tools allow the programmer to define new events and event listeners, but, as expected, this feature only introduces extra complexity for casual programmers.

3.5.2 JACIE's Event-driven Programming

The JACIE language adopts event-driven programming style. The transition states discussed in Section 3.2 represent some main events handled by this language. When events are generated at different points of the sub-processes, statements for event listeners would be able to respond accordingly and could pass to specific program codes appropriately and efficiently.

The research also recognised that, in addition to the five main events described above, there are also other events commonly used in collaborative applications. Some of these are session events defined in support of interaction protocols adopted. Others are low-level

events. Table 3.3 lists all the events handled by the JACIE language. It also shows how they are generated.

The event handling facility simplifies much coding effort as it promotes high-level abstraction as opposed to low-level construct. As far as the programming step is concerned, the programmers just need to be aware of what kinds of events expected and what are the steps to be taken, rather than how to generate the events and how to notify the active events. As can be seen, for session events, some of them are directly related to the protocol adopted, e.g., `turn`, `group turn`, `request control` and `reservation`. Also, related to the events, the JACIE language provides many system variables to be accessed by the programmer. Detail discussion is presented in Chapter 4.

Main Events	
<code>server start</code>	triggered when the server process is started and ready to listen to incoming connection
<code>session start</code>	triggered when connection is successfully established
<code>session</code>	triggered when enough participants are online
<code>session end</code>	triggered when the participant disconnects his/her session or the process is terminated naturally or the physical connection is interrupted
<code>server end</code>	triggered when all client connections are terminated
Session Events	
<code>waiting</code>	triggered when the required number of online users has not been reached
<code>observer connection</code>	triggered when a remote user is assigned an observer status; normally it will be used by server process to send the current states of the session
<code>turn</code>	triggered by the server when the current turn is the user's turn
<code>group turn</code>	triggered by the server when the current group turn is the user's group turn
<code>request control</code>	triggered by the user to flag the server to request for turn (in reservation floor control management)
<code>reservation</code>	triggered when the user has made request control but yet to get his turn (in reservation floor control management)
<code>client abort</code>	triggered when the client abort the program in the middle of the execution
<code>server abort</code>	triggered when the server abort the program in the middle of the execution
<code>new message</code>	triggered when message queue is not empty
Mouse Events	
<code>mouse clicked</code>	triggered when the user clicks a mouse button on the canvas
<code>mouse pressed</code>	triggered when the user presses a mouse button on the canvas
<code>mouse released</code>	triggered when the user releases a mouse button on the canvas
Key Input Event	
<code>text entered</code>	triggered when the user types into text field bar and presses enter

Table 3.3 JACIE-supported events

3.6 Communication Channels

3.6.1 Overview of Communication Channels

Communication channels or collaborative tools are nothing new for interacting and collaborating users across the network. Many general-purpose collaborative tools and services are available for network users to take advantage of. Microsoft NetMeeting [137], for example, has a chat channel, a audio/video channel, a whiteboard channel and a program sharing facility. These features are both practical and effective ways for any networked user-user communication.

Interactive and collaborative applications often require these facilities to effectively complement the collaborative working activities over the Internet. While it is possible to integrate a tool like NetMeeting in an interactive and collaborative application (through low-level COM programming), many interactive and collaborative application developers normally design their own channels. The programming process is undoubtedly tedious.

3.6.2 JACIE's Communication Channels

Realising that in a networked application, user-user and user-server communications may take place in a number of different forms of media, the JACIE language has a built-in support for various communication channels — *chat*, *whiteboard*, *voice*, *video*, *shared workspace* and *message*. It is just a matter of specifying which channel needs to be integrated in the application. More than one channel can be used at one time.

Chat Channel

Chat channel is used in for online text-based communications. Normally it is implemented as a window or a dialog box with a text input area and message display area. Text entered

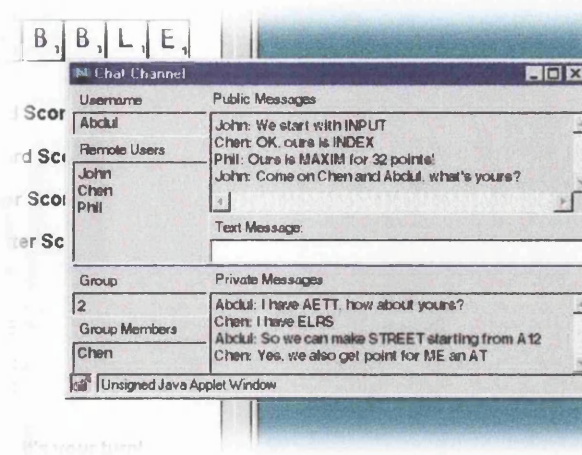


Figure 3.7 *A chat channel*

by any user will be sent to the server which then broadcasts to other users to be displayed in a message display area. The messages in the message display area typically contain some indication of their senders. In a group environment the chat channel normally has two message display areas and two text input areas — one for public messages and the other one for private or group messages. Figure 3.7 shows a chat channel with public and private messages. Through this channel general and group communications may take place in support of other collaborative activities.

Whiteboard Channel

Whiteboard channel is a dedicated shared canvas on which arbitrary drawings are placed by communicating users. Common sketching tools (e.g. pen, brush, line, rectangle, circle, colour palette, text, etc.) are normally provided. To differentiate among concurrent users, each user may also control a different pointer. The channel is normally implemented in a special window and in a group environment an additional whiteboard window is used for a private drawing canvas. Figure 3.8 shows a whiteboard channel used for scribbling medium.



Figure 3.8 A whiteboard channel

Voice Channel

Voice channel is used for online voice-based communications. Voice information from different clients are combined at the server which then broadcasts to all clients. Voice control may be adjusted locally and turn control may be exercised in a traditional “one speaks, the rest listen” manner. Figure 3.9 shows the volume adjuster for a voice channel.

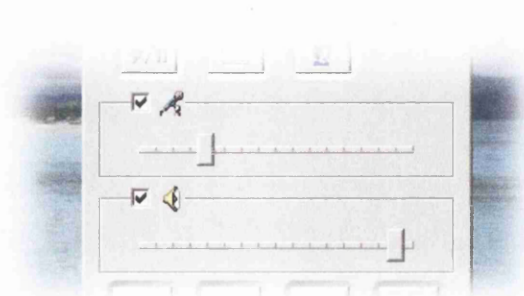


Figure 3.9 A voice channel volume adjuster

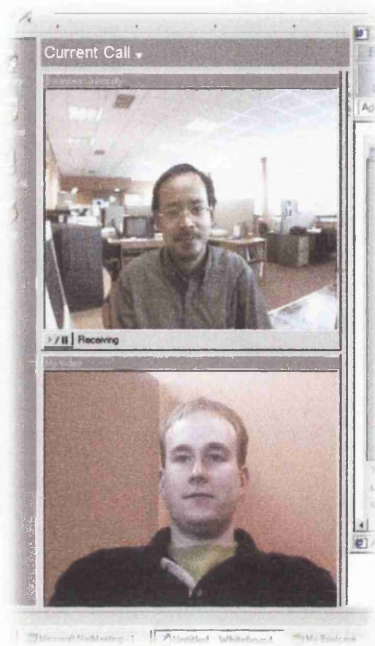


Figure 3.10 A video channel with receiving and transmitting live videos

Video Channel

Figure 3.10 shows a video channel in action. A video channel is used for online video-based communications. With a camera attached to the computers, remote users view one another as the collaborative application is running. Video quality varies depending upon the type of network connection, window size and graphics capabilities. This channel may be desirable to use but since it requires extra cost for the facility as well as high network bandwidth, it may not be applicable to all situations.

Shared Workspace

All communication channels described above are just tools for real-time computer-mediated conferencing. The real collaborative application could not be carried out just by employing them. A shared workspace is therefore required. Shared workspace or canvas channel is a workspace designed specifically to display and interact with shared objects for predefined collaborative activities under the control of predefined interactive protocols. In

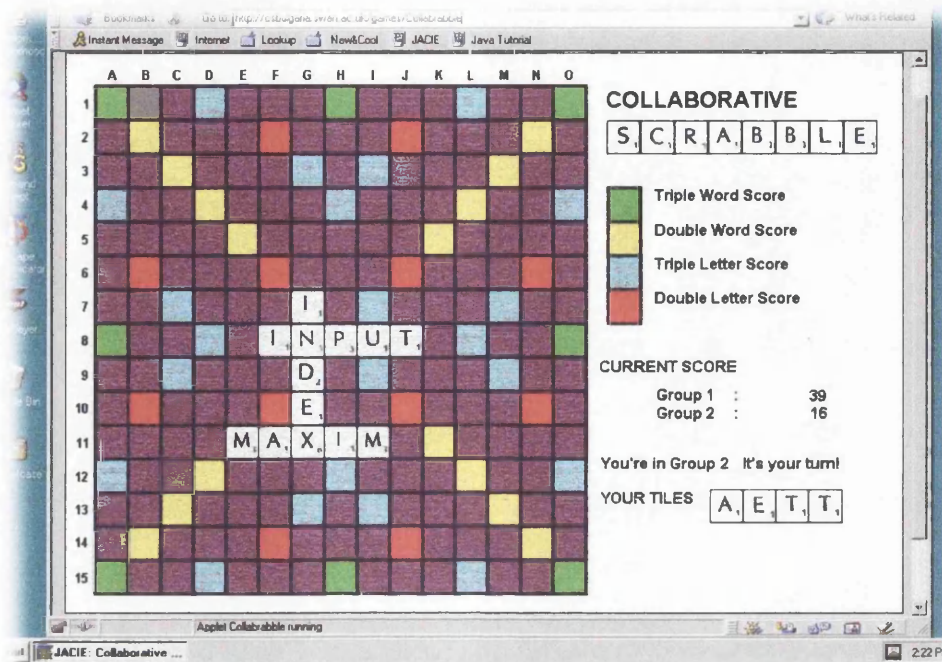


Figure 3.11 A sample application employing a shared workspace

3D virtual environments, it may be represented as a virtual room with objects and embodiments. In an image-based environment, it may be represented by a common canvas with a notion “what you see is what I see” (WYSIWIS). This workspace can be used to implement the three types of networked interactive and collaborative applications discussed earlier in this chapter.

Generally speaking, all graphical interactive and collaborative applications require some shared workspace in which collaborative activities can be accomplished. JACIE with its graphics support offers a satisfactory and straightforward solution. Figure 3.11 shows an application employing a shared workspace.

Message Channel

Message channel is mainly for message passing between a client machine and a server host. It can be used for channelling system messages that form the collaboration

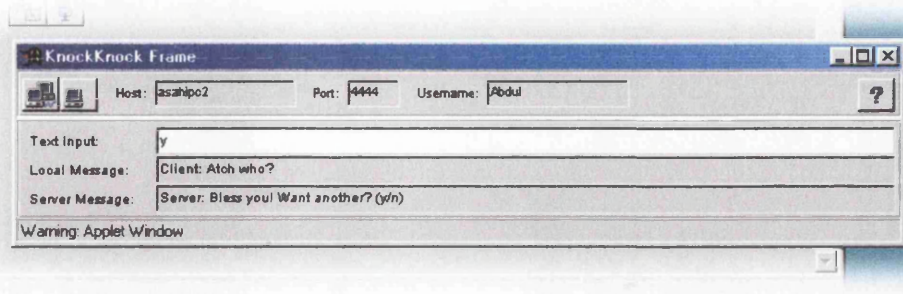


Figure 3.12 A sample application employing input/output messages

infrastructure. It can also be used for channelling user-defined messages that work in collaboration with a canvas channel or any other text input-output facility. The arrival of system messages through this channel will activate the corresponding event handler and for user-defined messages they will be queued and be processed in a manner specific to the application. Figure 3.12 shows message bars for user's text input and server's text output employed by an application.

From the above discussion, a set of built-in channels in the JACIE language provides the basic communication needs for collaborative applications. In addition to "static" channels, JACIE's workspace channel provides some facility for the dynamic collaborative applications. The channels also reduce the complexity in programming with different APIs which are required in communication and user interface design.

3.7 Support for Graphics

This research observed that images and simple drawings are likely to dominate the graphic requirements of the three categories of interactive and collaborative applications described earlier. The research concluded that the graphics in most of these applications involve mainly images and simple 2D graphic drawings, since the use of complex 3D graphic modelling in such applications would not be cost-effective in terms of expertise and effort. In fact, the very same assumption is made by most hypermedia authoring tools

except for Virtual Reality Modelling Language (VRML) [75] that was designed for 3D graphics. This assumption allows JACIE to reduce the complexity of its display functions, a desirable feature for any scripting language.

In order to facilitate effective canvas management and device-independent display, JACIE places its emphasis on a set of grid-based operations to support the display of and interaction with images and graphic primitives including lines, text and rectangles. Grids on a graphic canvas or image provide a higher-level coordinate system. Instead of addressing each pixel of the graphic displays of various resolutions, the grid system gives a more logical and manageable means to developers and end users alike.

The JACIE language may be limited in graphics but it suits the applications for which 3D graphics plays an insignificant role. In terms of programming efficiency, it manages to prevent JACIE from becoming an excessively complex and big language.

3.8 Common User Access

Common user access is important as it could provide a familiar and universal “look and feel” for all JACIE-generated applications. The design of this graphical user interface has taken into account all the requirements and the features of common image-based interactive and collaborative applications. To promote user-friendliness, simplicity is also another factor that has been taken into consideration.

Figure 3.13 shows the standard layout of the user interface components within JACIE-generated applications. On top of the layout is the menu bar. It consists of two important buttons — connect and disconnect. Visually sensible image icons have been used to replace the Java-standard text only buttons. These follow text labels and text fields for displaying or specifying the hostname where the server program resides, the port number where the networked application service takes place and the username of the client. The menu bar also consists of image icons representing buttons for each of the communication

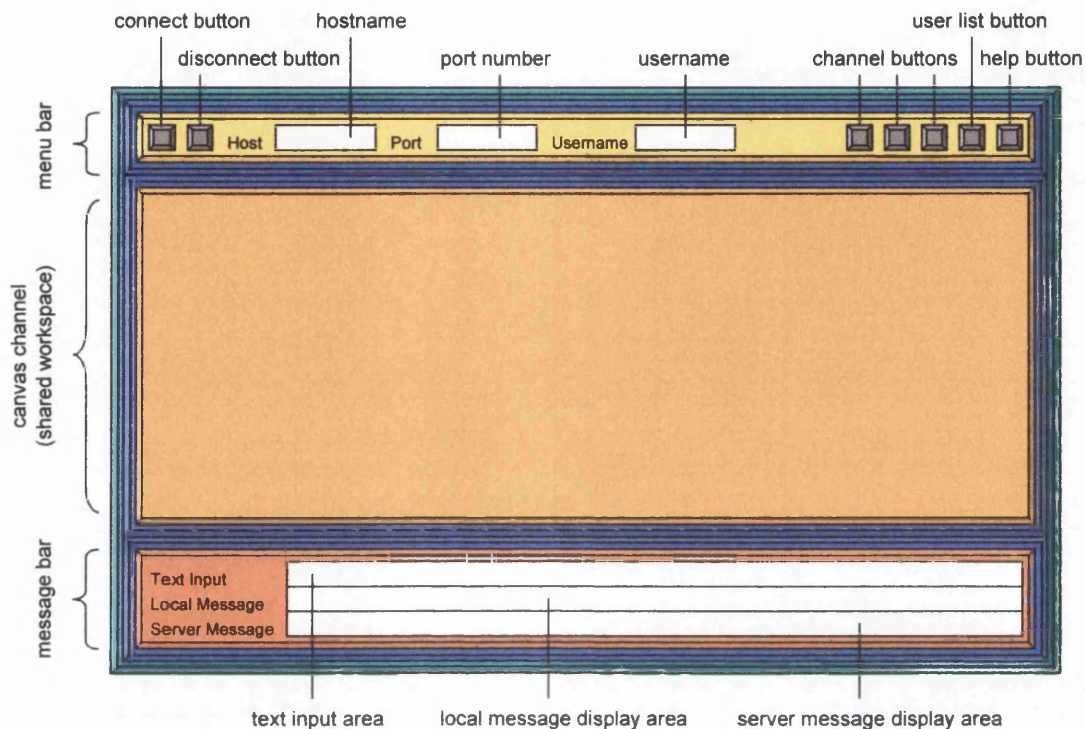


Figure 3.13 JACIE's standard user interface components

channels employed by the applications. Again the image icons have been carefully designed for better visual effect. During the active session, clicking on any of these buttons will open a dialog window for the particular channel. Two other buttons are user list button and the optional help button. The user list button is only applicable to collaborative applications of types 2 and 3. During the active session, clicking on this button will open a dialog window listing all the online users and observers (if applicable). In a group environment the dialog window will also list all the members within each group. The last button in the menu bar, namely the help button also known as the about button, is used for displaying help text. Depending on how the help text is being specified in the application program, clicking on this button can display a one-line text on a local message bar or a separate dialog window with multiple lines of help text.

In the middle of JACIE's standard layout is shared workspace or canvas channel. This is where most of the interactive and collaborative activities take place. All the JACIE's

graphic facilities are also applicable within this space. This component can be an option if the application does not require any graphic support as interaction can also occur through text only basis.

The message bar, at the bottom, accommodates text input area and two message display areas. The text input area provides the facility for users to enter text. The two message display areas are for local messages and server messages. Unlike the text input area that will be made available only if text input is required, the two message display areas are standard in all JACIE-generated programs. They are manipulated by JACIE scripts. The server message display will also display system messages throughout the active session.

Two variations of the menu bar are implemented to accommodate requirements for two interaction protocols if they are in use. As shown in Figure 3.14, if the floor management protocol in use is *tapping*, the user who has the turn should be able to pass the turn to other user (or group) of his choice. By selecting the username (or group number) from the choice list and clicking the pass turn button, a special message will be sent to the server signalling who should get the next turn. In another situation, if the floor management protocol in use is *reservation*, the user has to signal the server that he wants to reserve for his turn. This is done by clicking the reservation button. On reservation, the reservation

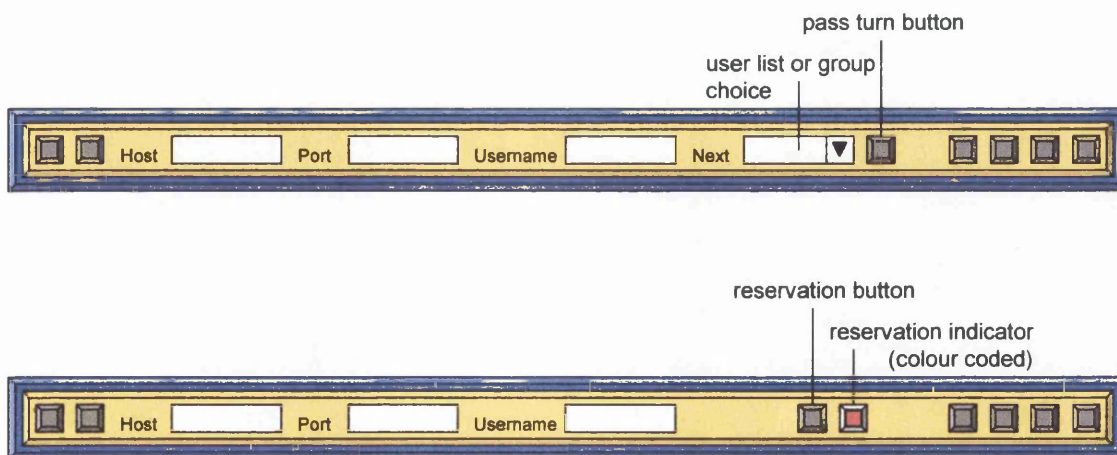


Figure 3.14 Variations of JACIE's menu bar to accommodate (a) tapping protocol or (b) reservation protocol

indicator will change from green to red. The indicator will change its colour to green again when the user gets his turn.

3.9 Summary

This chapter has discussed the design considerations of the JACIE language. Two important principles for this research, namely *special-purpose* and *programming efficiency*, have been focused. Features like scripting, single program, template-based, built-in channels, limited graphics and interaction protocols are the highlights of this research. They are valuable features for many interactive and collaborative applications. The features, handled nicely by the language and the compiler, provide programming efficiency to casual as well as hard-core programmers. The JACIE's common user access provides sufficient functions to many applications.

Chapter 4

Language Specifications

JACIE, a **J**ava-based **A**uthoring language for **C**ollaborative **I**nteractive **E**nvironments, is a scripting language designed to support rapid prototyping and implementation of net-centric, multimedia and collaborative applications. These types of applications typically incur expensive development as they involve low-level programming and require highly skilled programmers. The JACIE language is also designed for developing applets or applications in the Internet environment as the Internet has become a standard communication infrastructure for client-server human-computer interactions and multi-user human-human communication.

The JACIE language is rich in features. It is similar to other programming languages as it has some standard data types, operators, expressions and a choice of basic statements. Its strength is distinctively reflected in its high-level abstraction of various types of communication and interaction in collaborative applications, thus hiding to a great extent the complexities of network programming from application developers. This language also features choice of graphics and multimedia capabilities which makes it suitable for developing user-friendly and effective graphical applets or applications. This chapter discusses in detail the features of the JACIE scripting language.

4.1 Template-based Scripting Language

As mentioned in Chapter 2, unlike most of the Internet-based programming languages or software development tools which require separate programs for server and client, JACIE simplifies programming effort by employing a “single program” to specify both server processes and client processes. It is the compiler’s job to generate server and client programs to run on different computers. Further simplification is made through a template-based programming style which divides each program into a set of standard components. Figure 4.1 shows the three main components of a JACIE program. Each of these components is described below.

<pre>JACIE { applet name X; appletlauncher image "X.gif"; configuration {...} messages {...} client implementation { declaration {...} on canvas {...} on session start {...} on session {...} on session end {...} } server implementation { declaration {...} on server start {...} on session start {...} on session {...} on session end {...} on server end {...} } }</pre>	<p>SYSTEM CONFIGURATION COMPONENT Define program type (applet or application) and name. Define launcher (optional for applet) Specify networking parameters, channels, protocols, etc. Declare message identifiers.</p> <p>CLIENT BODY COMPONENT Declare variables and methods. Initialise canvas such as drawing a background image. Perform processes upon established connection. Main client session control. Perform processes upon session termination.</p> <p>SERVER BODY COMPONENT Declare variables and methods. Initialise server processes. Perform processes upon each user’s connection. Main server session control. Perform processes upon user’s session termination. Housekeeping processes upon all users termination.</p>
---	--

Figure 4.1 A set of standard JACIE components

4.1.1 System Configuration Component

The first component is for system configuration statements. It contains statements that specify program name and type as well as some basic networking and interaction parameters. Besides the predefined messages, the JACIE programmer may also define message identifiers in its component that will be used for message transfers between the client processes and the server processes.

4.1.2 Client Body Component

In a client/server environment, processes acting as clients and servers are normally distributed on different computers. In the second component the client body of a JACIE program specifies the code for a client process. All interactions between the client and the server are through message transfers by means of message identifiers declared earlier. The client body consists of several program constructs. The first construct, `declaration`, is for declaring all variables and methods used within the component. The `on canvas` construct follows the `declaration` construct. This construct specifies the default workspace canvas (such as background image) on which all user-defined interactions and collaborations will take place. The canvas image may be changed at a later stage during the execution by other graphics statements. The next three constructs which are `on session start`, `on session` and `on session end` define the main interaction and communication activities of a client process.

4.1.3 Server Body Component

The server body component specifies code for a server process. It consists of six constructs. The functions of the `declaration`, `on session start`, `on session`, and `on session end` are very similar to those in the client body. The two additional constructs, namely `on server start` and `on server end`, are used to interact with the operating

system for maintaining the system status of the server. The `on server start` construct initialises the server process upon invocation of the server program. The server process will stay alive, waiting for the client to establish connection from that point on. The `on server end` construct performs some house cleaning operations when all client connections are terminated and sets the server back to the waiting state for next actions. Details of all configuration statements will be discussed in Section 4.3.

4.2 Data Types, Operators and Expressions

The JACIE language is a strongly typed language, i.e., every variable and expression has a type that is known at compile time. Types limit the values a variable can hold and restrict the resulting expression. It also limits the operations supported on those values and helps detect errors at compile time.

4.2.1 Data Types

The language has five primitive data types and one compound data type. The five primitive types in are `int`, `float`, `boolean`, `image` and `string`. A character is represented by a string of length 1. It also supports arrays as its compound type. It is a means of collecting and managing the primitive types. Table 4.1 describes each one of the primitive types in JACIE.

Primitive Types	Size/Format	Description
<code>int</code>	32-bit 2's complement	Integer
<code>float</code>	32-bit IEEE 754	Single-precision floating point
<code>boolean</code>	true or false	Boolean
<code>image</code>	gif or jpeg typed images	Image
<code>string</code>	a series of characters between double quotation marks ("...")	Character string; a character is represented by a string of a single character

Table 4.1 JACIE's primitive data types

Arithmetic Operators			
+	addition	/	division
-	subtraction/negation	%	modulus
*	multiplication		
Relational and Conditional Operators			
>	greater than		or
>=	greater than or equals to	&	and
<	less than	^	exclusive or
<=	less than or equals to		logical or
==	equals to	&&	logical and
!=	not equals to	!	not

Table 4.2 JACIE's operators

4.2.2 Operators and Expression

Like any other programming languages, JACIE has various types of operators that can be used to form expressions. Basically the operators can be divided into arithmetic operators and relational and conditional operators. Table 4.2 shows all the operators supported by JACIE.

All the arithmetic operators, except %, can be applied to any arithmetic expressions. Operator % (for remainder of division) can only be applied to `int` expressions. Operator + can also be used to concatenate `string` values. JACIE neither supports tertiary conditional operators nor bitwise operators. All JACIE expressions and operator precedence follow the rules of Java.

4.3 Grammar

An extended BNF notation will be used to describe JACIE's grammar in addition to the description of each one of the JACIE statements and features. Table 4.3 shows the meta-symbols used throughout the chapter. With the same convention, meta-symbols are distinguished from keyword symbols by plain and **bold** fonts respectively.

Symbols	Meaning
< >	non-terminal
::=	"is defined as"
bold	terminal
	alternation
[]	"optional"
{ }	"zero or more occurrences"
{ }+	"one or more occurrences"

Table 4.3 Meta-symbols of BNF

4.3.1 Main Constructs

A JACIE program starts with a keyword JACIE followed by an open brace "{" and ends with closed brace "}". Using the syntax shown below, the programmer can specify whether to create an application or an applet for the client program.

```

<JACIE program> ::= JACIE {
                    <create application> | <create applet>
                }

<create application> ::= application name <identifier> ;
                    configuration {
                        <program configuration>
                    }
                    messages {
                        <message definition>
                    }
                    client implementation {
                        <client program implementation>
                    }
                    server implementation {
                        <server program implementation>
                    }

<create applet> ::= applet name <identifier> ;
                    [ <create applet option> ]
                    configuration {
                        <program configuration>
                    }
                    messages {
                        <message definition>
                    }
                    client implementation {
                        <client program implementation>
                    }
                    server implementation {
                        <server program implementation>
                    }

<create applet option> ::= appletlauncher
                        <text button launcher> | <image button launcher> ;

<text button launcher> ::= text <string>

<image button launcher> ::= image <string>

```


The first three statements of the configuration statements are the most important statements as they specify the host where the server program will reside, the port number where the communication service will take place and the username of the client.

```

<specify hostname>      ::= host
                        <string> | prompt ;

<specify port number>  ::= port
                        <integer number> | prompt ;

<specify username>    ::= username
                        <string> | prompt ;

```

The hostname can be a string representing a valid Internet address under either the domain name or the dotted decimal IP address. Integer numbers above 1024 are suggested for a port number to avoid conflicting with well-known TCP ports. Alternatively, the keyword `prompt` may be used to allow the end-user to specify the hostname, port number or his/her username during execution. The standard user interface for connect/disconnect and text fields for specifying hostname, port number and username is shown in Figure 4.3.

JACIE supports five communication channels through which remote users can interact while engaging in some collaborative activities. The channels are `canvas`, `chat`, `whiteboard`, `voice` and `video`. One or more channels can be specified as follows:

```

<specify channel>      ::= channel <channel name> { , <channel name> } ;

<channel name>        ::= canvas | chat | whiteboard | voice | video

```

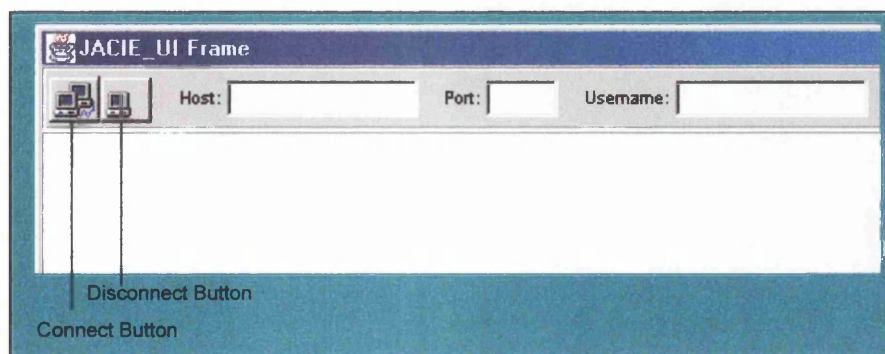


Figure 4.3 Standard user interface for connection/disconnection in JACIE-generated programs

By specifying any of the channel names, other than `canvas`, a button for the communication channel will be included in the program's user interface. During an active session, a dialog window for any particular channel will be opened when the user clicks on the channel button. By default, `canvas` channel will also be included if any graphics statement is specified in the program code. Figure 4.4 shows the channel buttons and canvas workspace.

The JACIE language also allows programmers to specify a one-line text or a filename of a text file for the description or the help file of the application being developed. This facility is made available through the "about" button of the JACIE interface. The effect of clicking on this button, if it is a one-line text, is that the text string will be displayed on the local message bar. Alternatively, if it is a file, a dialog box will display the whole text. The statement is an `about` statement and is written as follows:

```
<specify about> ::= about
                  <string> | <about file> ;
<about file> ::= file <string>
```

Another useful configuration statement is the `number of users` statement. This statement specifies the number of concurrent users allowed and is used either because of the requirement of the application being developed or a restriction is imposed in order to achieve acceptable server performance. The programmer may specify a fixed number or a range of numbers as follows:

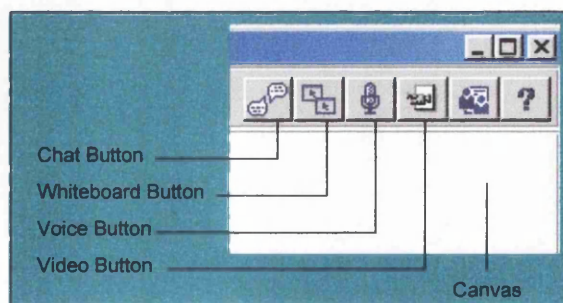


Figure 4.4 *Standard buttons for communication channels in JACIE-generated programs*

If `userdefined` protocol is chosen, the first remote users assigned to each group will be the leaders and these leaders are required to choose the members (one at a time) before the actual session can take place. If the group protocol is `random`, the server will assign the groups in a random order and if the group protocol is `alternate`, users are grouped according to the order of connection time. The default protocol of group is `alternate`.

Other than the `host`, the `port` and the `username` statements, not all of the above configuration statements are needed in all applications. It depends on the type of application to be developed. For applications of type single-user server-based interactions, no other statements are necessary. However, the `number of users` statement can be used to limit the concurrent users and prevent performance degradation. Applications of type "multi-user server-mediated interactions" require `number of users` and `protocol` statements. If the remote users require some means of communication, the `channel` statement can be used. The `number of groups` and `protocol of group` statements are only required in group collaborative applications. The `number of observers` statement is only applied to multi-user and group collaborative applications. The `about` statement is an option for all types of applications.

Following configuration is the `messages` construct. This construct contains an optional list of identifiers that represent the message definition. This message definition is used in communication statements for interaction between the client and the server.

```
<message definition> ::= [ <identifier> {, <identifier> } ]
```

```
JACIE {
  ...
  configuration {
    host "csbean.swan.ac.uk";
    port 3333;
    username prompt;
    about "2 player puzzle";
    channel chat, canvas;
    number of users 2;
    protocol contention;
  }
  messages {
    puzzleBlock, remoteX, remoteY;
  }
  ...
}
```

Figure 4.5 A sample code showing JACIE's configuration statements

Two of the communication statements that rely solely on this message definition are `send` and `receive`, and will be discussed later in this chapter. Figure 4.5 illustrates some related statements applied under the `configuration` and `messages` constructs of one JACIE program.

4.3.3 Client Implementation and Server Implementation Constructs

The `client implementation` and `server implementation` of JACIE codes are defined by the following syntax:

```

<client program implementation> ::= declaration
                                < variable and method declaration list>
                                on canvas
                                <compound statement>
                                on session start
                                <compound statement>
                                on session
                                <compound statement>
                                on session end
                                <compound statement>

<server program implementation> ::= declaration
                                < variable and method declaration list>
                                on server start
                                <compound statement>
                                on session start
                                <compound statement>
                                on session
                                <compound statement>
                                on session end
                                <compound statement>
                                on server end
                                <compound statement>

<variable and method declaration list> ::=
    { <variable declaration list>
      <method declaration list> }

```

The `declaration` construct is where all the variables and methods within the respective implementations are to be declared. The `on canvas` construct contains mainly graphics statements that make up the default shared workspace. This is where user-defined interactions and output displays will take place.

<code>on server start</code>	triggered when the server process is started; untriggered when server process ready to listen to incoming connection
<code>on session start</code>	triggered when the user clicks connect button and a connection is successfully established; untriggered when start session flag being signalled
<code>on session</code>	triggered when start session flag being signalled; untriggered when the statements in <code>on session</code> construct terminates naturally or the user clicks disconnect button or physical connection interrupted
<code>on session end</code>	triggered when the statements in <code>on session</code> construct terminates naturally or the user clicks disconnect button or physical connection interrupted
<code>on server end</code>	triggered when all the active clients leave the session

Table 4.4 *JACIE's main events*

As mentioned earlier, the `on session start`, the `on session` and the `on session end` are the three main events in JACIE applications. They consist of statements that need to be executed at different times and states of the active session. Some statements are also applied to only one of the states. The `on server start` and the `on server end` contain statements for housekeeping operations of the server. The former is for initialising the server process before the first remote user establishes its connection and the latter is for reinitialising the server process after all client connections are terminated.

Table 4.4 shows how the events are being triggered and untriggered. The choices of statement within main events are critical to make the generated codes executable.

4.3.4 Variable Declaration Statements

Being a strongly-typed language, JACIE requires all variables to be declared before they can be referred to. The scope of the variables is within the implementation (`client implementation` or `server implementation`). Local variable declarations are allowed but are limited within method declaration and in `for` statement. In general, variable declarations should follow the syntax:

```

<variable declaration list> ::= { [shared] <data types> <variable declarator> ; }
<data types>                ::= <primitive type> | <compound type>
<primitive type>            ::= int | float | boolean | image | string
<compound type>             ::= <primitive type> [ <expression> ]
                               | <compound type> [ <expression> ]
<variable declarator>       ::= <identifier> [ = <variable initialiser> ]
<variable initialiser>      ::= <expression> | <array initialiser>
<array initialiser>         ::= { <variable initialiser list> }
<variable initialiser list> ::= { <variable initialiser>, } <variable initialiser>

```

JACIE's variable declaration format resembles Java's. The difference between JACIE and Java lies in the modifier keyword. Modifier shared in JACIE is used in the context of server implementation where the single-copy shared variables are referred to by all instances of remote users. On the contrary, the "unshared" variables are local to each client connection.

```

JACIE {
  ...
  client implementation {
    declaration {
      int gridx;
      int gridy;
      int xfree = 5;
      int yfree = 5;
      ...
      image blank = "blank.jpg";
      image[2][6][6] p =
      {
        {"lim00.jpg", "lim10.jpg", "lim20.jpg", "lim30.jpg", "lim40.jpg", "lim50.jpg"},
        {"lim01.jpg", "lim11.jpg", "lim21.jpg", "lim31.jpg", "lim41.jpg", "lim51.jpg"},
        {"lim02.jpg", "lim12.jpg", "lim22.jpg", "lim32.jpg", "lim42.jpg", "lim52.jpg"},
        {"lim03.jpg", "lim13.jpg", "lim23.jpg", "lim33.jpg", "lim43.jpg", "lim53.jpg"},
        {"lim04.jpg", "lim14.jpg", "lim24.jpg", "lim34.jpg", "lim44.jpg", "lim54.jpg"},
        {"lim05.jpg", "lim15.jpg", "lim25.jpg", "lim35.jpg", "lim45.jpg", "blank.jpg"}
      };
      {
        {"rim00.jpg", "rim10.jpg", "rim20.jpg", "rim30.jpg", "rim40.jpg", "rim50.jpg"},
        {"rim01.jpg", "rim11.jpg", "rim21.jpg", "rim31.jpg", "rim41.jpg", "rim51.jpg"},
        {"rim02.jpg", "rim12.jpg", "rim22.jpg", "rim32.jpg", "rim42.jpg", "rim52.jpg"},
        {"rim03.jpg", "rim13.jpg", "rim23.jpg", "rim33.jpg", "rim43.jpg", "rim53.jpg"},
        {"rim04.jpg", "rim14.jpg", "rim24.jpg", "rim34.jpg", "rim44.jpg", "rim54.jpg"},
        {"rim05.jpg", "rim15.jpg", "rim25.jpg", "rim35.jpg", "rim45.jpg", "blank.jpg"}
      };
    };
  }
  ...
}

```

Figure 4.6 A sample code showing JACIE's variable declaration statements

<relational expression>	<pre> ::= <additive expression> <relational expression> < <relational expression> <relational expression> > <relational expression> <relational expression> <= <relational expression> <relational expression> >= <relational expression> </pre>
<additive expression>	<pre> ::= <multiplicative expression> <additive expression> + <additive expression> <additive expression> - <additive expression> </pre>
<multiplicative expression>	<pre> ::= <unary expression> <multiplicative expression> * <multiplicative expression> <multiplicative expression> / <multiplicative expression> <multiplicative expression> % <multiplicative expression> </pre>
<unary expression>	<pre> ::= + <unary expression> - <unary expression> <unary expression not plus minus> </pre>
<unary expression not plus minus>	<pre> ::= <postfix expression> ! <unary expression> </pre>
<postfix expression>	<pre> ::= <primary> <name> rnd (<expression>) <system variable> </pre>
<primary>	<pre> ::= <literal> (<expression>) <method invocation> <array access> </pre>
<literal>	<pre> ::= <int literal> <float literal> <boolean literal> <string literal> <>null literal> </pre>
<method invocation>	<pre> ::= <name> ({ <argument list> }) </pre>
<argument list>	<pre> ::= { <expression> , } <expression> </pre>
<array access>	<pre> ::= <name> { [<expression>] }+ </pre>
<name>	<pre> ::= <identifier> java_<identifier> </pre>

4.3.5 Method Declarations

The declaration construct also allows methods to be defined. Like any other programming language a method declaration either specifies the type of value that the method returns or keyword `void` which has to be used to indicate that the method does not

return a value. JACIE also allows method overloading as long as the overloaded method has different signature. In general, method declaration in JACIE should follow this syntax:

```

<method declaration list> ::= { [shared] <method header> <method body> }

<method header>           ::= <data type><identifier> ( [<formal parameter list>] )
                           | void <identifier> ( [<formal parameter list>] )

<formal parameter list>  ::= { <formal parameter> , } <formal parameter>

<formal parameter>       ::= <data type> <identifier>

<method body>            ::= <compound statement>

```

An example of method declarations in JACIE is shown in Figure 4.7.

4.3.6 Basics Statements

JACIE adopts comments format similar to C, C++ or Java. Basically it can either be a traditional multi-line comments that start with `/*` and end with `*/` or an end-of-line comment that start with `//` and end with a line terminator. JACIE also adopts Java-style documentation comment. A separate tool can be developed for program documentations. All comments will be ignored by the compiler and will not appear in the generated Java codes.

```

JACIE {
  ...
  client implementation {
    declaration {
      ...
      void moveDown(int block, int xclick, int yclick, int yfree) {
        for (int y = yfree; y > yclick; y=y-1)
          p[block][xclick][y] = p[block][xclick][y-1];
        p[block][xclick][yclick] = blank;
      }
      void moveRight(int block, int xclick, int yclick, int xfree) {
        for (int x = xfree; x > xclick; x=x-1)
          p[block][x][yclick] = p[block][x-1][yclick];
        p[block][xclick][yclick] = blank;
      }
      ...
    }
    ...
  }
  ...
}

```

Figure 4.7 A sample code showing JACIE's method declaration

JACIE features all the basic constructs of procedural programming languages. These includes an assignment statement, a method invocation and control flow statements. Most of the syntax closely follows the Java style with some exceptions for simplification purposes.

The grammar syntax below shows some basic statements in JACIE. The details of each statement will be described in different sections of this chapter.

<comment>	::= {<traditional comment> {<end of line comment>} {<documentation comment>}
<traditional comment>	::= /* {<comment content>} */
<end of line comment>	::= // {<comment content>} <line terminator>
<documentation comment>	::= /** {<comment content>} */
<compound statement>	::= { [<statement list>] }
<statement list>	::= { <statement> }+
<statement>	::= <expression statement> <if then statement> <if then else statement> <for statement> <while statement> <return statement> <exit statement> <compound statement> <text input statement> <print text statement> <clear message bar statement> <foreground colour statement> <refresh screen statement> <clean canvas statement> <move to statement> <draw grid statement> <paint grid statement> <draw line statement> <draw image statement> <draw string statement> <canvas definition statement> <specify canvas statement> <event control statement> <pause statement> <send statement> <pass turn statement> <abort session statement>

In JACIE, simple assignments to a variable or an array are allowed but compound assignments (e.g. +=, -=, *=, /=, etc) and auto-increment/decrement (++ , --) are not. All binary operators except assignment operator are left associative. Control flow statements

for conditional execution of a statement and looping statements for iterations operate the same way as in other programming languages. JACIE does not support switch-like statement that transfers control to one of several statements.

Syntaxes for assignment and control flow statements are shown below. A sample code fragment involving flow control statements is shown in Figure 4.8.

```

<expression statement> ::= <statement expression> ;
<statement expression> ::= <assignment> | <method invocation>
<assignment> ::= <assignee> = <assignment expression>
<assignee> ::= <name> | <array access>
<if then statement> ::= if ( <expression> ) <statement>
<if then else statement> ::= if ( <expression> ) <statement> else <statement>
<for statement> ::= for ( [<for init>] ; [<expression>] ; [<for update>])
                        <statement>
<for init> ::= <statement expression list>
                | <local variable declaration>
<local variable declaration> ::= <data type> <variable declarator>
<for update> ::= <statement expression list>
<statement expression list> ::= { <state expression> , } <statement expression>
<while statement> ::= while ( <expression> ) <statement>
<return statement> ::= return [<expression>] ;
<exit statement> ::= exit ;

```

4.3.7 Input-Output Statements

The JACIE interface includes a one line text field bar for text input and two message bars, namely `localmessage` bar and `servermessage` bar, for text output. The two message bars are standard features in JACIE-generated programs but the text field bar will only be created when the text input statement is in use. By default, a print statement will print on `localmessage` bar. The `servermessage` bar will also display system messages throughout the active session. The syntax for text input-output statements is shown below.

```

...
on session {
  on MOUSECLICK {
    if (localBlock == 0 && GETGRID == puzzleGrid1) {
      ...
      if (gridx == xfree) {
        ...
        if (gridy > yfree)
          moveUp(localBlock,gridx,gridy,yfree);
        if (gridy < yfree)
          moveDown(localBlock,gridx,gridy,yfree);
        ...
      }
      else if (gridy == yfree) {
        ...
        if (gridx < xfree)
          moveRight(localBlock,gridx,gridy,xfree);
        if (gridx > xfree)
          moveLeft(localBlock,gridx,gridy,xfree);
        ...
      }
    }
  }
  else {
    ...
  }
}
...

```

Figure 4.8 *A sample JACIE code with flow control statements*

```

<text input statement> ::= input <receiver list> ;
<receiver list> ::= { <assignee> , } <assignee>
<print text statement> ::= print [<message bar>] [<expression list>] ;
<clear message bar statement> ::= clear <message bar> ;
<message bar> ::= servermessage | localmessage

```

Figure 4.9 shows a complete JACIE program that makes full use of the input-output message bars. The program named `Echo2`, on connection establishment, allows a user to type in a string of characters on text input bar and send it to the server. On receiving the string, the server will then echo back to the user which will then display it on the server message bar. The interaction may continue until the user clicks the disconnect button. Figure 4.10 shows the `Echo2` program in action.

```

/**
 * Programme   : Echo2.jacie
 * Description  : This program creates an application which allows user to send a text
 *               string to the server. The server then echoes the string back to
 *               the client.
 */

JACIE {
  application name Echo2;
  configuration {
    host prompt;
    port 2222;
    username prompt;
    about "Echo application by Abdul Samad Haji Ismail"; }
  messages {
    EchoMe, ServerEcho }
  client implementation {
    declaration {
      string TypeMessage;
      string EchoedMessage; }
    on canvas { }
    on session start { }
    on session {
      print "Please enter your text";
      input TypeMessage;
      send EchoMe TypeMessage;
      receive ServerEcho EchoedMessage;
      print servermessage EchoedMessage; }
    on session end {
      clear localmessage;
      clear servermessage; } }
  server implementation {
    declaration { string ClientMessage; }
    on server start { }
    on session start { }
    on session {
      receive EchoMe ClientMessage;
      send ServerEcho ClientMessage; }
    on session end { }
    on server end { } }
}

```

Figure 4.9 A sample JACIE program utilising input-output message bars

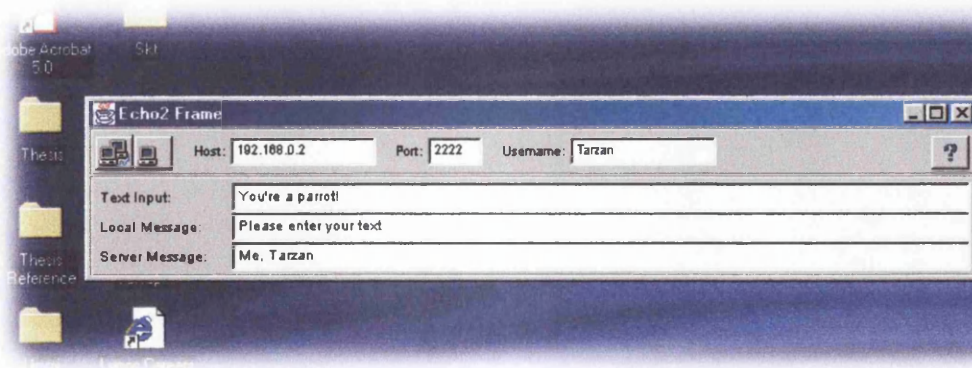


Figure 4.10 The running Echo2 client program

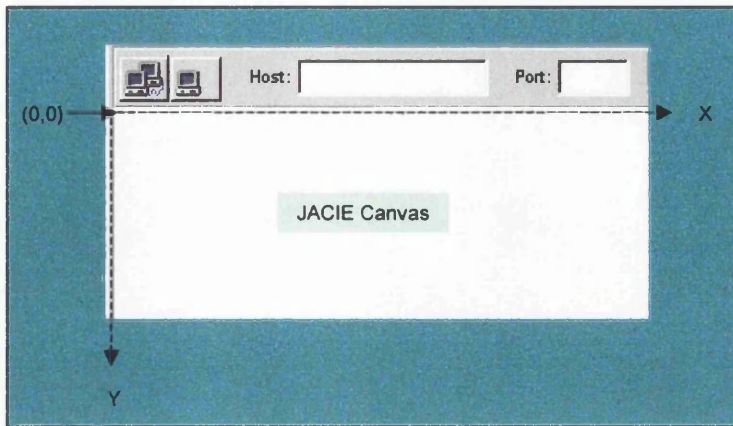


Figure 4.11 *Standard canvas area for JACIE-generated programs with its coordinate system*

4.3.8 Graphics Statements

All graphics statements are executed on canvas channel. JACIE's default canvas resolution is 632 by 360. The default size of the resolution of the canvas is in consideration to the default resolution for all the JACIE-generated client programs which is 640 by 480. The default resolution can be redefined, if necessary, by using `canvas size` statement. The syntax is as follows:

```
<canvas size statement> ::= canvas size <pair expression> ;
```

Graphic objects can only be drawn within this area. The coordinate system for the canvas starts from (0,0) at the upper left corner with X coordinate increasing to the right and Y coordinate increasing downward. Figure 4.11 shows the canvas area of JACIE-generated programs with its coordinate system.

JACIE has a sufficient number of graphics statements that can be used for primitive graphics and image manipulations. It facilitates effective canvas management and device-independent display by placing its emphasis on a set of grid-based operations. Grids on a

canvas provide a higher level coordinate system. Instead of addressing each pixel of the graphic displays at various resolutions, the grid system gives more logical and manageable means to developers and end users alike.

The first two graphics statements are `foreground` and `background`. These statements set the foreground colour and the background colour of the canvas. There are 13 predefined colours in JACIE. If any of the primitive graphics statements do not specify explicitly the graphic colour, the foreground colour will be used. By default, the foreground colour is black and the background colour is white. The syntax is shown below:

```

<foreground colour statement> ::= foreground <colour name> ;
<background colour statement> ::= background <colour name> ;
<colour name>                ::= black | blue | green | cyan | red | magenta | yellow |
                               white | gray | darkgray | lightgray | orange | pink
<refresh screen statement>    ::= refresh ;
<clean canvas statement>      ::= clean ;

```

A JACIE-generated program refreshes the canvas after each main event. But there are times when a screen refresh is needed at once. The `refresh` statement is to be used to refresh the canvas at any point of execution. The `clean` statement, on the other hand, is used to restore the canvas to the original initial state as defined in `on canvas` construct. This statement is used when more than one canvas is in use (as defined by `define canvas`) and the program logic requires the program to use the original canvas or at any other time the canvas needs to be restored to the initial graphic state.

As mentioned earlier, JACIE's emphasis is on grid-based graphic operations on top of absolute positioning operations. The statement that defines the grid location and parameters is `draw grid`. This statement specifies the name of the grid, the upper left position of the grid, the number of rows and columns, the cell size, and the grid lines' width and colour. The defined grids are then referred to by name and the cells coordinate system. Just like the canvas coordinate system, the upper left cell is referred to as (0,0) with X coordinate increases to the right and Y coordinate increases downward.

Syntactically, it is allowed to define a grid that overlaps with other grids. Extra precaution has to be made when defining overlapping grids as only the last operation on the overlapped cells is visible and mouse events on overlapped cells may result in unexpected outcome. Another graphics statement directly related to grid is `paint grid`. It will paint the specified cell with the given `colour` name. The syntax is shown below:

<code><draw grid statement></code>	<code>::= draw <grid name> [<draw at>] [<draw step>] [<draw size>] [<draw colour>] [<draw width>] ;</code>
<code><grid name></code>	<code>::= grid <identifier></code>
<code><draw at></code>	<code>::= at <pair expression></code>
<code><draw step></code>	<code>::= step <pair expression></code>
<code><draw size></code>	<code>::= size <pair expression></code>
<code><draw colour></code>	<code>::= colour <colour name></code>
<code><draw width></code>	<code>::= width <expression></code>
<code><paint grid statement></code>	<code>::= paint <grid name> [<draw at>] [<draw colour>] ;</code>

The canvas and the defined grids maintain the state of the drawing context in terms of foreground colour and current active coordinate. For the current active point or cell (for grid-based), by default, it starts from (0,0). This active point or cell can be set to other coordinate by using `move to` statement as below:

<code><move to statement></code>	<code>::= move to [<grid name>] <pair expression> ;</code>
<code><pair expression></code>	<code>::= <expression> , <expression></code>

The current active point or cell is the default value for all primitive graphics statements, i.e. `draw from` for `draw line` statement, or `draw at` for `draw image` and `draw string` statements. Alternatively these graphics statements may specify explicitly the coordinate points required.

JACIE provides three statements for specifying graphic primitives. They are `draw line`, `draw image` and `draw string`. The syntax is as follows:

<code><draw line statement></code>	<code>::= draw line [<grid name>] [<draw from>] to <pair expression> [<draw colour>] [<draw width>] ;</code>
--	--

<draw image statement>	::= draw image [<grid name>] <expression list> [<draw at>] [<draw size>] [<draw flip>] ;
<draw string statement>	::= draw string [<grid name>] <expression list> [<draw at>] [<draw font>] [] [] ;
<draw from>	::= from <pair expression>
<draw flip>	::= flip <flip choice>
<flip choice>	::= horizontally vertically diagonally
<draw font>	::= font
	::= arial courier times
	::= <expression>
	::= plain bold italic bolditalic

A line is drawn with the `draw line` statement by either specifying only the terminal point (in this case the current active coordinate or cell will be used as the starting point) or the starting point and the terminal point. If a line is to be drawn on the named grid, the starting point and the terminal point will be in the middle of the cells. The line colour and the thickness can also be specified.

The `draw image` statement allows an image to be drawn either directly on the canvas or relatively on the cell grid. The image can be scaled to a preferred size and can be flipped horizontally, vertically or diagonally.

The last primitive graphics statement in JACIE is `draw string`. This statement will draw the specified string either on the canvas or on the grid cell. If the string is drawn on canvas, the specified coordinate is the position of the lower left corner of the text. If it is drawn on the grid, the text will be on the specified cell two pixels away from the grid lines (horizontal and vertical). The font size, type and style can also be specified. The three standard font types are `arial`, `courier` and `times`.

In addition to the default canvas specified in `on canvas` construct, JACIE also allows multiple canvases to be defined. These canvases, specified by `define canvas` statement, may contain primitive graphics statements as well as other statements which can be called and displayed, with `use canvas` statement, at any time during execution. This facility

allows graphic primitives to be drawn behind the scenes. It also allows different canvases to be displayed, be hidden and redisplayed, useful for the purpose of working on different canvases or to create animation effects. The syntax for these two statements are shown below:

```
<canvas definition statement> ::= define canvas <identifier> <compound statement> ;
<specify canvas statement>    ::= use canvas <identifier> ;
```

Like all other JACIE statements, the graphics statements of JACIE will be translated to Java statements. In Java, Abstract Window Toolkit (AWT) drawing system controls when and how a program can draw. It also requires rapid execution of paint and update methods of `java.awt.Graphics` [60]. As this is not the problem of JACIE, when coding

```
client implementation {
  declaration {
    ...
  }
  on canvas {
    draw image bground at 40,0;
    draw grid puzzleGrid1 at 54,60 step 6,6 size 40,40 colour lightgray width 1;
    draw grid puzzleGrid2 at 334,60 step 6,6 size 40,40 colour lightgray width 1;
  }
  on session start {
    ...
    print "Displaying the puzzle pieces... please wait";
    for (int i=0;i<6;i=i+1)
      for (int j=0;j<6;j=j+1)
        draw image grid puzzleGrid1 p[0][i][j] at j,i;
    for (int i=0;i<6;i=i+1)
      for (int j=0;j<6;j=j+1)
        draw image grid puzzleGrid2 p[1][i][j] at j,i;
    ...
  }
  on session {
    on MOUSECLICK {
      if (localBlock == 0 && GETGRID == puzzleGrid1) {
        gridx = GETGRIDX; gridy = GETGRIDY;
        if (gridx == xfree) {
          ...
          if (gridy > yfree)
            moveUp(localBlock,gridx,gridy,yfree);
          if (gridy < yfree)
            moveDown(localBlock,gridx,gridy,yfree);
          ...
        }
        ...
      }
    }
    for (int i=0;i<6;i=i+1)
      for (int j=0;j<6;j=j+1)
        draw image grid puzzleGrid1 p[0][i][j] at i,j;
    for (int i=0;i<6;i=i+1)
      for (int j=0;j<6;j=j+1)
        draw image grid puzzleGrid2 p[1][i][j] at i,j;
    ...
  }
  ...
}
```

Figure 4.12 A sample JACIE code with some graphics statements

in JACIE, extra care has to be made when mixing the graphics statements with the event control statements (to be described below). Otherwise it may destroy the perceived performance of the program.

A sample JACIE code employing graphics statements is shown in Figure 4.12. Notice the different circumstances where the statements are being used. GETGRID, GETGRIDX and GETGRIDY are predefined system variables (refer the following section).

4.3.9 Event Control Statements

JACIE provides the means to handle different kinds of events generated by some predetermined runtime behaviour. In addition to the main events described in Table 4.4, other events can be categorised into *session events*, *mouse events* and *keyboard events*.

Table 4.5 lists all the events and describes how the events are triggered. The statement that handles these events accordingly is `on <event>` statement, where `<event>` is one of the thirteen recognised events.

Session Events	
on WAITING	triggered when the required number of online users has not been reached
on OBSERVERCONNECTION	triggered when a remote user is assigned an observer status; normally it will be used by server process to send the current states of the session
on TURN	triggered by the server when the current turn is the user's turn
on GROUPTURN	triggered by the server when the current group turn is the user's group turn
on REQUESTCONTROL	triggered by the user to flag the server to request for turn (in reservation floor control management)
on RESERVATION	triggered when the user has made request control but yet to get his turn (in reservation floor control management)
on CLIENTABORT	triggered when the client abort the program in the middle of the execution
on SERVERABORT	triggered when the server abort the program in the middle of the execution
on NEWMESSAGE	triggered when message queue is not empty
Mouse Events	
on MOUSECLICK	triggered when the user clicks a mouse button on the canvas
on MOUSEPRESS	triggered when the user presses a mouse button on the canvas
on MOUSERELEASE	triggered when the user releases a mouse button on the canvas
Key Input Event	
on TEXTENTERED	triggered when the user types into text field bar and presses enter

Table 4.5 JACIE-handled events

```

<event control statement> ::= on <event> <statement>

<event>
    ::= WAITING | OBSERVERCONNECTION | TURN | GROUPTURN |
       REQUESTCONTROL | RESERVATION | SERVERABORT |
       CLIENTABORT | NEWMESSAGE | MOUSECLICK | MOUSEPRESS |
       MOUSERELEASE | TEXTENTERED

<pause statement> ::= wait <expression> ;

```

JACIE-handled events are closely related to some JACIE system variables. These system variables can be referred to in place of expressions (refer <expression> syntax). The predefined system variables are shown below:

```

<system variable> ::= USERNAME | USERNUMBER | GROUPNUMBER |
                   CURRENTTURN | CURRENTGROUPTURN | MESSAGEID |
                   GETX | GETY | GETGRID | GETGRIDX | GETGRIDY |
                   GETTEXT

```

As their names imply, USERNAME returns the username as specified by the remote user itself and USERNUMBER returns the usernumber assigned by the server upon establishing connection. GROUPNUMBER returns the user's group number assigned by the server in a group environment when the required number of online users has been met. CURRENTTURN returns the current turn assigned by the server in non-contention server-mediated interaction environment. Similar to CURRENTTURN, the GROUPTURN returns the current group turn. In contention-style floor management, CURRENTTURN and GROUPTURN are not applicable.

Other system variables are to be used in conjunction with event statements. Initially these system variables have null values. When the expected events are triggered, the respective system variables will return a value which can be referred to accordingly. On mouse events for example, GETX and GETY will return the absolute coordinate of the canvas at which the event occurred. If it has occurred on the defined grid, the grid name referred to as GETGRID and the cell coordinates referred to as GETGRIDX and GETGRIDY will return the required values. Similarly, GETTEXT will return a string after the user presses enter key at the text field bar. On the other hand, MESSAGEID can be used in conjunction with on NEWMESSAGE statement or independently before receive statement to ensure that the right

```

on session start {
  on WAITING {
    // statements executed while waiting for more users
    ...
  }
  // statements executed after number of expected users met
  ...
}
on session {
  on MOUSECLICK {
    if (localBlock == 0 && GETGRID == puzzleGrid1) {
      gridx = GETGRIDX; gridy = GETGRIDY;
    }
    ...
  }
  on NEWMESSAGE {
    ...
  }
  ...
}

```

Figure 4.13 *A sample JACIE code showing some event control statements*

message is read from message queue. A sample code fragment involving event control statements, together with the related system variables, is shown in Figure 4.13.

4.3.10 Communication Statements

The final category of JACIE statements is communication statements. These statements specify interaction primitives for communicating client processes and server processes. The two important statements are `send` and `receive`. Since every sent message needs to be received, and vice-versa, they must be used in pair. The interaction should occur within the same session construct of `client implementation` and `server implementation`. Failure to ensure this pair operates in synchronisation will lead to a runtime error.

Each pair of `send` and `receive` statements should specify the message identifier defined earlier in `messages` construct. This is followed by the data to be sent or received. JACIE allows any type of data to be sent. Implementation-wise, this data will be converted and wrapped before it can be sent, and it will be unwrapped and reconverted before it can be received. A list of data separated by commas can be sent through one `send` statement and

the received statement has to receive the list accordingly. The syntax for communication statements is shown below.

<code><send statement></code>	<code>::= send <identifier> [<expression list> [to <send choice>]] ;</code>
<code><send choice></code>	<code>::= server all others group</code>
<code><receive statement></code>	<code>::= receive <identifier> [<receiver list>] ;</code>
<code><pass turn statement></code>	<code>::= pass turn [<expression>] ;</code>
<code><abort session statement></code>	<code>::= abort ;</code>

The send statement also allows sending of data either in point-to-point or one-to-many mode. By default a send statement specified at the client is directed to the server, and a send statement specified at a server is directed to the client. But if there are times when sending of data is meant for all the online users (including the client itself), the `to all` phrase can be used. On the other hand, if sending of data is meant for all the online users except for the client, the `to others` phrase is to be used. Sending of data can also be made to members of the group with the `to group` phrase.

Another important communication statement which is used in multi-user and group interactions is `pass turn`. The statement `pass turn`, without parameter, will inform the server that the client has used his turn and it is up to the server to determine the next turn. The client can also pass his turn to a specific user by specifying the other party's user number. In inter-group interaction, if one member of the group passes the turn his action will represent the group decision. The subsequent `pass turn` statement by the group members will be ignored by the server.

The final communication statement in JACIE is `abort`. This statement will abandon the current session state and move to the next state. If it occurs at client logic, the server will be notified. Similarly, if it occurs at server logic, the client will be notified.

A sample JACIE code fragment with communication statements highlighted is shown in Figure 4.14. Notice how the `send` and `receive` statements are handled in pairs.


```

messages {
  puzzleBlock, remoteX, remoteY
}
client implementation {
  declaration { ... }
  on canvas { ... }
  on session start {
    ...
    receive puzzleBlock localBlock;
    ...
  }
  on session {
    on MOUSECLICK {
      ...
      send remoteX gridx; send remoteY gridy;
      ...
    }
    on NEWMESSAGE {
      receive remoteX rgridx; receive remoteY rgridy;
      ...
    }
  }
  on session end { ... }
}
server implementation {
  declaration { ... }
  on server start { ... }
  on session start {
    block = USERNUMBER - 1;
    send puzzleBlock block;
    ...
  }
  on session {
    on NEWMESSAGE {
      receive remoteX x; receive remoteY y;
      send remoteX x to others; send remoteY y to others;
    }
  }
  on session end { ... }
  on server end { ... }
}

```

Figure 4.14 A sample JACIE code showing communication statements

4.4 Interfacing to Java Language

Considering that some applications may require functions which JACIE cannot supply, JACIE compensates this by providing a means to interface with the Java language. This facility enables experienced programmers to utilise Java for the implementation of complex code segments, for example, complicated graphics, numerical computation, or logic control. All reference to a Java class should have `Java_*` prefix. Java codes can also be embedded in the JACIE script by a special tag as follows and will be copied directly to the client or the server program.

```

...
int x[5] = {100, 200, 250, 50, 100};           // JACIE code
int y[5] = {50, 50, 200, 200, 50};           // JACIE declaration and
...                                           // initialisations
Java {                                       // Java code
    public void paint (Graphics g) {        // calling Java method
        g.setColor(Color.blue);
        g.fillPolygon(JACIE_x, JACIE_y, 5);
    }
}
...                                           // JACIE code

```

Figure 4.15 *A sample Java code segment in a JACIE program*

```

<embedded java code> ::= Java {
                        <java codes>
                        }

```

All reference to JACIE variables within the `Java` construct should have `JACIE_*` prefix.

Figure 4.15 shows a sample Java code segment in a JACIE program.

4.5 Summary

This chapter has described the specifications of the JACIE language. The design specifications of this language are based on the design principles discussed in Chapter 3. With its template-based style, a syntax-directed editor [92] could be developed to simplify the coding of JACIE scripts. The chapter has also showed how the scripting language of JACIE, with many high level abstractions, simplifies the process of developing interactive and collaborative environment applications. To ease the programming effort further, JACIE uses a “single program” to specify both server and client processes. It is the compiler’s job to generate server and client programs to run on different computers. With its capability to interface with Java, many other Java high-level data types can be used in addition to its simpler built-in low-level data types. This feature also enables experienced programmers to utilise Java for the implementation of complex code segments. A prototype JACIE compiler has been developed based on this specification.

Chapter 5

Compiler Implementation

A prototype of the JACIE Compiler has been developed to translate JACIE source codes into Java as the target language. Conventional abstractions and programming interfaces were considered in the design process so that it would be much easier to understand and implement. This chapter discusses the compiler construction methodology used throughout the development process.

5.1 Application Frameworks

The JACIE compiler adopts certain frameworks for its generated target codes. While this is hidden from the JACIE programmers, the effective and efficient frameworks determine the usefulness of the compiler as well as the effectiveness and the efficiency of the generated codes.

5.1.1 Session Management Protocol and Delivery Management Protocol

The JACIE's session management protocol and delivery management protocol are in addition to floor management protocol and group management protocol previously introduced in Chapter 3. Unlike the floor management protocol and the group management

protocol, which are high-level protocols, the session management protocol and the delivery management protocol are low-level protocols.

JACIE adopts Transmission Control Protocol (TCP) for its session management. TCP has been chosen instead of User Datagram Protocol (UDP) because it provides a number of mechanisms which implement quality of service (QOS) over the connectionless, non-guaranteed Internet Protocol (IP) layer. Some mechanisms of TCP [79] among others, which directly contribute to JACIE-generated applications are as follows:

- *virtual streams* — The connection-oriented stream socket is continuously open for communication at all time. This facility provides an active point-to-point connection between the client machine and the server machine.
- *guaranteed unique delivery* — A sequence number is assigned to each segment transmitted. Upon receiving the segment a checksum is performed. Based on this checksum, the receiving end either responds with acknowledgement (ACK) (if data is intact) or retransmission required (if data is corrupted or missing). After all the segments have been fully transmitted they are reordered to form the original TCP packet. This facility ensures reliable data transmission.
- *full-duplex transmission* — TCP connections simultaneously transmit and receive data. This facility saves transmission time compared to a half duplex connection which requires a turnaround signal.

JACIE does not reserve any specific port for its network service. It is up to the application programmer to assign a port number. As long as each one has a different port number and this number is not in conflict with active well-known ports, any number of JACIE-generated server processes can be run concurrently. In short, TCP may involve network overhead but a reliable data transfer is of more importance for JACIE-generated applications.

Delivery management deals with message transfers from the client to the server and vice-versa during the interactivities. Messages of various types have to be delivered, interpreted

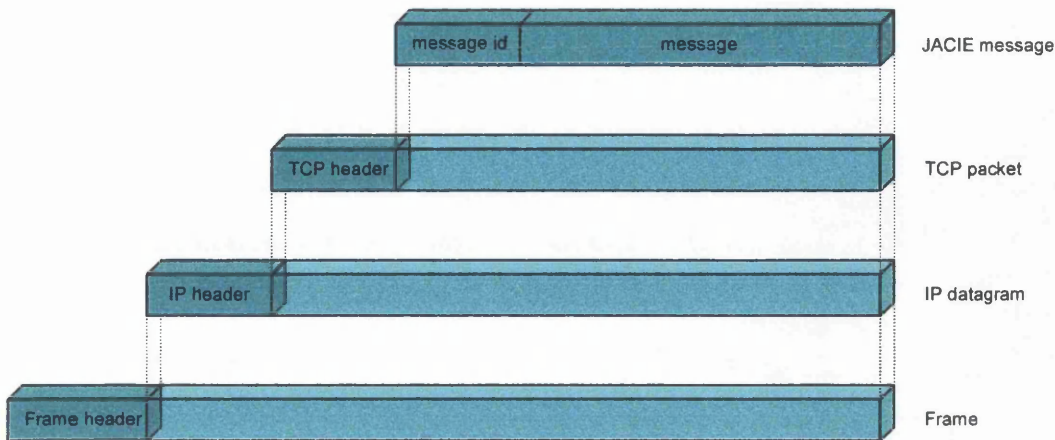


Figure 5.1 JACIE message in relation to other network layers

and responded to accordingly. JACIE manages messages by appending a header to any message to be transmitted. The header is a message identifier and, as its name implies, identifies the type of message to be delivered. Upon receiving this message the respective session handler (to be called later as `session assistant`) will recognise the type of message and will act accordingly or will forward to the logic handler (called `delivery assistant`, if it is on a server, or the `client program`, if it is on a client) for further actions.

Figure 5.1 shows how a JACIE message is encapsulated within TCP/IP protocols before it is delivered. It also shows a message identifier as the header for the JACIE message. In general JACIE message identifiers are classified as either of type system-defined or of type user-defined. The system-defined message identifiers are associated with JACIE standard messages while the user-defined message identifiers are those defined by the application programmer and are to be used in application logic. Each one of the system-defined message identifiers will be explained in Section 5.1.4. User-defined message identifiers are to be used in conjunction with send and receive facilities.

5.1.2 Socket Programming and JACIE's Collaborative Architecture

Socket programming [33,79] has been chosen for the underlying network programming structure for this research. It is expected to give better performance, and through higher-level abstractions of the JACIE language, the complexities are not apparent. Another strong, but complicated feature of Java is multithreading which has been encapsulated in the JACIE language. The multithread issues such as synchronisation (i.e., to prevent *deadlock*, *race condition* and *starvation*) and scheduling of resources has been made invisible to the JACIE programmer.

In a normal stream socket networking model of a client/server environment [33,79], a server process runs on a specific computer and has a socket that is bound to a specific port number. The server waits and listens to the socket for a client to make a connection request. Any client computer requesting a connection should know the hostname of the server or its IP address on which the server process is running and the port number to

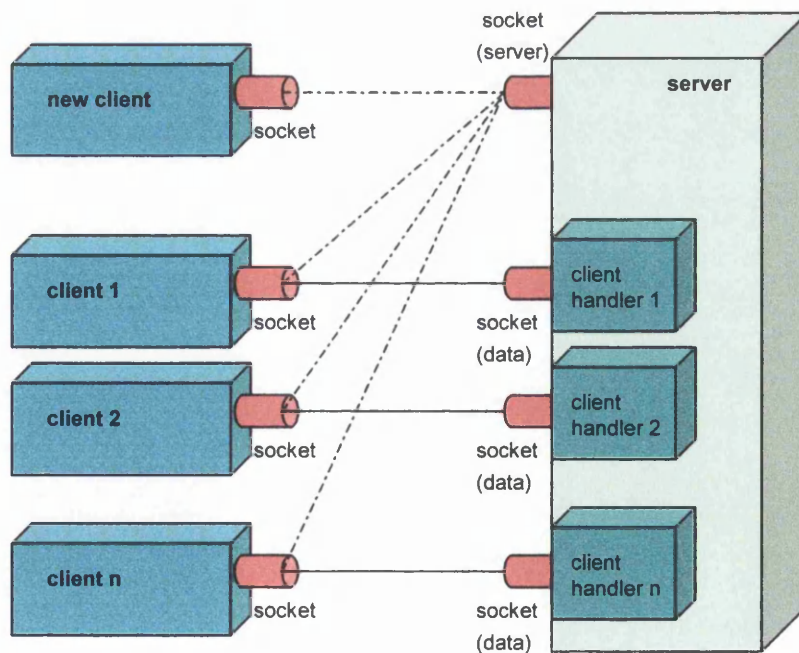


Figure 5.2 General model for network connections between clients and server

which the service is connected. If the client's request is accepted, the server allocates a new socket for a private connection. Further communications between the client and the server are through this private connection. The server then continues to listen to the original socket so that more clients can establish connection and make use of the offered network service. Figure 5.2 illustrates the network connections between clients and server.

Besides handling new connections and assigning new sockets, the server has to read from or write to many sockets which are connected to many clients. The server can service them simultaneously through the use of a multiple thread — one main thread which handles new connections and another thread per client for a private connection.

As shown in Figure 5.2, the role of client handlers is self-contained to a particular client. The client/server interaction for individual connections can take place independent of other connections. The application logic may also be placed in this client handler.

Realistically, the model shown in Figure 5.2 is inadequate if it is to be applied to most interactive and collaborative applications. This research has shown that the role of the server is often more than just to establish and to maintain clients' connections. In addition to managing active connections, the server for an interactive and collaborative application is also responsible for:

- i. managing the shared and individual program logic,
- ii. managing multiple clients' interactivities in an orderly manner,
- iii. managing inter-group and intra-group collaboration.

The new extended model is introduced and adopted in consideration of the server's multiple roles and is illustrated in Figure 5.3.

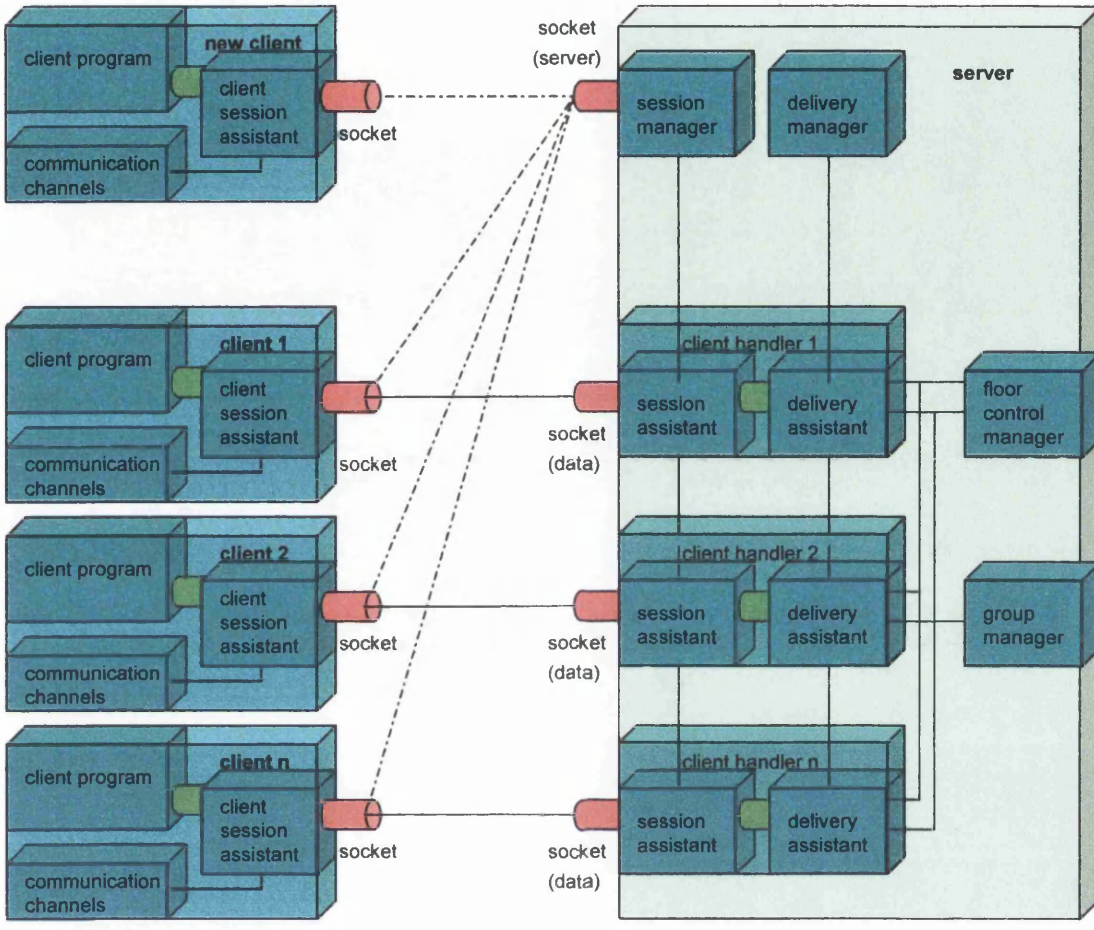


Figure 5.3 Extended model for networked interactive and collaborative applications

As shown in Figure 5.3 the server's multiple roles are manifested by different managers and assistants. On the server side there are session manager, session assistant, delivery manager, delivery assistant, floor control manager and group manager. On the client side there is a client session assistant which is the front-end of the client program. Table 5.1 lists the function of each manager and assistant.

Manager/Assistant Names	Functions
Session Manager	Manages joining of remote users at arbitrary time; keeps track of all client handlers.
Session Assistant	Handles the active connection of each specific client; informs Session Manager of departing client; forwards client messages to and from Delivery Assistant
Delivery Manager	Manages the shared server application logic; maintains program states and behaviours
Delivery Assistant	Handles private or client-specific application logic; maintains client-specific program states and behaviours
Floor Control Manager	Coordinates users' turn accordingly in multiuser and group environments
Group Manager	Manages grouping and intra-group interaction
Client Session Assistant	Client's front-end communicator; forwards server messages to and from client program.

Table 5.1 *Functions of Managers and Assistants in interactive and collaborative applications*

The extended model also shows the use of queues between `session assistant` and `delivery assistant`; `client session assistant` and `client program`. Queues are needed to guarantee the order of message deliveries.

5.1.3 Multithreading

Threads are adopted to allow parallelism or pseudo-parallelism to be combined with sequential execution and blocking system calls [141]. Threads (or sometimes called *lightweight processes*) are normal features in distributed applications [79,112,142].

Implementation based on the above models is made possible with the use of multiple threads. The concurrent running threads solve the problem of hanging while the server is waiting for new connections. Another benefit of threading is, by having a thread associated with each client, the client's connection and application logic can be dealt with independently. Referring to the extended model, the `session manager's` process will also be running in the main thread. It will continuously listen for client connection requests on the server socket. For each successful client connection two other threads are created — one for the `session assistant` and one for the `delivery assistant`. Hence the interactions (reading and writing) between the server `session assistant` and the `client`

`session assistant` may proceed through the data socket. Upon receiving messages from the client, if the messages are related to the application logic they will be sent to the message queue which will be picked up by the `delivery assistant` for actions and response.

In server-mediated interactive applications and group collaborative applications, the `delivery assistant` coordinates its tasks with other managers. It has to refer to the `delivery manager` for shared resources. It has to inform the `floor control manager` upon completing his turn so that the `floor control manager` will assign a new turn accordingly and inform all other active clients through the `session manager`. It also refers to the `group manager` for groupwork activities.

The benefit from multiple threads is not without a cost. Performance of the server may be degraded if it is not implemented carefully. New problems may arise when attempting to share resources which may lead to unexpected results. And above all, programming multithreaded applications is a real challenge.

Race condition may occur between the threads when attempting to access data more or less simultaneously and getting the wrong result [112,141]. This problem can occur in the queues employed between the `session assistant` and the `delivery assistant` as well as between the `client session assistant` and the `client program`. Race condition may also occur in the `delivery manager` which maintains some shared resources to be accessed by all instances of the `delivery assistants`.

Another issue that needs to be considered in multithreaded programming is to ensure fairness to several concurrent threads which are competing for common resources. By definition, a system is fair when each thread gets enough access to limited resource to make reasonable progress [23]. Part of being fair is to prevent *starvation* and *deadlock*. Starvation is a situation in which all the programs continue to run indefinitely but fail to make any progress [111,141]. This is a common phenomenon in any kind of resource allocation scenario. Deadlocks in distributed systems, like the proposed model, are similar

to deadlocks in single processor systems, only worse. They can be communication deadlocks and resource deadlocks [142]. In multithreading programming a deadlock happens when a thread holding a lock is waiting for a lock that is held by another thread while the other thread is waiting for a lock held by the first thread [112,120]. Referring to the above model, the situation can happen in all the *managers* and the *assistants*.

JACIE has addressed these complexities and issues of multithreading by hiding it completely from the user but at the same time has used it wisely and safely in the generated Java codes.

5.1.4 JACIE-aware Interactivities

Network connection messages are communication messages operating at the system level. This type of message deals with connection and disconnection of the virtual stream socket. The collaborative system messages are application messages predefined in JACIE to support basic interactivities between the client and the server. These messages are distinguished by their system-defined message identifiers. Unlike network connection messages, not all predefined collaborative system messages are used by the applications. Different types of collaborative applications require different collaborative system messages. The last type of messages, namely application messages, are application-specific messages and are to be delivered, interpreted and responded by the running client and server processes. As mentioned above, the application messages are distinguished by their user-defined message identifiers. The proposed scripting language JACIE allows the declaration of these message identifiers so that they can be used in conjunction with send and receive statements. The first two message types are hidden behind the scenes and the generated programs will handle these messages accordingly.

Detailed discussions of the three categories of messages and their purposes are presented below. Since client-server interactivities start when a client computer issues a connection request from a waiting server, only three main events are considered. The events are *on*

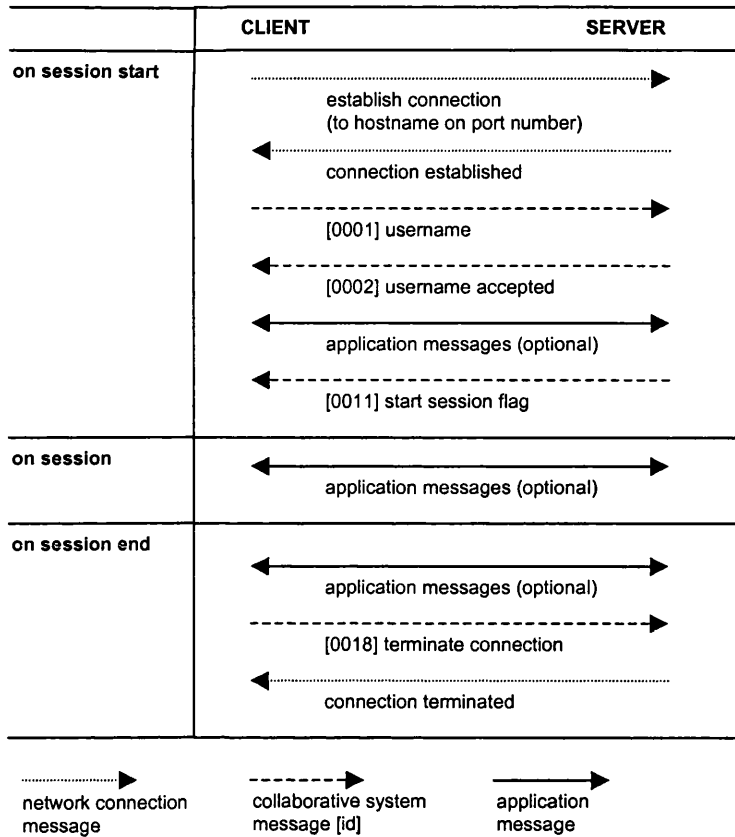


Figure 5.4 Server-based interactivities

session start, *on session*, and *on session end*. The other two server events, *on server start* and *on server end*, are mainly for housekeeping purposes.

Server-based Interactivities

Figure 5.4 shows typical server-based interactivities. Among the different types of interactivities this is the simplest of all. Besides the standard network connection messages for establishing and terminating the connection, there are some minimal collaborative system messages involved. These collaborative system messages are for [0001] sending username, [0002] receiving acknowledgement that the username is being accepted, [0011] receiving start session flag to indicate that the session may start and [0018] informing the server of end of session. The numbers in square bracket “[...]” represent

system-defined message identifiers. Another system message that may be applicable to this type of interactivity is [0005] number of users exceeded. This message may be sent by the server to a new client if the application limits the number of concurrent users and this limit has been reached. The figure also shows that message passing for application messages may take place during the three main events. These are the only messages that need to be dealt with by the JACIE programmer.

Server-mediated Interactivities

Figure 5.5 shows some typical server-mediated interactivities. As expected this type of interactivity is more complicated than the first one. The network connection messages are standard features. During an on session start more system messages are passed between the server and the clients. Besides the regular [0001] username, [0002] username accepted and [0011] start session, other collaborative system messages involved are [0007] user number, [0008] online users, [0009] waiting and [0010] waiting over. The user number message sent by the server contains the client's user number which later is used in conjunction with turn message to determine whether the current turn is his or her turn. Also, during this stage, the client will receive several online user messages which contain information on the current online users — their usernames and user numbers. The server will also inform other online users of his or her presence. If the number of users imposed has not been reached, the server will also send a waiting flag to notify all online users to wait for more new connections. As expected, a waiting event will be generated. Again if a new connection is successfully established the new online users will be notified to all. System message waiting over will be sent to all if enough users are currently online and only then the session will start.

During the active session at least two more collaborative system messages will be in use (if floor control protocol is not contention). The messages are [0014] turn and [0016] pass turn. The turn message carries the current turn number from which a turn event will be generated if the current turn number matches the user number. When the user passes his



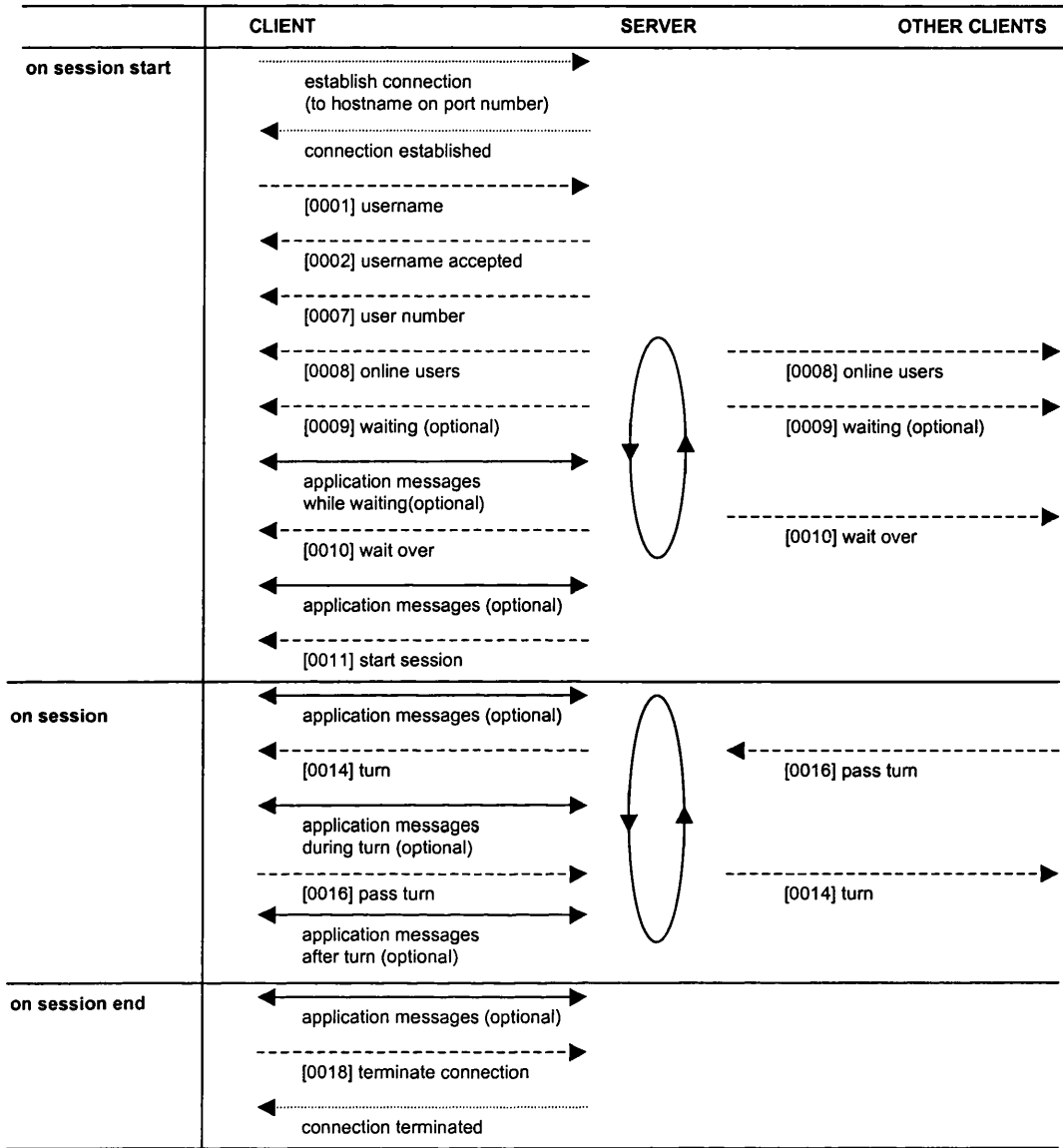


Figure 5.5 Server-mediated interactivities

or her turn, a pass turn message will be sent to the server, and the server (with the help of floor control manager) will notify others of a new turn. The process goes on until the session is ended.

Also shown in Figure 5.5, message passing for application messages can occur in many situations. The first one is when waiting event is generated and the second is after the number of users set has been reached but before the main session starts. During the active

session message passing can occur in at least 3 situations -- at any time during the session, during his or her turn and after he or she passes the turn. Not all situations need to be considered by the JACIE programmer. It all depends on the application requirements.

There are also other messages that may be delivered but not shown in the figure. They are [0003] username accepted with observer status, [0004] username similar to the existing online user, [0005] number of users exceeded, [0006] number of users and observers exceeded and [0017] reserve turn. The proposed tool supports the management of online users with an observer status. Unlike the regular users, the observers can get the current state of the running application but as passive viewers they are not part of the collaborative users. The observers can join and leave at any time without interrupting the active session flow. JACIE also requires the username to be unique. When the server recognises that the username sent is similar to the existing user or the number of users exceeded or the number of observers exceeded, the client computer will be notified, a special message will be displayed and the connection will be terminated gracefully. A reserve turn message is a special message used in conjunction with the reservation floor control protocol. This message is issued by the client and the server, upon receiving it (with the help of `floor control manager`), will place the user's turn in queue.

Group Collaboration Interactivities

Interactivities between clients and servers in group collaboration applications are quite similar to the server-mediated interactivities discussed above. As shown in Figure 5.6 the obvious difference is two additional messages in `on session start` phase. The messages are [0012] user's group number that notifies which group the user belongs to and [0013] user-group information that associates all online users with available groups.

As mentioned earlier, in JACIE's group management, the server does not determine an individual group member's turn but rather it needs to be resolved at the group level. For that reason, in `on session` phase, the turn message is replaced with [0015] group turn. In

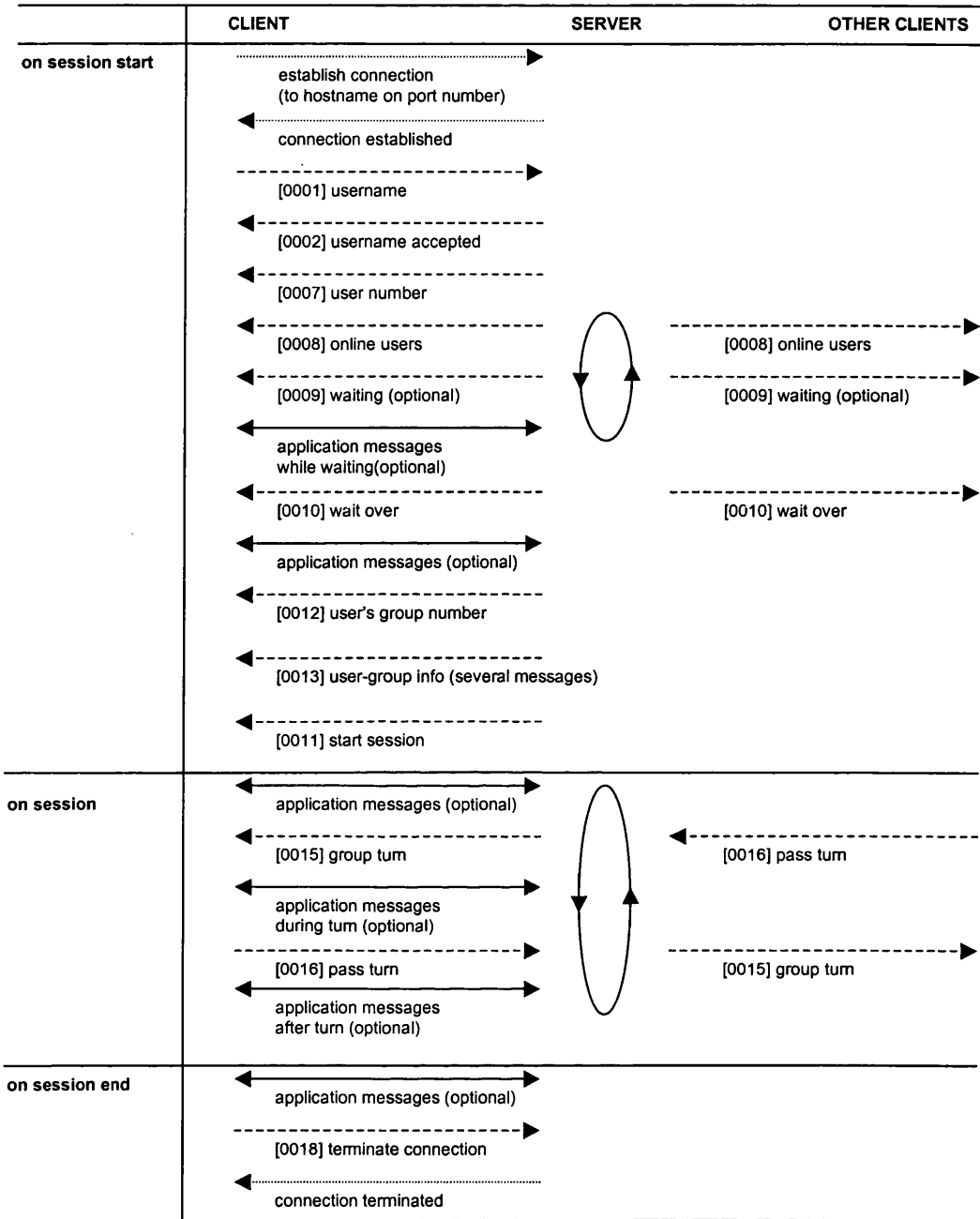


Figure 5.6 Group collaboration interactivities

effect, if one member of the group sends [0016] pass turn message, his action will represent the group decision. Other pass turn messages sent by any of the group members will be ignored by the server.

Identifier No	Identifier Definition	Interaction Type		
		1	2	3
0001	Username	/	/	/
0002	Username accepted	/	/	/
0003	Username accepted with observer status		/	/
0004	Username similar to the existing online users		/	/
0005	Number of users exceeded		/	/
0006	Number of users and observers exceeded		/	/
0007	User number		/	/
0008	Current online users (their user numbers and names)		/	/
0009	Waiting for more users		/	/
0010	Enough users – ready to start		/	/
0011	Start session	/	/	/
0012	User's group number			/
0013	User-group information (User numbers and group numbers)			/
0014	Turn number		/	
0015	Group turn number			/
0016	Pass turn		/	/
0017	Reserve turn		/	/
0018	Terminate connection	/	/	/

Table 5.2 JACIE's system-defined message identifiers

Table 5.2 lists all the system-defined message identifiers supported by JACIE. As far as the JACIE programmer is concerned, all these are carried out behind the scenes.

5.2 Compiler Tools

5.2.1 Java-based Compiler Tools

There are several Java-based compiler development tools. They include JavaCC (Java Compiler Compiler, which previously was known as Jack), JLex (Java Lexical Analyzer), JFlex (Java Fast Lexical Analyzer) and CUP (Constructor of Useful Parsers). JavaCC [140] which was released by Sun is used to automatically generate parsers with an integrated lexical analyser. During the initial development stage of the JACIE Compiler, the JavaCC release 0.5 was an unfinished product [100]. Recently its new release showed many improved features. One of the significant features of JavaCC is in its capability to generate top-down (recursive descent) parsers as opposed to bottom-up parsers generated by *yacc*-like tools. Additional features are the integration of lexical and grammar

specifications in one file, a tree building preprocessor, documentation generation, etc [101, 151].

JLEX [14], developed at Princeton University, on the other hand, was quite a mature product even at its early release. This lexical analyser generator works together with CUP [78]. JLex and CUP follow closely to the well-known lex and yacc [11]. An optimised version of JLex, JFlex, [94] provides a faster and efficient implementation. Due to the wide availability of reference materials on JLex/JFlex and CUP [5] they were adopted in the development of JACIE compiler. JFlex was chosen for its faster operations as compared to JLex.

5.2.2 JFlex — Java Fast Lexical Analyser

JFlex uses *regular expressions* for specifying lexical tokens. This abstraction is used in many modern compilers. The formalism adopted by JFlex is the finite automaton. The specifications include keywords, operators, comments and literals. JFlex specifications also include lexical rules, regular expressions and actions (written in Java code) which are executed when the scanner matches the associated regular expression. The language-specific regular expressions were converted to nondeterministic finite automata (NFAs) and then converted to deterministic finite automata (DFAs). The final output of JFlex is a Java class named `Scanner` — a lexical analyser that interpretes the DFA and returns the token values to be used in the next phase of the compiler.

5.2.3 CUP — Constructor of Useful Parsers

Like JFlex, CUP also complies with another useful abstraction used in the modern compilers for parsing - *context-free grammars*. It is a tool for generating Look-Ahead Left-to-Right (LALR) parsers from simple specifications. It serves the same role as the widely used yacc [88]. CUP written in Java, uses embedded Java codes, and produces parsers which are implemented in Java. The CUP specifications consist of *package and*

import specifications, user code components, symbol (terminal and non-terminal) lists, precedence declarations and the language grammar.

The package specification is optional and is used to bundle related classes generated by the CUP. The import specification is also optional and the package name specified will be copied to the source file for the parser class allowing various classes from that package to be used directly in user-supplied action codes. A series of optional declarations that allow user code to be included as part of the generated parser is specified in user code components. The symbol lists, precedence declarations and the grammar are for syntax analysis and semantic actions.

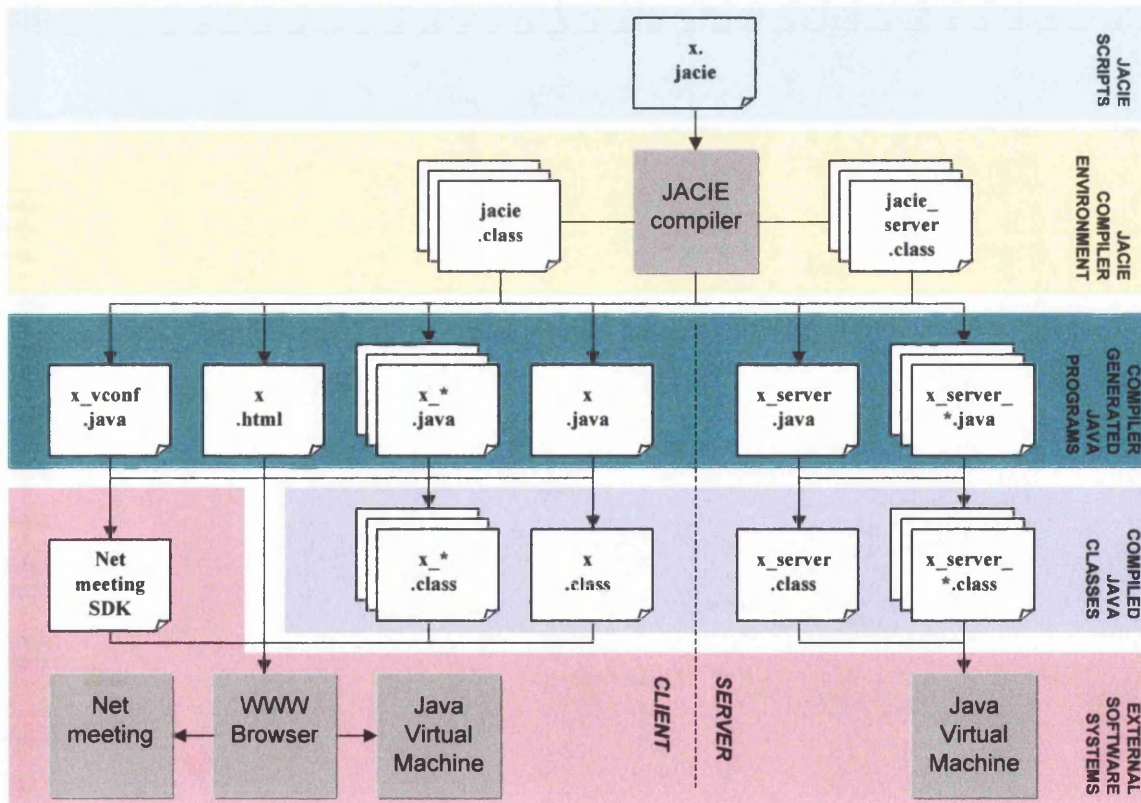


Figure 5.7 Software architecture of JACIE

5.3 JACIE Compiler Environment

Figure 5.7 illustrates the software architecture of JACIE. It shows the different layers of processes that a JACIE script has to go through before it can be deployed. In general, like any other compiler environment, the JACIE scripts will undergo a lexical analysis and a syntax parsing process. During the code generation process, several JACIE compiler classes will be used. The explanation below gives a detailed outlook of the development environment.

5.3.1 Lexical Analysis

Lexical specifications of the JACIE language have been written in JFlex format. They consist of macro declarations which are abbreviations for regular expressions.

The code below is one part of the JFlex specifications for the JACIE language.

```

/* main character classes */
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]

WhiteSpace = {LineTerminator} | [ \t\f]

/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}

TraditionalComment = "/*" [^*] {CommentContent} \** "/"
EndOfLineComment = "//" {InputCharacter}* {LineTerminator}
DocumentationComment = "/*" {CommentContent} \** "/"

CommentContent = ( [^*] | \**[^*/] ) *

/* identifiers */
Identifier = [:jletter:][:jletterdigit:]*

/* integer literals */
IntegerLiteral = 0 | [1-9][0-9]*

/* floating point literals */
FloatLiteral = {FLit1}|{FLit2}|{FLit3}|{FLit4}

FLit1 = [0-9]+ \. [0-9]+ {Exponent}?
FLit2 = \. [0-9]+ {Exponent}?
FLit3 = [0-9]+ {Exponent}
FLit4 = [0-9]+ {Exponent}?

Exponent = [eE] [+|-]? [0-9]+

/* string literals */
StringCharacter = [^\r\n\"\\]

```

The macros shown are for comments, identifiers and numeric data literals. As shown above and as described in Chapter 4, JACIE supports three types of comments, namely `TraditionalComment`, `EndOfLineComment` and `DocumentationComment`. This is followed by the `Identifier` macro which matches each string that starts with a character of `jletter` followed by zero or more characters of class `jletterdigit`. The `jletter` and `jletterdigit` correspond to the Java function `Character.isJavaIdentifierStart` and `Character.isJavaIdentifierPart`. Also shown are macros for `IntegerLiteral` and `FloatLiteral`. They follow the standard formats for integer and floating point literals.

The following JFlex code shows some of the lexical rules that represent keywords, literals, comments and identifiers for the JACIE language. The returned token values will be used by CUP.

```

/* keywords */

"int"           { return symbol(sym.INT); }
"float"        { return symbol(sym.FLOAT); }
"boolean"      { return symbol(sym.BOOLEAN); }
"string"       { return symbol(sym.STRING); }
"image"        { return symbol(sym.IMAGE); }

"JACIE"        { return symbol(sym.JACIE); }
"java"         { return symbol(sym.JAVA); }

"application"  { return symbol(sym.APPLICATION); }
"applet"       { return symbol(sym.APPLET); }
"appletlauncher" { return symbol(sym.APPLETLAUNCHER); }
"text"         { return symbol(sym.TEXT); }
"name"         { return symbol(sym.NAME); }
"configuration" { return symbol(sym.CONFIGURATION); }
"messages"     { return symbol(sym.MESSAGES); }
"client"       { return symbol(sym.CLIENT); }
"server"       { return symbol(sym.SERVER); }
"implementation" { return symbol(sym.IMPLEMENTATION); }
...

/* separators */
"("            { return symbol(sym.LPAREN); }
")"           { return symbol(sym.RPAREN); }
"{"           { return symbol(sym.LBRACE); }
"}"           { return symbol(sym.RBRACE); }
...

/* operators */
"="           { return symbol(sym.EQ); }
">"          { return symbol(sym.GT); }
"<"          { return symbol(sym.LT); }
"!"          { return symbol(sym.NOT); }
...

```

```

/* boolean literals */
"true"      { return symbol(sym.BOOLEAN_LITERAL, new Boolean(true)); }
"false"     { return symbol(sym.BOOLEAN_LITERAL, new Boolean(false)); }

/* null literal */
>null"     { return symbol(sym.NULL_LITERAL); }

/* string literal */
\"         { yybegin(STRING); string.setLength(0); }

/* numeric literals */
{IntegerLiteral} { return symbol(sym.INTEGER_LITERAL, new Integer(yytext())); }
{FloatLiteral}   { return symbol(sym.FLOATING_POINT_LITERAL,
                             new Float(yytext().substring(0,yylength()-1))); }

/* comments */
{Comment}       { /* ignore */ }

/* whitespace */
{WhiteSpace}    { /* ignore */ }

/* identifiers */
{Identifier}    { return symbol(sym.IDENTIFIER, yytext()); }
...

```

5.3.2 Syntax Parsing

A CUP specification has a preamble. This preamble declares list of terminal and non-terminal symbols, followed by the JACIE grammar rules.

There were 166 terminal symbols and 169 non-terminal symbols declared in the prototype of the JACIE compiler. Below are parts of the code from the `jacie.cup` file that declare the terminal and non-terminal symbols. The terminal symbols are related to the ones defined in the lexer described earlier.

```

terminal JACIE;
terminal JAVA;

terminal APPLICATION;
terminal APPLET;
terminal APPLETLAUNCHER;
terminal TEXT;
terminal NAME;
terminal CONFIGURATION;
terminal MESSAGES;
terminal CLIENT;
terminal SERVER;
terminal IMPLEMENTATION;
...

// data types
terminal INT, FLOAT, BOOLEAN, STRING, IMAGE;

```

```

...
// symbols
terminal LPAREN, RPAREN, LBRACE, RBRACE, LBRACK, RBRACK;
terminal SEMICOLON, COMMA, UNDERSCORE, EQ;
...
// literals
terminal java.lang.Number INTEGER_LITERAL;
terminal java.lang.Number FLOATING_POINT_LITERAL;
terminal java.lang.Boolean BOOLEAN_LITERAL;
terminal java.lang.String STRING_LITERAL;
terminal java.lang.String IDENTIFIER;
terminal NULL_LITERAL;
...
// expressions
non terminal Expression assignment;
non terminal Expression assignment_expression;
non terminal Expression conditional_expression;
non terminal Expression conditional_or_expression;
non terminal Expression conditional_and_expression;
non terminal Expression inclusive_or_expression;
non terminal Expression exclusive_or_expression;
...
// statements
non terminal Statement compound_statement;
non terminal StatementList compound_statement_option;
non terminal StatementList statement_list;
non terminal Statement statement;
...
non terminal Statement expression_statement;
non terminal Statement if_then_statement;
non terminal Statement if_then_else_statement;
non terminal Statement for_statement;
non terminal Statement while_statement;
non terminal Statement return_statement;
non terminal Statement exit_statement;
non terminal Statement text_input_statement;
non terminal Statement print_text_statement;
...

```

Below is another extract from the `jacie.cup` file. It shows a sample of JACIE grammar rules for `client_program_implementation` and `server_program_implementation`.

```

client_program_implementation
    ::= DECLARATION variable_and_method_declaration_statement
        ON CANVAS LBRACE compound_statement_option:slist1 RBRACE
        { : new OnCanvasStatement(slist1); : }
        ON SESSION START LBRACE compound_statement_option:slist2 RBRACE
        { : new OnSessionStartStatement();
          OnSessionStartStatement.init(slist2,1); : }
        ON SESSION LBRACE compound_statement_option:slist3 RBRACE
        { : new OnSessionStatement();
          OnSessionStatement.init(slist3,1); : }
        ON SESSION END LBRACE compound_statement_option:slist4 RBRACE
        { : new OnSessionEndStatement();
          OnSessionEndStatement.init(slist4,1); : }
        ;

```

```

server_program_implementation
    ::= DECLARATION variable_and_method_declaration_statement
        ON SERVER START LBRACE compound_statement_option:slist1 RBRACE
        { : new OnServerStartStatement(slist1); : }
        ON SESSION START LBRACE compound_statement_option:slist2 RBRACE
        { : OnSessionStartStatement.init(slist2,2); : }
        ON SESSION LBRACE compound_statement_option:slist3 RBRACE
        { : OnSessionStatement.init(slist3,2); : }
        ON SESSION END LBRACE compound_statement_option:slist4 RBRACE
        { : OnSessionEndStatement.init(slist4,2); : }
        ON SERVER END LBRACE compound_statement_option:slist5 RBRACE
        { : new OnServerEndStatement(slist5); : }
    ;

```

Since a compiler should do more than recognising whether or not a sentence belongs to the grammar of a language, a semantic analysis is required.

5.3.3 Semantic Analysis and Translation

Semantic actions of the JACIE language are encoded as fragments of Java program code attached to grammar productions of CUP specifications. Referring to the above CUP specifications, these semantic actions appear in the right side as code strings within { : ... : } delimiters. Whenever the parser reduces a rule, it will execute the corresponding semantic action fragment. Abstract syntax of JACIE statements are among other codes specified in these fragments. The translation of abstract syntax into Java codes completes the JACIE compiler processes. Since the target language is Java which is a high level language, other phases of a compiler are not implemented in the JACIE compiler.

Table 5.3 lists all the supporting components of the JACIE compiler together with their corresponding descriptions. The compiler consists of a collection of Java codes generated by JFlex and CUP, abstract syntax of JACIE statements and Java code templates for client and server programs. Other files are supporting Java classes called during the compilation of JACIE programs and some precoded classes automatically integrated into the client and server programs.

File name	Description
jacie.flex	The JFlex lexical specification of the JACIE language.
jacie.cup	The CUP specification that contains the JACIE grammar rules and semantic actions.
Scanner.java	JACIE lexical analyser class (generated by JFlex upon compilation of jacie.flex)
parser.java	The Java class that contains a production table and an action table (generated by CUP upon compilation of jacie.cup)
sym.java	The java class that contains symbol constants (generated by CUP upon compilation of jacie.cup)
Lexer.java	An interface class called by scanner.java (supplied by CUP)
EscapedUnicodeReader.java	A utility class to handle UNICODE characters (supplied by CUP)
Timer.java	A utility class to determine compilation time (supplied by CUP).
jaciec.bat	A DOS batch file that provides a shortcut for compilation of programs written in JACIE script.
JACIECParser.java	The JACIE parser – the main program for the JACIE compiler.
JACIECConfig.java	A utility class used by the JACIE compiler to maintain application parameters.
JACIECVariableTable.java	A utility class used by the JACIE compiler to maintain variable symbol table for type checking.
JACIECMessageTable.java	A utility class used by the JACIE compiler to maintain message identifier symbol table.
JACIECGridTable.java	A utility class used by the JACIE compiler to maintain JACIE grid symbol table.
JACIECStatement.java	Abstract syntax of all JACIE statements with translation to Java code.
JACIECExpression.java	Abstract syntax of all JACIE expressions with translation to Java code.
JACIECChannel.java	A utility class that contains channel constants.
JACIECFloorMgmtProtocol.java	A utility class that contains floor management protocol constants.
JACIECGroupProtocol.java	A utility class that contains group protocol constants.
JACIECPrintFile.java	A utility class for printing to output file.
JACIECFilecopy.java	A utility class for copying precoded class files to a target directory.
JACIECApplicationTemplate.java	A template that creates a class for the main structure of the JACIE-generated client application (x.java).
JACIECAppletTemplate.java	A template that creates a class for the main structure of the JACIE-generated client applet (x.java) and the corresponding HTML file (x.html).
JACIECFrameTemplate.java	A template that creates a window frame class used by applications or applets with launchers (x_Frame.java).
JACIECContainerTemplate.java	A template that creates container class to be embedded in web page or the frame and in which most client logics are specified (x_Container.java).

Table 5.3 JACIE compiler files.

File name	Description
JACIECMenuBarTemplate.java	A template that creates custom designed menubar that contains standard user-interfaces for connection/disconnection, channel buttons, a userlist button and about/help button (<code>x_MenuBar.java</code>).
JACIECCanvasTemplate.java	A template that creates canvas upon which all graphics are performed and to be in the container (<code>x_Canvas.java</code>).
JACIECMessageBarTemplate.java	A template that creates the text input bar, the local message bar and the server message bar (<code>x_MessageBar.java</code>).
JACIECClientSessionAssistantTemplate.java	A template that creates a class that is responsible for handling message transfers for the client program (<code>x_SessionAssistant.java</code>).
JACIECUserListTemplate.java	A template that creates window frame class that displays users and groups (<code>x_UserList.java</code>).
JACIECChatChannelTemplate.java	A template that creates window frame class for chat channel (<code>x_ChatChannel.java</code>).
JACIECNetMeeting.java	A program that creates special class that can interface to Microsoft NetMeeting (<code>x_vconf.java</code>).
JACIECServerTemplate.java	A template that creates a class for the main structure of the JACIE-generated server application (<code>x_Server.java</code>).
JACIECSessionManagerTemplate.java	A template that creates a server class that is responsible for establishing new client connections (<code>x_Server_SessionManager.java</code>).
JACIECDeliveryManagerTemplate.java	A template that creates a server class that handles all shared server logics (<code>x_Server_DeliveryManager.java</code>).
JACIECFloorManagerTemplate.java	A template that creates a server class that handles floor management (<code>x_Server_FloorManager.java</code>).
JACIECGroupManagerTemplate.java	A template that creates a server class that handles group management (<code>x_Server_GroupManager.java</code>).
JACIECSessionAssistantTemplate.java	A template that creates a server class that handles each client connection (<code>x_Server_SessionAssistant.java</code>).
JACIECDeliveryAssistantTemplate.java	A template that creates a server class that handles most client specific server logics (<code>x_Server_DeliveryAssistant.java</code>).
JACIEBox.java	A precoded class used by some JACIE user-interface that draws etched rectangle with or without a title.
JACIEImageButton.java	A precoded class for custom designed buttons that support images.
JACIEGrid.java	A precoded class for maintaining and manipulation of JACIE grids.
JACIEAboutHelp.java	A precoded class for frame that displays program help file.
JACIEMessage.java	A precoded class for JACIE delivery messages.
JACIEMessageQueue.java	A precoded class for handling JACIEMessage in queue.

Table 5.3 JACIE compiler files (cont).

In addition to the files described in Table 5.3, there are also button images which form part of the JACIE standard user-interfaces. They are `Connect.gif`, `Disconnect.gif`, `Chat.gif`, `Whiteboard.gif`, `Voice.gif`, `Video.gif`, `Users.gif` and `About.gif`. They are grouped in the `graphics` directory of the JACIE compiler and are copied as needed by the JACIE compiler to the target directory.

5.3.4 JACIE-generated Java Programs

As mentioned earlier, the JACIE compiler generates Java source codes to be further compiled by any Java compiler to produce bytecodes — the platform-independent codes interpreted by any Java interpreter which is also known as Java Virtual Machine (JVM).

For the purpose of illustration, an arbitrary collaborative application written in the JACIE language is named `x.jacie`. By compiling the JACIE script using the following command,

```
jaciec x.jacie
```

the JACIE compiler will generate files which adhere to naming conventions shown in Table 5.4.

<code>x.java</code>	a Java applet or application that defines the main structure of the client program;
<code>x_*.java</code>	a set of application specific classes in Java generated by the compiler for supporting client operations;
<code>x.html</code>	an HTML file with a tag that includes the corresponding applet in the Web page;
<code>x_server.java</code>	a Java application that defines the main structure of the server program;
<code>x_server_*.java</code>	a set of application specific classes in Java for supporting server operations;
<code>x_vconf.java</code>	a program which is written in Java for supporting voice, video and whiteboard channels through Microsoft NetMeeting, and is generated only if there are such channels defined in <code>x.jacie</code> .
<code>JACIE*.java</code>	A set of precoded classes in Java for supporting client and server operations

Table 5.4 Naming conventions of JACIE-generated files

The `x_*.java` classes consist of `x_Frame`, `x_Container`, `x_MenuBar`, `x_Canvas`, `x_MessageBar`, `x_SessionAssistant`, `x_UserList` and `x_ChatChannel`. The `x_server_*.java` classes are `x_server_SessionManager`, `x_server_FloorManager`, `x_server_DeliveryManager`, `x_server_GroupManager`, `x_server_SessionAssistant` and `x_server_DeliveryAssistant`. The classes with names starting with JACIE are pre-coded files and are to be copied, if required, during compilation. They are `JACIEBox`, `JACIEImageButton`, `JACIEGrid`, `JACIEAboutHelp`, `JACIEMessage` and `JACIEMessageQueue`. All of these pre-coded classes are to support client operations, except for the `JACIEMessage` and the `JACIEMessageQueue` which are also used for server operations. The detailed descriptions of these files will be discussed in the following section.

During the early development of the JACIE prototype compiler, Microsoft NetMeeting has been chosen as part of the system for its wide availability and its support for a range of videoconferencing hardware. Microsoft Visual J++ has been used for integration of Java with Microsoft ActiveX APIs [103]. Platform-independent voice, video and whiteboard channels are feasible in the future implementation.

5.4 Software Components of JACIE-generated Java Programs

Utilising the object-oriented nature of Java programming, the Java codes generated by the JACIE compiler are organised in such a way that they become a collection of flexible, modular and reusable classes. This is important to simplify the code generation process of the JACIE compiler and to provide a manageable means for JACIE programmers to understand, modify or enhance the generated codes, if required.

5.4.1 Types of JACIE-generated Java Programs

The JACIE scripting language can produce one of four types of client programs — an applet, a separate window invoked by an image launcher within an applet, a separate window invoked by a text button within an applet, and a normal application (refer to previous sections for details). For the purpose of illustration we will call these four client programs JCApplet1, JCApplet2, JCApplet3 and JCApp respectively.

Figure 5.8 shows the screen shots of different types of JACIE-generated client programs.

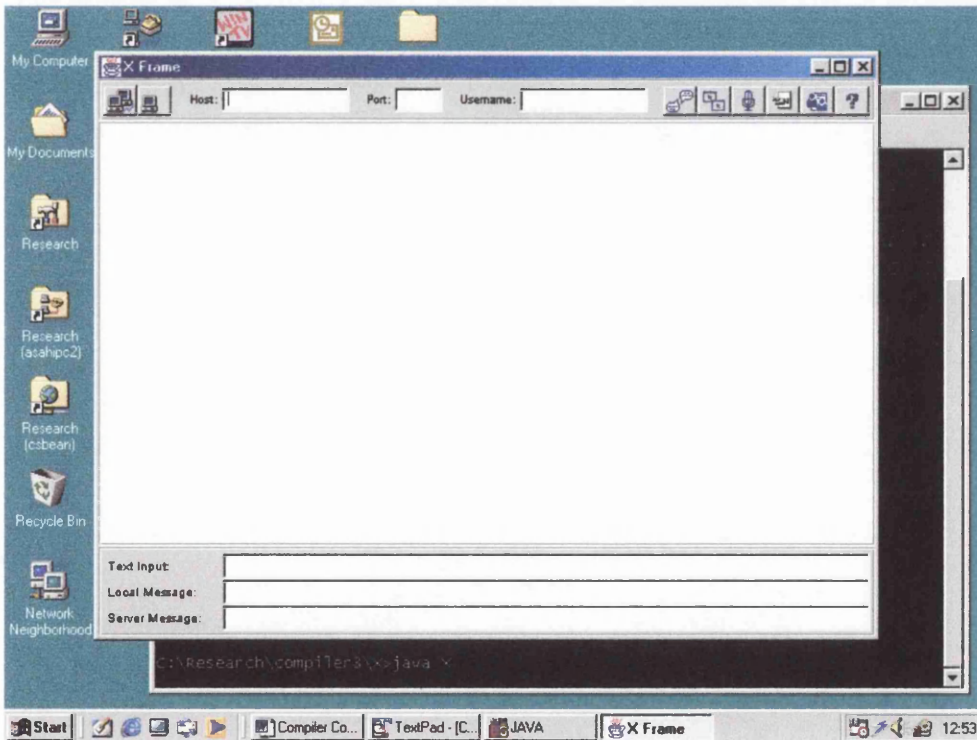


Figure 5.8a *The user interface for JACIE-generated application (JCApp)*

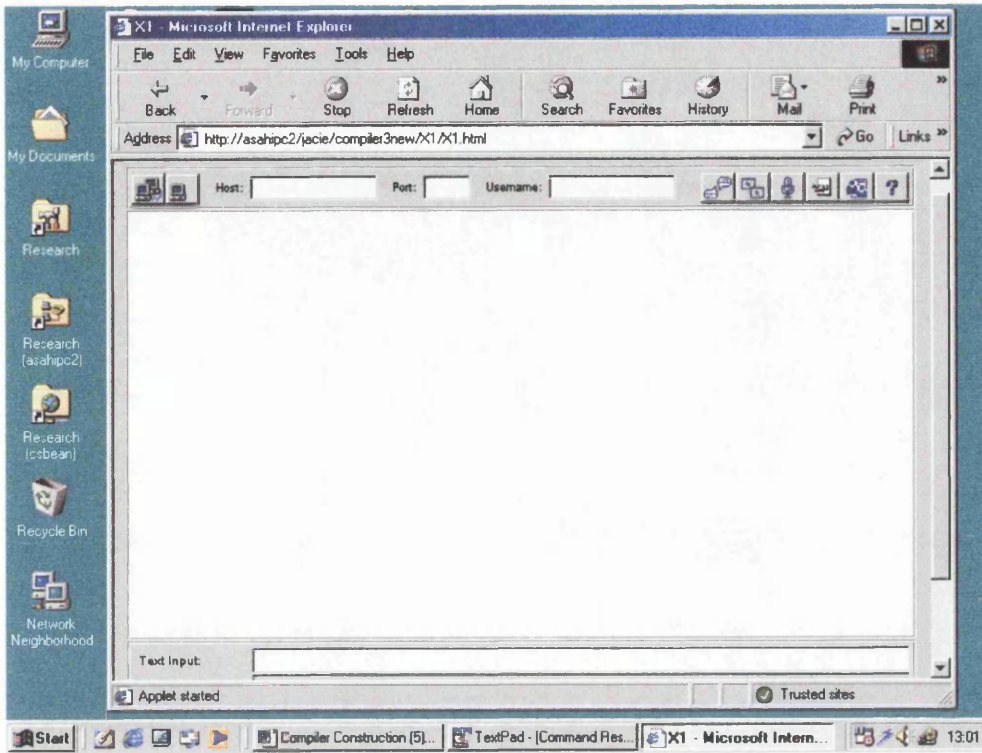


Figure 5.8b The user interface for JACIE-generated applet (*JCAppllet1*)

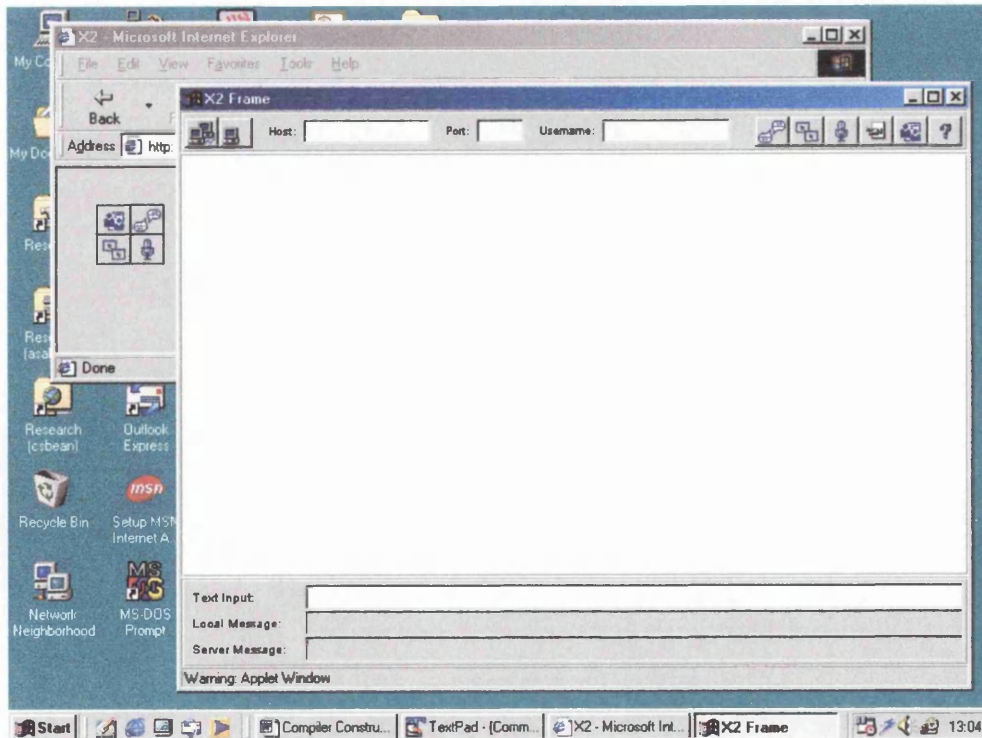


Figure 5.8c The user interface for JACIE-generated applet with an image button launcher (*JCAppllet2*)

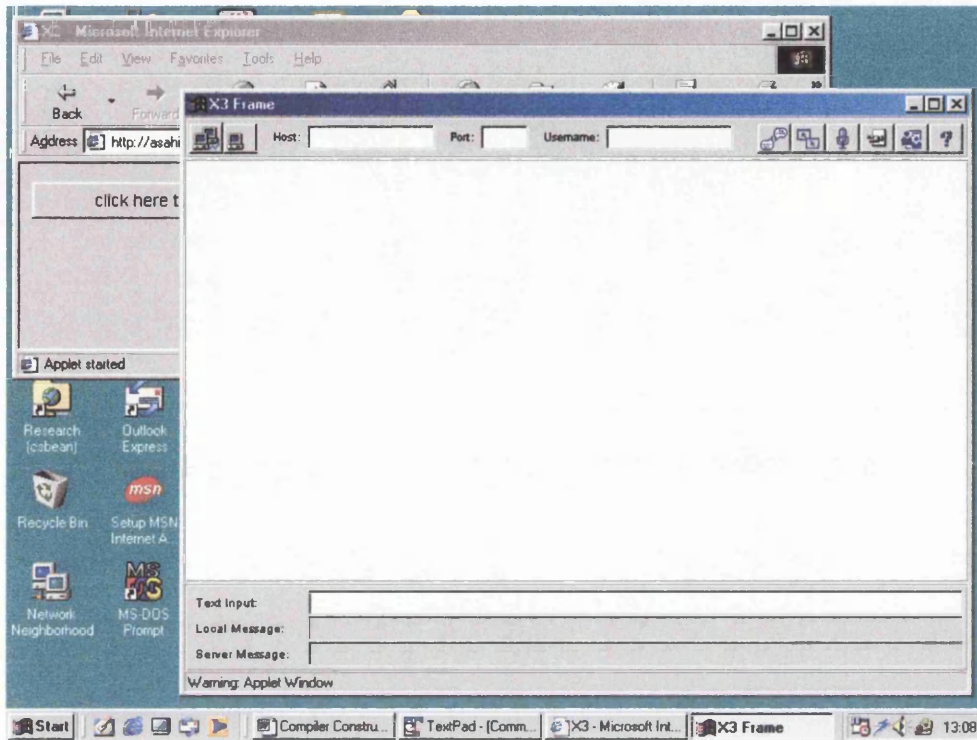


Figure 5.8d The user interface for JACIE-generated applet with a text button launcher (*JCApplet3*)

5.4.2 Software Components of JACIE-generated Client Programs

JACIE employs a *composite* design pattern for its standard user interface [58]. Figure 5.9 shows the nested layout of the components. The very outmost component is an applet window itself (if it is an applet) or a frame (if it is an application or an applet with a separate window). This applet or frame has a container that holds all other components. Basically there are three main panel components — menu bar, canvas and message bar. Each one of these panels are surrounded by a special box, an etched rectangle, for better visual appearance.

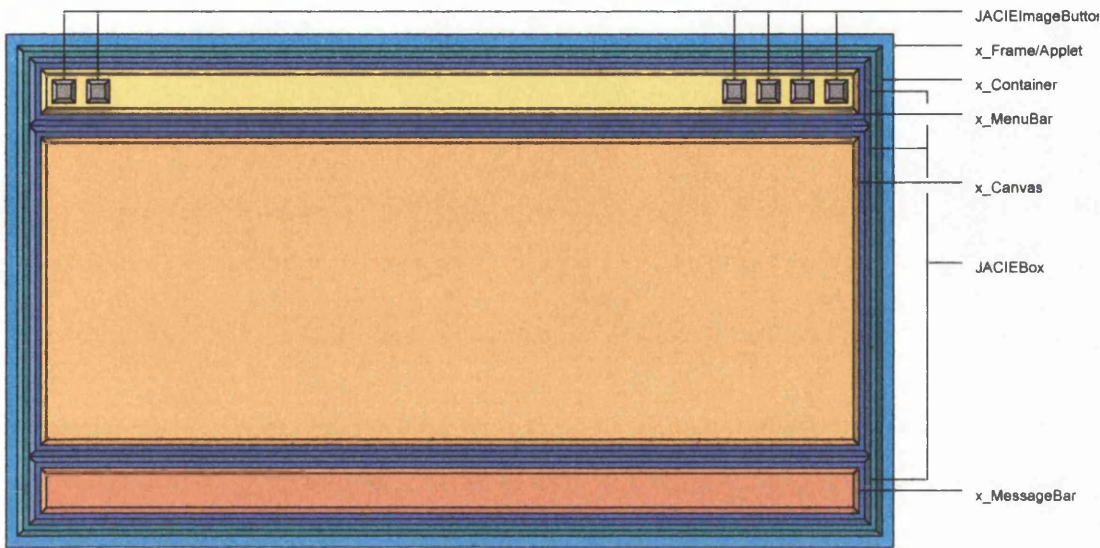


Figure 5.9 Nested layout diagram of JACIE user interface

Figure 5.10 and Figure 5.11 show the class diagram of the client programs. Referring to Figure 5.10, the first client program is the `x1` class, a Java applet that runs within a Java-enabled browser. This class creates and contains an `x_Container` object, a subclass of Java Abstract Window Toolkit (AWT)'s container (`java.awt.Container`) holds all the GUI components of the client program. The `x_Container` class also implements `java.lang.Runnable`, a standard Java interface for thread control. The second and the third client programs are the `x2` and the `x3`. These two small applets only contain a launcher that creates `x_Frame`, a window object for the client program. The fourth client program, `x`, the application, also uses this `x_Frame`. The `x_Frame`, which is a subclass of a `java.awt.Frame`, creates and contains an `x_Container`, the same container used by the applet `x1`.

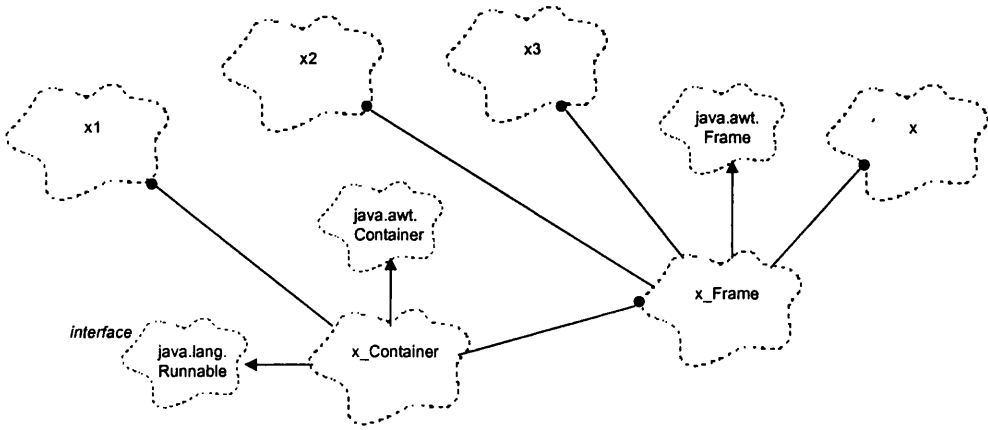


Figure 5.10 A class diagram of JACIE-generated client programs (1)

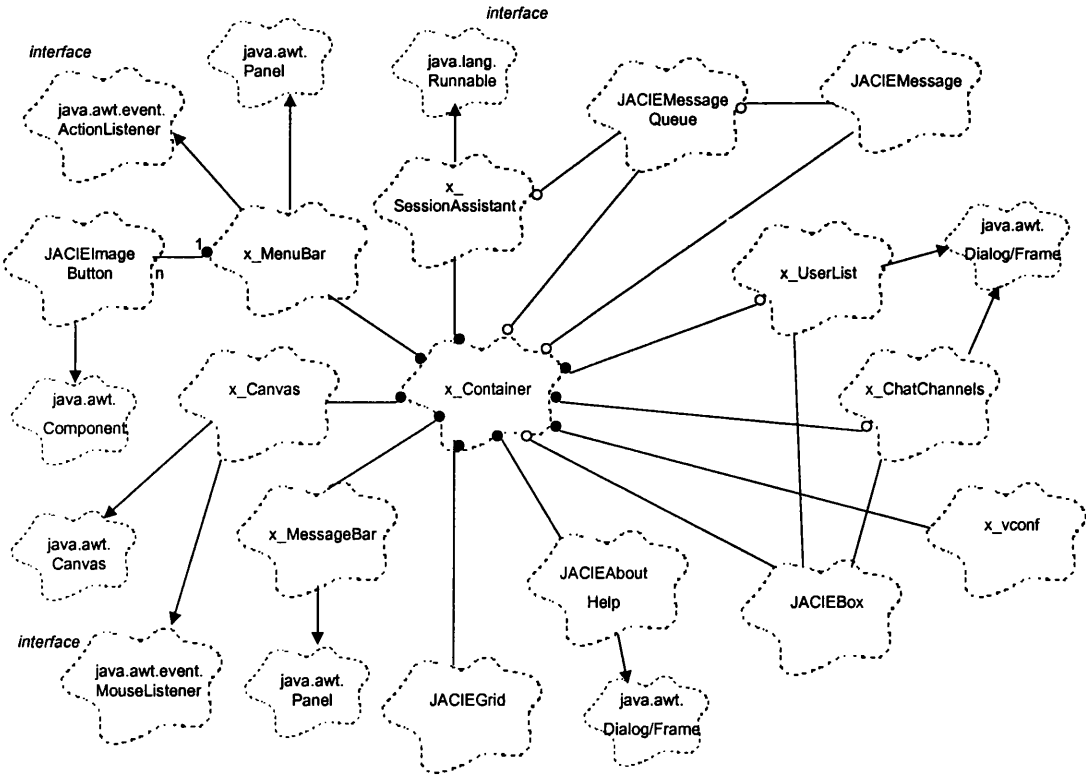


Figure 5.11 A class diagram of JACIE-generated client programs (2)

Referring to Figure 5.11, `x_Container` creates and contains three major components — `x_MenuBar`, `x_Canvas` and `x_MessageBar`. `x_MenuBar` and `x_MessageBar` are subclasses of `java.awt.Panel`, while `x_Canvas` extends `java.awt.Canvas`. Besides these three class components, the JACIE compiler also creates other supporting classes namely `JACIEBox`, `JACIEImageButton`, `JACIEAboutHelp`, `x_SessionAssistant`, `x_UserList`, `x_ChatChannel`, `x_vconf`, `JACIEMessage` and `JACIEMessageQueue`.

The `JACIEImageButton`, a lightweight component to enhance the Java text-only standard button, extends `java.awt.Component`. Since Java limits the `java.awt.Dialog` class to be called by `java.awt.Frame`, the `JACIEAboutHelp`, the `x_UserList` and the `x_ChatChannel` classes are implemented as subclasses of `java.awt.Dialog` if they are being embedded in `x_Frame`, but become subclasses of `java.awt.Frame` if they are being embedded in `x_JACIEApplet`.

Another class is the `x_SessionAssistant`, which extends `java.lang.Thread`. An instance of this class handles all the communication interfaces with the server program over the network. This object also creates an instance of `JACIEMessageQueue`, which is used to guarantee the order of message delivery and execution. `JACIEMessageQueue` object handles `JACIEMessage` objects and these objects are to be removed and executed by `x_Container`.

The last class is the `x_vconf`, a special class that uses Microsoft NetMeeting SDK to enable the JACIE-generated client program to interface to Microsoft NetMeeting, if necessary. If the client program is an applet, the JACIE compiler will also create an HTML file, with a tag that includes the corresponding applet in the Web page.

5.4.3 Software Components of JACIE-generated Server Programs

Java server classes produced by the JACIE compiler have been designed with consideration of client-server interactivities (refer to Chapter 3). The main server class is named as `x_Server.java`, where `x` is the name given by the JACIE programmer. Referring to Figure 5.12, this `x_Server` class will create `x_Server_SessionManager`, `x_Server_DeliveryManager`, `x_Server_FloorControlManager` and `x_Server_GroupManager`. Since `x_Server_SessionManager` is executed in independent threads, it extends `java.lang.Thread`.

Not all of these managers are to be generated at compile time. Besides `x_Server_SessionManager`, the others will be generated depending on the interaction protocols being adopted. During runtime, the `x_Server_SessionManager` will create many instances of `x_Server_SessionAssistant` depending on the number of remote

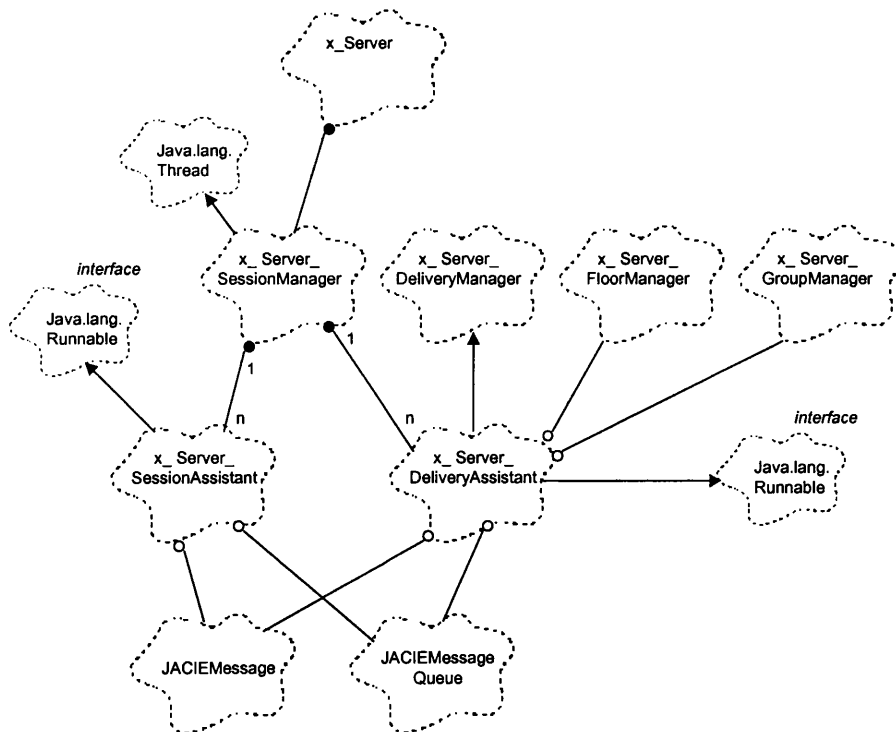


Figure 5.12 A class diagram of JACIE-generated server programs

clients. This connection handler object maintains each client in one thread, and because of this the class is specified as a subclass of `java.lang.Thread`. Each client will also be allocated with a `x_Server_DeliveryAssistant` object. This is where most of the server logics are located. The `x_Server_DeliveryAssistant` extends `x_Server_DeliveryManager` which contains all the shared variables and methods. Since the `x_Server_DeliveryAssistant` is performing separate tasks executed in parallel, the class is also made to implement `java.lang.Runnable`. Another important JACIE-generated class for supporting the server programs is `JACIEMessageQueue` whose instance will be used by each instance of `x_Server_SessionAssistant` and `x_Server_DeliveryAssistant`.

5.5 Running the JACIE Compiler

5.5.1 Configuring the JACIE Compiler

The minimum requirements for the JACIE compiler are the runtime versions of the two compiler tools adopted, JFlex and CUP, and any Java 1.1 development tool. Theoretically, since the JFlex and the CUP were written in the Java language and the newer versions of Java are compatible with older versions, any computer platform that supports Java Virtual Machine (JVM) 1.1 can be used to run the JACIE compiler. For this research, JFlex, CUP and the Sun's Java Development Kit (JDK) 1.1.8 have been installed and configured on an IBM compatible PC. A normal text editor has been used to create the JFlex and the CUP specifications as well as other supporting Java classes for the JACIE language (refer Table 5.3).

The JFlex specification of the JACIE language was named `jacie.flex`. Figure 5.13 shows the output of running the JFlex. As shown, the `scanner.java` file was generated and this scanner was used by the CUP and the compiler operations.

5.5.2 Compiling JACIE Scripts

Any text editor can be used to write a JACIE program. Typically, this program has `.jacie` as the filename extension. Assuming that the JACIE compiler files have been placed in the `jaciec` directory of the root, at the system command line and in this JACIE compiler directory, the command to compile the JACIE script is as follows:

```
C:\jaciec>jaciec filename.jacie
```

Upon compiling the JACIE script, if the file does not contain any errors, a directory named after the application or applet name will be created. Two other subdirectories under this directory, namely `client` and `server`, will also be created as a storage space for all the generated Java codes for the client and the server processes respectively. All the necessary graphic images (for the standard buttons and the application/applet-specified graphics) will be copied to the `client` subdirectory. If the generated client program is meant to be an applet, the HTML file will also be generated and included in the `client` subdirectory. Practically speaking, this HTML file inside which the applet is placed, only contains the bare minimum HTML code. It is up to the JACIE programmer to add more HTML code to make it more presentable. All the Java files are to be compiled with a Java 1.1 compiler to produce Java classes in bytecodes. The server's compiled codes can then be copied to a server machine on which the server process will be running. If the client program is a stand-alone application, the client's compiled codes should then be copied to all clients' machines before the client program can be run. In the case of an applet, the client's compiled codes should be copied to the HTTP (or Web) server on which the applet will be called by means of the HTML file. Constrained by the Java security restriction [23], the HTTP server is also the host on which the server process will be running.

As an example, Figure 5.15 shows an output of compiling `Puzzle2.jacie`, a two-player game that requires remote players to compete rearranging scrambled pieces. All the JACIE compiler-generated Java codes, were placed in a subdirectory named after the applet

name, Puzzle2, under jaciec directory. Figure 5.15 also lists all the files generated and copied by the JACIE compiler. These files were then compiled using Sun's JDK and the compiled java classes (together with the corresponding HTML file) were copied to the Web server.

5.5.3 Running JACIE-generated Client and Server Programs

Like any other client/server programs, the server program has to be started first before the client program can communicate with it. The server machine's name (or also known as the

```
Start compiling JACIE Programme...
Applet name: Puzzle2
with text launcher: Click here to display the puzzle
Copying marina2.jpg to Puzzle2\client ...
Copying blank.jpg to Puzzle2\client ...
Copying lim00.jpg to Puzzle2\client ...
Copying lim10.jpg to Puzzle2\client ...
Copying lim20.jpg to Puzzle2\client ...
Copying lim30.jpg to Puzzle2\client ...
Copying lim40.jpg to Puzzle2\client ...
Copying lim50.jpg to Puzzle2\client ...

...

Creating client applet with text launcher Puzzle2.java in Puzzle2\client ...
Creating Puzzle2_Frame.java in Puzzle2\client ...
Creating Puzzle2_Container.java in Puzzle2\client ...
Copying JACIEGrid.java to Puzzle2\client ...
Creating Puzzle2_MenuBar.java in Puzzle2\client ...
Copying Connect.gif to Puzzle2\client ...
Copying Disconnect.gif to Puzzle2\client ...
Copying Users.gif to Puzzle2\client ...
Copying About.gif to Puzzle2\client ...
Copying JACIEImageButton.java to Puzzle2\client ...
Creating Puzzle2_Canvas.java in Puzzle2\client ...
Creating Puzzle2_MessageBar.java in Puzzle2\client ...
Creating Puzzle2_SessionAssistant.java in Puzzle2\client ...
Copying JACIEMessage.java to Puzzle2\client ...
Copying JACIEMessageQueue.java to Puzzle2\client ...
Copying JACIEBox.java to Puzzle2\client ...
Creating Puzzle2_UserList.java in Puzzle2\client ...
Creating HTML file Puzzle2.html in Puzzle2\client ...
Creating server application Puzzle2_Server.java in Puzzle2\server ...
Creating to Puzzle2_Server_SessionManager.java in Puzzle2\server ...
Creating to Puzzle2_Server_SessionAssistant.java in Puzzle2\server ...
Creating to Puzzle2_Server_DeliveryAssistant.java in Puzzle2\server ...
Copying JACIEMessage.java to Puzzle2\server ...
Copying JACIEMessageQueue.java to Puzzle2\server ...

End of compilation
No errors. Parsing took 10s 380ms
```

Figure 5.15 Output of compiling Puzzle2.jacie program

host), on which the server program will be running, has to be the one specified by the `host` statement in the `configuration` construct. If the `host` name is specified as `prompt`, the host name should be known by the communicating clients by other means to enable the clients to connect to this server. This is because the client's user interface does not specify the name of this host.

The server program is run using the Java interpreter as follows

```
java filename_server (if port number is specified in the  
configuration construct)
```

or with port number argument,

```
java filename_server portnumber (if port number specified in the  
configuration construct is prompt)
```

As mentioned in the earlier chapters, the port number is the socket through which the client-server communication services will take place. Once the server process is running, it will stay waiting and listening to the socket for remote clients to make connection requests.

There are two ways to invoke the client program. These depend on the type of the client program. If the client program is an application, the copied client codes can be run using the Java interpreter as follows:

```
java filename
```

The standard JACIE frame will be displayed and by providing all the required connection information (`host`, `port` and `username`) followed by clicking on the connect button, the client-server connection can be established.


```
C:\jacie\Puzzle2>java Puzzle2_Server
Server name: Puzzle2_Server
Server process started on port 3333
Session Manager process started
```

Figure 5.16 *The running process of Puzzle2_Server*

If the client program is an applet, the HTML file is loaded by specifying the URL of the HTML file in the Java-enabled Web browser's location or address field. Alternatively, an `appletviewer` command (provided in the JDK) can also be used as follows:

```
appletviewer URL
```

Figure 5.16 shows the waiting server process of `Puzzle2_Server` running on HTTP server named `asahipc2` on port `3333`. Figure 5.17 shows the screen shot of the running client program within a Web browser. More examples will be shown in Chapter 6.

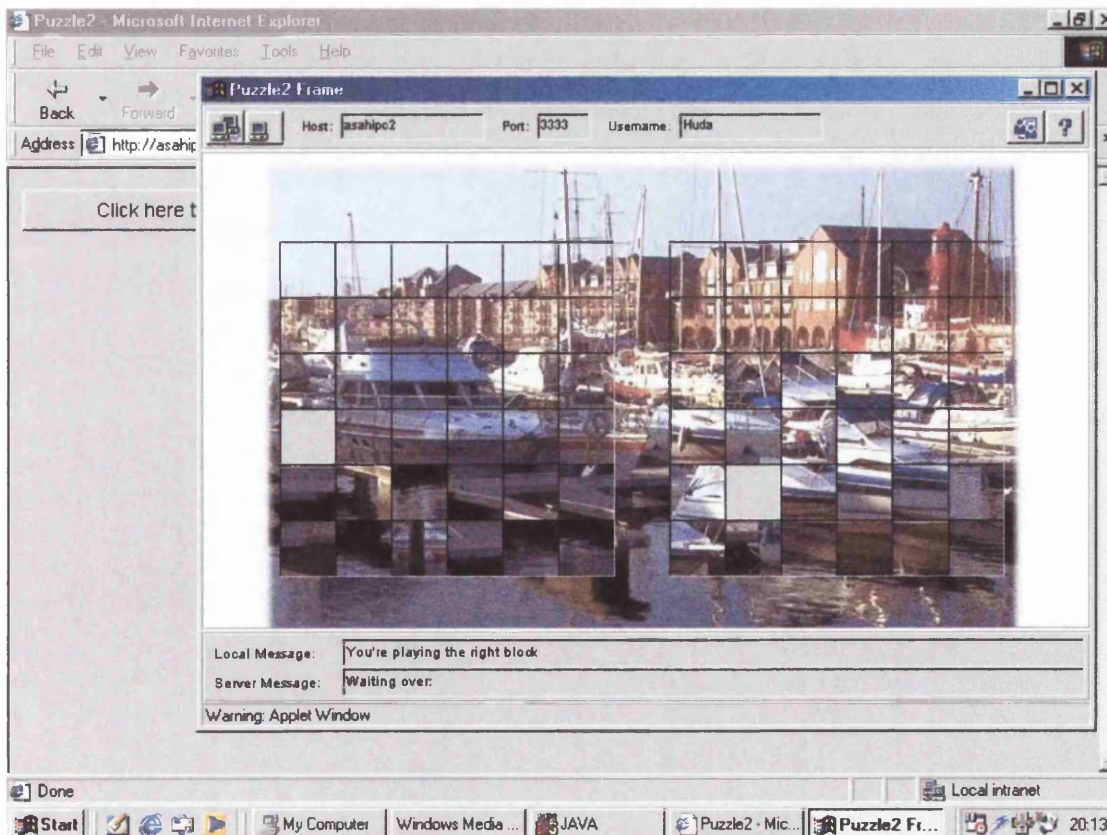


Figure 5.17 *The Puzzle2 applet running in Microsoft Internet Explorer*

5.6 Summary

About 80 percent of JACIE language features have been implemented in this compiler prototype. Several programs written in JACIE scripts have been successfully translated to Java codes using this compiler. Compilation time was remarkably fast. The generated Java codes could be easily comprehended by a Java programmer with some experience in network programming. Compilation of these Java codes was smooth and error-free. The client programs as well as the server programs were executed correctly and interactions between them took place accordingly.

The JACIE compiler verifies that the JACIE scripting language can be a viable tool for developing interactive and collaborative applications. More importantly a rapid prototyping of net-centric, multimedia and collaborative applications is now made possible.

Chapter 6

Example Applications

Several interactive and collaborative applications have been written in JACIE to prove the concept and to demonstrate the usability and functionality of the language and the compiler. They also test the compiler and the underlying component classes of the server and the client programs. The scripts were compiled with the implemented JACIE compiler which produced Java programs for the respective clients and servers. These Java programs were compiled with Sun's JDK1.1.8, deployed and executed on various client and server machines over the department's local area network (LAN). For the applets which were embedded in Web pages, Microsoft Internet Explorer has been used as the Web browser. Technically, any Web browser can also be used as long as it supports the Java Virtual Machine (JVM).

This chapter reports the development and the execution of three example applications implemented in JACIE. They were chosen on the basis of their distinctive interactive modes and their diverse feature requirements. For one simple application, comparisons with Java codes will be made to highlight the statements used (in Java and JACIE) and the programming steps required. At the same time the versatility and the advantages of JACIE over Java will be discussed and demonstrated. The three example applications are:

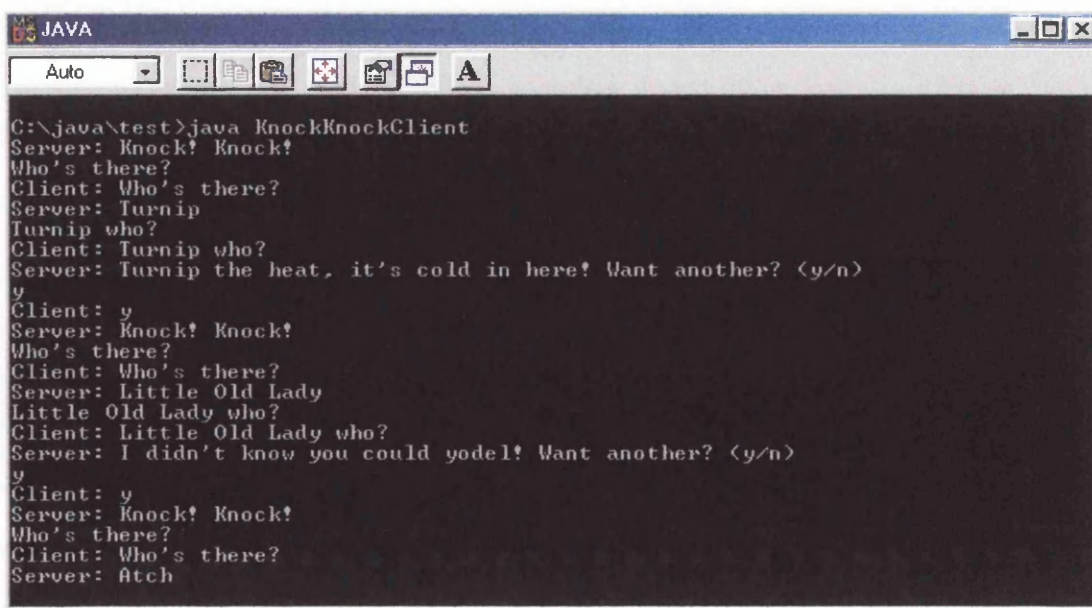
- 1) networked knock knock jokes,
- 2) multi-user network troubleshooting,
- 3) collaborative scrabble.

6.1 A Server-based Interactive Application: Networked Knock Knock Jokes

Knock knock jokes are the all-time brainteaser for all ages. The client/server implementation of these jokes is a simple example of server-based interactive applications. The application can be designed in such a way where the server does the “knocking” and serves up the jokes and the client does the “questioning”. A sample knock knock jokes could be expressed in the following format:

```
Server : Knock! Knock!  
Client : Who's there?  
Server : Atch  
Client : Atch who?  
Server : Bless you!
```

For the purpose of comparison, the implementation of this application in Java as written by Campione and Walrath [23] has been used for adaptation in JACIE. This simple example illustrates how sockets are created and dealt in Java and JACIE. It also demonstrates how reading and writing are carried out between a client program and a server program.



```
C:\java\test>java KnockKnockClient  
Server: Knock! Knock!  
Who's there?  
Client: Who's there?  
Server: Turnip  
Turnip who?  
Client: Turnip who?  
Server: Turnip the heat, it's cold in here! Want another? (y/n)  
y  
Client: y  
Server: Knock! Knock!  
Who's there?  
Client: Who's there?  
Server: Little Old Lady  
Little Old Lady who?  
Client: Little Old Lady who?  
Server: I didn't know you could yodel! Want another? (y/n)  
y  
Client: y  
Server: Knock! Knock!  
Who's there?  
Client: Who's there?  
Server: Atch
```

Figure 6.1 The running KnockKnock client program as implemented by Campione and Walrath

Figure 6.1 shows the screen shot of the running client program of the `KnockKnock` application as implemented by Campione and Walrath. It also shows the interaction between the server and the client during the “knocking” and the “questioning” session.

6.1.1 Java Implementation

In Java (as implemented by Campione and Walrath) this application which supports multiple client requests consists of four classes – the server program implemented by three classes, `KKMultiServer`, `KKMultiServerThread` and `KnockKnockProtocol`, and the client program implemented by a single class, `KnockKnockClient`.

Program 6.1a is the Java code for `KKMultiServer` class, the first class for the server program. As shown, two Java packages are referred to: `java.net` and `java.io`. The `java.net` package provides the `ServerSocket` class which implements a socket that servers can use to listen for and accept connections to clients. The `java.io` package provides the facility for reading and writing to and from the socket.

Program 6.1a *Java implementation of Knock Knock Jokes*

```
import java.net.*;
import java.io.*;

public class KKMultiServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = null;
        boolean listening = true;

        try {
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Could not listen on port: 4444.");
            System.exit(-1);
        }

        while (listening)
            new KKMultiServerThread(serverSocket.accept()).start();

        serverSocket.close();
    }
}
```

As shown, the code uses an *exception* for error-handling capability. `KKMultiServer` loops forever, listening for client connection requests on a `ServerSocket`. Each client's request is queued at port 4444 and consequently come into the same `ServerSocket`. For each request `KKMultiServer` accepts the connection, creates a new `KKMultiServerThread` object to process it, hands it the socket returned by the method `accept` and starts the new thread.

Program 6.1b shows the code for `KKMultiServerThread`, the second class for the server program. The `KKMultiServerThread` extends `Threads` because for each client an instance of this class will be created. The independent `KKMultiServerThread` object communicates to the client by reading from and writing to the socket. This program requires the use of exception handling.

Program 6.1b *Java implementation of Knock Knock Jokes (cont)*

```
import java.net.*;
import java.io.*;

public class KKMultiServerThread extends Thread {
    private Socket socket = null;

    public KKMultiServerThread(Socket socket) {
        super("KKMultiServerThread");
        this.socket = socket;
    }

    public void run() {
        try {
            PrintWriter out = new PrintWriter(socket.getOutputStream(),true);
            BufferedReader in = new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
            String inputLine, outputLine;
            KnockKnockProtocol kkp = new KnockKnockProtocol();

            outputLine = kkp.processInput(null);
            out.println(outputLine);

            while ((inputLine = in.readLine()) != null) {
                outputLine = kkp.processInput(inputLine);
                out.println(outputLine);
                if (outputLine.equals("Bye"))
                    break;
            }
            out.close();
            in.close();
            socket.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The last class for the server program of Knock Knock Jokes implemented by Campione and Walrath is `KnockKnockProtocol`. As shown in Program 6.1c, the program gets the socket's input and output stream and opens readers and writers on them. A `KnockKnockProtocol` object is created with which the current joke and the current state within the joke are tracked. Interactions between the client and the server occur as long as the client and the server still have something to say to each other.

Program 6.1d is the Java class for the client program of Knock Knock Jokes. The instances of `KnockKnockClient` speak to the `KKMultiServer` mentioned above. The program needs to open a socket that is connected to the server running on a specific hostname and a specific port. In the given example, the hostname is "taranis" and the port number to which `KKMultiServer` is listening to is 4444. Since the server speaks first, the client must listen first. The client does this by reading from the input stream attached to the socket. It then displays the text to the standard output and waits for a response from the user who needs to type into the standard input. After the user has typed a carriage return, the client sends the text to the server through the output stream attached to the socket. The interaction continues until the server says "Bye."

As can be seen, four Java classes are defined to achieve a manageable Java coding for the implementation of Knock Knock Jokes application. Some of the Java features used are different network sockets, various input/output streams, exception handling, multiple threads and various modifiers in class and member variable declarations.

Program 6.1c Java implementation of Knock Knock Jokes (cont)

```
import java.net.*;
import java.io.*;

public class KnockKnockProtocol {
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;

    private static final int NUMJOKES = 5;
    private int state = WAITING;
    private int currentJoke = 0;

    private String[] clues = { "Turnip", "Little Old Lady", "Atch", "Who", "Who" };

    private String[] answers = { "Turnip the heat, it's cold in here!",
        "I didn't know you could yodel!",
        "Bless you!",
        "Is there an owl in here?",
        "Is there an echo in here?" };

    public String processInput(String theInput) {
        String theOutput = null;

        if (state == WAITING) {
            theOutput = "Knock! Knock!";
            state = SENTKNOCKKNOCK;
        } else if (state == SENTKNOCKKNOCK) {
            if (theInput.equalsIgnoreCase("Who's there?")) {
                theOutput = clues[currentJoke];
                state = SENTCLUE;
            } else {
                theOutput = "You're supposed to say \"Who's there?!\" " +
                    "Try again. Knock! Knock!";
            }
        } else if (state == SENTCLUE) {
            if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {
                theOutput = answers[currentJoke] + " Want another? (y/n)";
                state = ANOTHER;
            } else {
                theOutput = "You're supposed to say \"" +
                    clues[currentJoke] +
                    " who?\" " +
                    "! Try again. Knock! Knock!";
                state = SENTKNOCKKNOCK;
            }
        } else if (state == ANOTHER) {
            if (theInput.equalsIgnoreCase("y")) {
                theOutput = "Knock! Knock!";
                if (currentJoke == (NUMJOKES - 1))
                    currentJoke = 0;
                else
                    currentJoke++;
                state = SENTKNOCKKNOCK;
            } else {
                theOutput = "Bye.";
                state = WAITING;
            }
        }
        return theOutput;
    }
}
```


Program 6.1d Java implementation of Knock Knock Jokes (cont)

```
import java.io.*;
import java.net.*;

public class KnockKnockClient {
    public static void main(String[] args) throws IOException {
        Socket kkSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            kkSocket = new Socket("taranis", 4444);
            out = new PrintWriter(kkSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(kkSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: taranis.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: taranis.");
            System.exit(1);
        }

        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
        String fromServer;
        String fromUser;

        while ((fromServer = in.readLine()) != null) {
            System.out.println("Server: " + fromServer);
            if (fromServer.equals("Bye."))
                break;
            fromUser = stdIn.readLine();
            if (fromUser != null) {
                System.out.println("Client: " + fromUser);
                out.println(fromUser);
            }
        }
        out.close();
        in.close();
        stdIn.close();
        kkSocket.close();
    }
}
```

6.1.2 JACIE Implementation

The whole programming steps for this application have been simplified significantly in JACIE. The four Java classes above have been reduced to one JACIE script with less code and less hassle.

Program 6.2a JACIE implementation of Knock Knock Jokes

```
JACIE {
  application name KnockKnock;
  configuration {
    host "taranis";
    port 4444;
    username prompt;
    about "Knock Knock Jokes adapted from The Java Tutorial and rewritten in JACIE";
  }
  messages {
    ClientMessage, ServerMessage
  }
  ...
}
```

The JACIE's implementation of the application starts with system configuration statements. Program 6.2a shows the related statements.

All JACIE programs start with a keyword `JACIE` followed by an open brace “{” and ends with closed brace “}”. The statement

```
application name KnockKnock;
```

simply means that this is an application (not an applet) named `KnockKnock`.

Under the `configuration` construct the statements

```
host "taranis";
port 4444;
username prompt;
about "Knock Knock Jokes adapted from The Java Tutorial and rewritten in JACIE";
```

specify the host where the server program will reside (in this case, “taranis”), the port number where the communication service will take place (in this case, 4444) and the username of the client. Syntactically, the `username` statement is optional. By design, JACIE-created programs require the client's username to be specified before connection can be established. The statement `username prompt` means that the username will be provided at runtime. Alternatively, since this application is meant for a single-user interaction (even though the server can handle multiple clients concurrently) and a unique username is unnecessary, a generic username can be used, e.g. `username "KnockKnockClient";`.

Program 6.2b JACIE implementation of Knock Knock Jokes (cont)

```
client implementation {
  declaration {
    string fromServer;
    string fromUser;
  }
  on canvas { }
  on session start {
    receive ServerMessage fromServer;
  }
  on session {
    if (fromServer == "Bye.")
      abort;
    print servermessage "Server: "+fromServer;
    input fromUser;
    print "Client: "+fromUser;
    send ClientMessage fromUser;
    receive ServerMessage fromServer;
  }
  on session end { }
}
```

Similarly, the `about` statement is also optional but if specified an “about” button will be generated on the client user interface from which the about message will be displayed when clicking on it.

The `messages` construct consists of `ClientMessage`, `ServerMessage` which refers to message identifiers that will be used during the message passing between the client and the server. The `ClientMessage` is the identifier for the messages sent by the client to the server and the `ServerMessage` is the identifier for the messages sent by the server to the client.

JACIE’s configuration and `messages` constructs are followed by a `client implementation` construct. The related statements are shown in Program 6.2b.

The `client implementation` starts with a declaration of variables and methods used by the client program segment. In this case no method is involved.

As shown, the two variables involved are `fromServer` and `fromUser`. As their names imply, the `fromServer` maintains the messages sent by the server (in this case the “Knock knock” and the “jokes”) and the `fromUser` maintains the messages sent by the client (in this case the “questions”).

Since no graphic canvas is involved for this application, the `on canvas` construct contains no statement.

The next construct, `on session start`, contains the following statement:

```
receive ServerMessage fromServer;
```

Basically the statement means that the first action to take on established connection receives a `fromServer` message which is a type `ServerMessage`. This `fromServer` message is used in the next phase of interactivities.

The `on session` construct that follows contains statements for the main interactivities between the client program and the server program. By design this language construct is a loop that will exit when the user clicks the disconnect button or will disconnect naturally through certain statements. The statements for this construct are shown below:

```
if (fromServer == "Bye.")
  abort;
print servermessage "Server: "+fromServer;
input fromUser;
print "Client: "+fromUser;
send ClientMessage fromUser;
receive ServerMessage fromServer;
```

In this case, the loop exits naturally when the server sends a `ServerMessage` with `fromServer` text value "Bye.". Otherwise, the client displays the text to the JACIE's standard `servermessage` output area and then reads the response (the `fromUser`) from the user who types into the JACIE's standard input area. After the user types a carriage return, the `fromUser` text is sent to the server as `ClientMessage`.

The final construct of the `client` implementation is `on session end`. In our example, no statement is required in this construct.

The client implementation is followed by a server implementation construct. Program 6.2c and Program 6.2d show the statements in this construct.

Program 6.2c JACIE implementation of Knock Knock Jokes (cont)

```

server implementation {
  declaration {
    string inputLine;
    string outputLine;
    int WAITINGSTATE = 0;
    int SENTKNOCKKNOCK = 1;
    int SENTCLUE = 2;
    int ANOTHER = 3;
    int NUMJOKES = 5;
    int state = WAITINGSTATE;
    int currentJoke = 0;
    string[5] clues = {"Turnip", "Little Old Lady", "Atch", "Who", "Who"};
    string[5] answers = {"Turnip the heat, it's cold in here!",
                        "I didn't know you could yodel!",
                        "Bless you!",
                        "Is there an owl in here?",
                        "Is there an echo in here?"};

    string processInput(string theInput) {
      string theOutput = null;
      if (state == WAITINGSTATE) {
        theOutput = "Knock! Knock!";
        state = SENTKNOCKKNOCK;
      } else if (state == SENTKNOCKKNOCK) {
        if (theInput=="Who's there?") {
          theOutput = clues[currentJoke];
          state = SENTCLUE;
        } else {
          theOutput = "You're supposed to say \"Who's there?! \" +
                    "Try again. Knock! Knock!";
        }
      } else if (state == SENTCLUE) {
        if (theInput == (clues[currentJoke] + " who?")) {
          theOutput = answers[currentJoke] + " Want another? (y/n)";
          state = ANOTHER;
        } else {
          theOutput = "You're supposed to say \"" + clues[currentJoke] +
                    " who?\" + \"! Try again. Knock! Knock!";
          state = SENTKNOCKKNOCK;
        }
      } else if (state == ANOTHER) {
        if (theInput == "y") {
          theOutput = "Knock! Knock!";
          if (currentJoke == (NUMJOKES - 1))
            currentJoke = 0;
          else
            currentJoke = currentJoke+1;
          state = SENTKNOCKKNOCK;
        } else {
          theOutput = "Bye.";
          state = WAITINGSTATE;
        }
      }
      return theOutput;
    }
  }
}

```

Program 6.2d JACIE implementation of Knock Knock Jokes (cont)

```
on server start { }
on session start {
  outputLine = processInput(null);
  send ServerMessage outputLine;
}
on session {
  receive ClientMessage inputLine;
  if (inputLine==null)
    abort;
  outputLine = processInput(inputLine);
  send ServerMessage outputLine;
  if (outputLine=="Bye.")
    abort;
}
on session end { }
on server end { }
}
```

Similar to the client implementation, the server implementation starts with a declaration of variables and methods used by the server program segment. In this case the variables and the methods used are very much the same as implemented by `KnockKnockProtocol` described before. Since the `KnockKnockProtocol` is a class (or in JACIE's terms a method which is a procedure) that provides the server's response to a client's questions, not much simplification can be done in JACIE. As seen, the method declaration in JACIE is similar to Java's. The code above also illustrates some minor difference between JACIE and Java in terms of array declaration. For JACIE the size of the arrays (in the example `clues` and `answers`) have to be explicitly specified.

The next construct of the server implementation is `on server start`. No statement is in use for this application.

This is followed by `on session start` construct. The statements for this construct are shown below:

```
outputLine = processInput(null);
send ServerMessage outputLine;
```

As expected, statements within the `on session start` of the server implementation work in pair with the statements within the `on session start` of the client

implementation. In this example application, `outputLine` has a value returned by the method `processInput(null)` which is "Knock! Knock!". This text string will be sent to the client as `ServerMessage`. Referring to the statement within the `on session start` of the client implementation,

```
receive ServerMessage fromServer;
```

on arrival of the `ServerMessage`, the `fromServer` gets the value of the text string which will be displayed by the `print` statement.

The next construct of the server implementation is `on session`. Similar to the one in the client implementation, by design it is a loop, and it exits when the user or the server issues disconnection or abort:

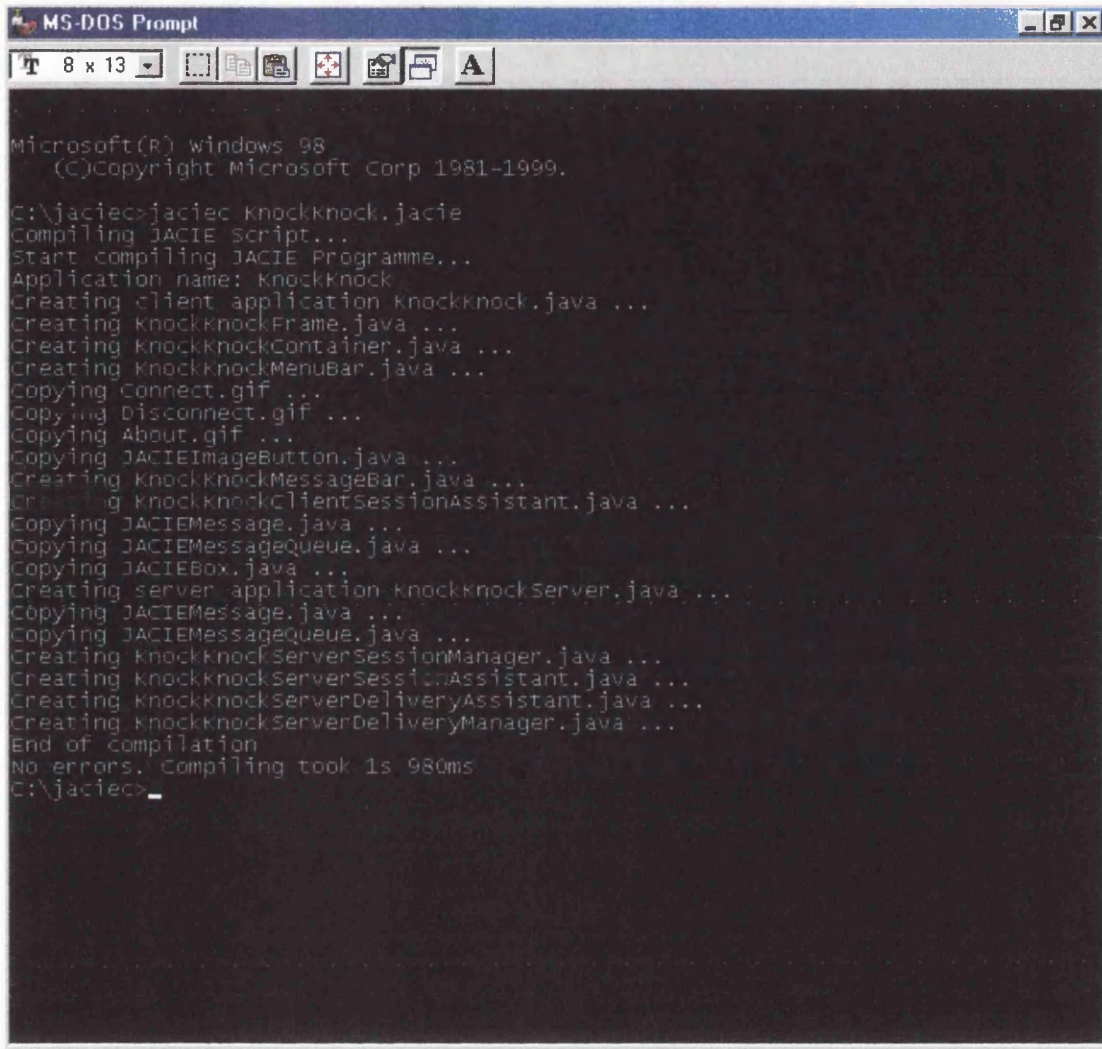
```
receive ClientMessage inputLine;  
if (inputLine==null)  
    abort;  
outputLine = processInput(inputLine);  
send ServerMessage outputLine;  
if (outputLine=="Bye.")  
    abort;
```

As shown, the communication statements (`receive` and `send`) within this construct work in tandem with the corresponding construct of the client implementation.

The `KnockKnock` application does not require any statement within the last two constructs: the `on session end` and the `on server end`. This completes the JACIE script for the application.

6.1.3 Compilation and Execution

Figure 6.2 shows the output of compiling the `KnockKnock.jacie` script. As can be seen, on compiling this script with the JACIE compiler, Java classes for the client program and the server program were generated. These Java classes were then compiled, deployed and



```
Microsoft(R) Windows 98
(C)Copyright Microsoft Corp 1981-1999.

C:\jaciec>jaciec knockknock.jacie
Compiling JACIE script...
Start compiling JACIE Programme...
Application name: KnockKnock
Creating client application KnockKnock.java ...
Creating KnockKnockFrame.java ...
Creating KnockKnockContainer.java ...
Creating KnockKnockMenuBar.java ...
Copying Connect.gif ...
Copying Disconnect.gif ...
Copying About.gif ...
Copying JACIEImageButton.java ...
Creating KnockKnockMessageBar.java ...
Creating KnockKnockClientSessionAssistant.java ...
Copying JACIEMessage.java ...
Copying JACIEMessageQueue.java ...
Copying JACIEBox.java ...
Creating server application KnockKnockServer.java ...
Copying JACIEMessage.java ...
Copying JACIEMessageQueue.java ...
Creating KnockKnockServerSessionManager.java ...
Creating KnockKnockServerSessionAssistant.java ...
Creating KnockKnockServerDeliveryAssistant.java ...
Creating KnockKnockServerDeliveryManager.java ...
End of compilation
No errors. Compiling took 1s 980ms
C:\jaciec>
```

Figure 6.2 *The output of compiling the KnockKnock.jacie script*

executed. The generated Java codes are not identical to the ones implemented by Campione and Walrath, as mentioned above, but they are of similar functionalities.

Figure 6.3 shows the `KnockKnockServer` running on “asahipc2” (the hostname “taranis” has been changed to “asahipc2” to correspond to the machine on the LAN). Figure 6.4, on the other hand, shows a series of screen shot of the running client program of the `KnockKnock` application. The major difference is in the JACIE’s standard user interface (compare this figure with Figure 6.1).

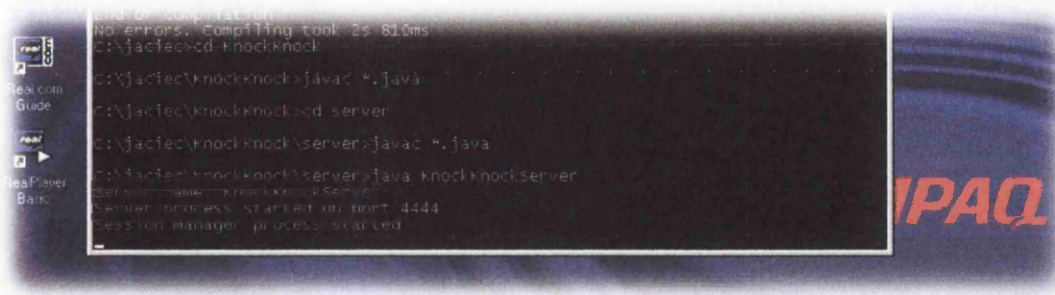


Figure 6.3 The KnockKnockServer program running on asahipc2

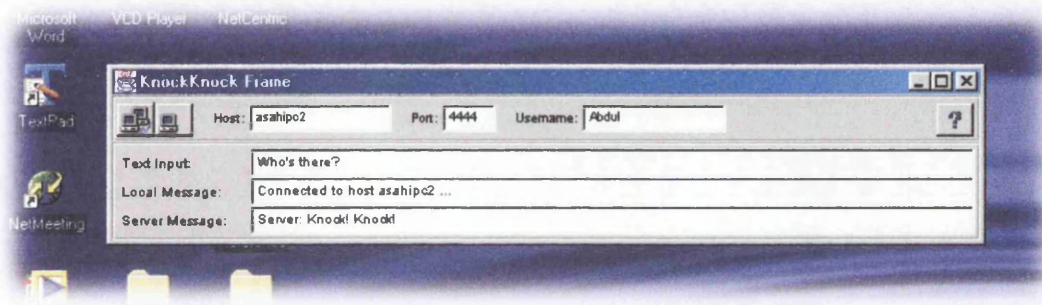


Figure 6.4 The running KnockKnock client program

6.1.4 Summary

Java has rich features for developing network applications. This “simple” application has demonstrated some typical Java features required in the implementation of any server-based interactive applications. Some of the features used are network socket communication handling, various types of input/output streams, exception handling, multi-threading operations and various modifiers in class and member variable declarations. With exception to advanced system programmers, these “rich” features are not within the reach of the rest of the programmers’ community.

JACIE's implementation shows how these features can be tackled sensibly. With comparable indentation, the four Java classes with 161 lines of code have been reduced to one JACIE script with 106 lines of code. In terms of number of words, about 30 percent less code have been achieved. If we disregard the class or the function that returns the different response to different user's input (`KnockKnockProtocol` for Java and `processInput` for JACIE) where nothing much can be simplified, JACIE boasts 50 percent less code. But the obvious absence of Java's strong-but-complex features demonstrated further the simplicity of the JACIE language.

6.2 A Server-mediated Interactive Application: Multi-user Network Troubleshooting

The second example application is a multi-user network troubleshooting (referred to as NTS). It is designed as a piece of teamwork courseware with which multiple concurrent users (in this case remote students) can engage in a collaborative activity of diagnosing simulated network problems. This type of application can be used to support teaching and learning of basic network troubleshooting.

Students are given the description of a problem in a set of interconnected networks. Each student is given control of one network and, thus, can only manipulate network devices under his control (e.g., examining individual device status, switching the device on or off, connecting or disconnecting the cable links, issuing "ping" command from any host within his control to local or remote hosts and performing software configuration). As no single user has the overall control of all networks, collaboration among participants is essential for successful troubleshooting. Some kind of communication channel is required.

Such a collaborative application is best served by JACIE. A canvas channel can be used for displaying a collection of interconnected networks – both for a global view and a local view. The global view is meant for displaying the overall topology of the networks, whereas the local view is where the individual student interacts and manipulates the devices under his control. A chat channel can be employed to facilitate communication

among participants. Due to the JACIE's current implementation of the video channel which is based on NetMeeting videoconferencing facility, the video channel may also be employed only if two students are involved.

Figure 6.5 shows the screen capture of the NTS. As displayed, the canvas is in the global view with all the network devices and their interconnections are shown. The three main buttons on top of the network layout are GLOBAL VIEW, LOCAL VIEW and PROBLEM. They are used to toggle from one view to another and to display the current troubleshooting problem.

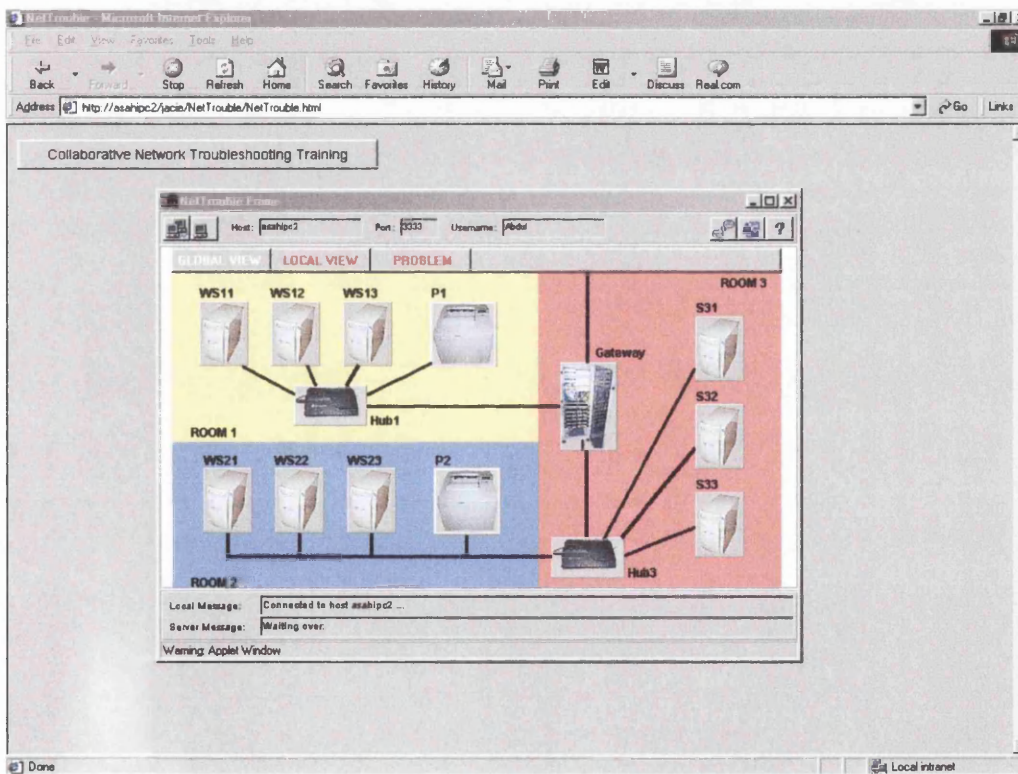


Figure 6.5 The global view of Network Troubleshooting course

Program 6.3a JACIE implementation of Network Troubleshooting

```
JACIE {
  applet name NetTrouble;
  appletlauncher text "Collaborative Network Troubleshooting Training";
  configuration {
    host "asahipc2";
    port 3333;
    username prompt;
    channel canvas, chat;
    about file "NetTrouble.txt";
    number of users 3;
    protocol contention;
  }
  messages {
    roomAssigned, problemStatement, checkDevice, userRequest,
    deviceConfig, problemSolved, exitProgram
  }
  ...
}
}
```

6.2.1 Implementation

The NTS application has been implemented as a JACIE script. It is meant for three-student collaboration. The NTS employs a *contention* protocol for its user-user interactions. Remote users (clients) can send messages to the server at any time without the need of turn control. The system configuration component of the JACIE script has the statements shown in Program 6.3a.

Unlike the first example application, the NTS is an applet. The following first two statements are related to this.

```
applet name NetTrouble;
appletlauncher text "Collaborative Network Troubleshooting Training";
```

The first statement specifies the client program type (which is an applet) and the program name (which is `NetTrouble`). The second statement simply means that a text button will be embedded in a Web page from which a separate applet window will be opened on activation. Within the `configuration` construct the `host` and `port` statements specify the host name where the server program will reside and the port number where the

communication service will take place. Since this is a multi-user application, the username of each participant has to be unique. The statement

```
username prompt;
```

means that the username will be supplied at runtime and by default the system will impose this constraint.

Also, unlike the first example application, the NTS employs two communication channels – `canvas` and `chat`. This is reflected in its statement:

```
channel canvas, chat;
```

The canvas is where all simulated network devices are drawn and user-system interaction for checking device status and configuration takes place. The user interaction through the canvas medium is in addition to JACIE's built-in text input/output message channel. Each user has his or her own network devices under his or her control. Thus the local canvas is designed particularly for each user but since the networks are interconnected, the connection/disconnection and configuration/reconfiguration of devices under his or her control affect the whole networks. A chat channel is therefore used for user-user communication. A user can convey his or her diagnostic report to one another or can suggest to other users some possible steps to be taken. By doing so, they collaboratively diagnose and locate the malfunctioning part.

The next statement:

```
about file "NetTrouble.txt";
```

basically generates an “about” button but, unlike the first application, at runtime clicking this button opens a dialog box that displays text from “`NetTrouble.txt`” file. This file is a help file which contains, among others, the syntaxes of commands for interaction with the system.

The following statements are another new features differing from the first application:

```
number of users 3;  
protocol contention;
```

Since this is a multi-user server-mediated interactive application, the JACIE language requires specifications for the number of concurrent users allowed and the user interaction protocol (or floor management protocol) to be employed. This application is designed for three users and by employing a *contention* protocol, its participants can send messages to the server at any time without the need of turn control.

There are a number of message identifiers used in the NTS. Some of them are as follows:

```
messages {  
    roomAssigned, problemStatement, checkDevice, userRequest,  
    deviceConfig, problemSolved, exitProgram  
    ...  
}
```

The `roomAssigned` message will be initiated by the server program and will be sent to the client program to notify which room the user is assigned to. The client program will then limit the user's local view as well as control network devices in this specific room. The `problemStatement` message is also initiated by the server to alert all the active users of the “network problem”. This message is to be sent at the start of each troubleshooting session. The user may also enquire about the “problem” at a later stage. During an active session, other message transfers (each one with its own unique message identifier) are also involved which, among others, are to check configuration (denoted by `checkDevice`), to reconfigure (denoted by `userRequest` and `deviceConfig`), etc.

The client body of the NTS consists of several program constructs. Like any other JACIE script, this client implementation component begins with a declaration construct where all the variables and methods used within the component are declared. An example extracted from the NTS application is given in Program 6.3b.

Program 6.3b JACIE implementation of Network Troubleshooting (cont)

```

client implementation {
  declaration {
    int roomNumber;
    string problem;
    int currentView = 1;      // 1 = Global, 2 = Local
    string instruction;
    string deviceName;
    string config;
    image globalViewLayout = "Layout.gif";
    image globalViewOn = "GlobalOn.gif";
    image globalViewOff = "GlobalOff.gif";
    image localViewOn = "LocalOn.gif";
    image localViewOff = "LocalOff.gif";
    image problemIcon = "ProblemIcon.gif";
    ...
  }
  string extractWord(string theText, int pos) {
    // this method parses theText and returns the word at pos position
    string word = "";
    int firstSep;
    int secondSep;
    int numberOfWords;
    if (theText==null || theText=="")
      numberOfWords = 0;
    else if (Java_theText.indexOf(" ")== -1)
      numberOfWords = 1;
    else if (Java_theText.indexOf(" ")==Java_theText.lastIndexOf(" "))
      numberOfWords = 2;
    else numberOfWords = 3;
    if (pos <= numberOfWords) {
      firstSep = Java_theText.indexOf(" ");
      secondSep = Java_theText.lastIndexOf(" ");
      if (pos==1) {
        if (numberOfWords==1)
          word = theText;
        else
          word = Java_theText.substring(0,firstSep);
      }
      else if (pos==2) {
        if (numberOfWords==2)
          word = Java_theText.substring(firstSep+1);
        else
          word = Java_theText.substring(firstSep+1,secondSep);
      }
    }
    ...
  }
  return word;
}

string deviceType(string deviceName) {
  // this method returns the device type denoted by deviceName
  if (deviceName=="WS11" || deviceName=="WS12" || deviceName=="WS13"
      || deviceName=="WS21" || deviceName=="WS22" || deviceName=="WS23")
    return "workstation";
  else if (deviceName=="S31" || deviceName=="S32" || deviceName=="S33")
    return "server";
  else if (deviceName=="P1" || deviceName=="P2")
    return "printer";
  else if (deviceName=="Hub1" || deviceName=="Hub3")
    return "hub";
  else if (deviceName=="Gateway")
    return "gateway";
  ...
  else
    return "unknown";
}
}
}

```

While the JACIE's variable declaration format for numeric and string data types is similar to many other programming languages, it differs greatly in terms of image declarations. JACIE allows assigning a string literal to an image variable. These string literals represent filenames of graphic images.

Some of the examples extracted from Program 6.3b are as follows:

```
image globalViewLayout = "Layout.gif";
image globalViewOn = "GlobalOn.gif";
image globalViewOff = "GlobalOff.gif";
```

Program 6.3b shows two methods declared for client implementation, namely `extractWord` and `deviceType`. The `extractWord` method simply parses the given text string and returns the word at the specified position. The method, utilised in conjunction with the instruction typed in by the user, identifies the syntactically correct instruction used. The method also shows how Java features are integrated in this JACIE script (as denoted by the prefix `Java_`). This is due to the fact that JACIE has no built-in facility for string manipulation. Some of Java's string accessor methods used in the example are `indexOf()`, `lastIndexOf()` and `substring()`.

The second method, `deviceType`, returns the type of the network device. The method is used on the client-side to ensure the right instruction is used on the right device before this instruction is sent to the server for further information or configuration. The method also shows a simple string comparison in JACIE, as opposed to Java, e.g.,

```
if (deviceName=="S31" || deviceName=="S32" || deviceName=="S33")
    return "server";
else if (deviceName=="P1" || deviceName=="P2")
    return "printer";
```

The recognised command instructions to be used by the communicating NTS users are shown in Table 6.1. Users can also refer to the syntax of these commands from the program's "about" facility. Except for getting the information, the current implementation of NTS does not allow the gateway to be reconfigured.

For servers, workstations or printers,	
switch on	to turn on the device
switch off	to turn off the device
change IPAddr x.x.x.x	To change the IP address of the device where x.x.x.x is the new IP address
change NetMask x.x.x.x	To change the netmask of the device where x.x.x.x is the new netmask
change DefaultGateway x.x.x.x	To change the default gateway of the device where x.x.x.x is the new default gateway
For servers or workstations,	
ping <DeviceName>	where <DeviceName> is the remote device name to ping (to test reachability)
For hubs,	
switch on	to turn on the hub
switch off	to turn off the hub
For cables,	
connect cable	to attach the cable to the network
disconnect cable	to detach the cable from the network

Table 6.1 *The NTS command instructions for network configurations*

The following declaration is the `on canvas` construct. This construct specifies the default workspace canvas on which user-defined interactions and output displays will take place. Some statements in this construct are shown in Program 6.3c.

Program 6.3c *JACIE implementation of Network Troubleshooting (cont)*

```

on canvas {
  foreground gray;
  define canvas globalView {
    draw grid ViewIcons at 10,10 step 100,25 size 3,1;
    draw image grid ViewIcons globalViewOn at 0,0;
    draw image grid ViewIcons localViewOff at 1,0;
    draw image grid ViewIcons problemIcon at 2,0;
    draw image globalViewLayout at 10,35;
    ...
  }
  define canvas localView1 {
    draw grid ViewIcons1 at 10,10 step 100,25 size 3,1;
    draw image grid ViewIcons1 globalViewOff at 0,0;
    draw image grid ViewIcons1 localViewOn at 1,0;
    draw image grid ViewIcons1 problemIcon at 2,0;
    // draw detail objects in room 1
    ...
  }
  define canvas localView2 {
    // draw ViewIcons2 grid
    // draw detail objects in room 2
    ...
  }
  define canvas localView3 {
    // draw ViewIcons3 grid
    // draw detail objects in room 3
    ...
  }
  use canvas globalView;
}

```

As shown, four canvases are defined, namely `globalView`, `localView1`, `localView2` and `localView3`. (Since the third example application will depend heavily on graphic features of JACIE, the graphic features explained in this section will not be stressed).

As can be seen, most of the statements used in `define canvas` are graphics statement (e.g. `draw grid`, `draw image`, etc.).

The statement

```
use canvas globalView;
```

within the `on canvas` construct specifies that the default canvas is the `globalView`. Other canvases will be used at a later stage and also depend on the assigned `roomNumber`. With this set of statements, an initial global view is drawn onto the canvas (as shown in Figure 6.5).

The next construct (as shown in Program 6.3d) is the `on session start`. Typical of any client-side interactive multi-user program, it has an event control statement on `WAITING` to handle any action if the required number of online users has not been reached. In this case, it will only print a message indicating that the system is waiting for more remote users.

Program 6.3d *JACIE implementation of Network Troubleshooting (cont)*

```
on session start {
  on WAITING {
    print "Waiting for other remote users";
  }
  print "Now we have enough people to perform network troubleshooting - enjoy!";
  wait 5;
  print "Open chat channel for interaction with other players";
  wait 5;
  use canvas globalView;
  receive roomAssigned roomNumber;
  print "You've been assigned room number "+roomNumber;
  receive problemStatement problem;
  deviceName = "notSelected";
  define canvas localView {
    if (roomNumber==1)
      use canvas localView1;
    else if (roomNumber==2)
      use canvas localView2;
    else
      use canvas localView3;
  }
}
```

```
on WAITING {  
  print "Waiting for other remote users";  
}
```

As soon as the alert from the server is received (behind the scene), notifying that enough users are online, the statements following the `on WAITING` will be executed:

```
print "Now we have enough people to perform network troubleshooting - enjoy!";  
wait 5;  
print "Open chat channel for interaction with other players";  
wait 5;  
use canvas globalView;  
receive roomAssigned roomNumber;  
print "You've been assigned room number "+roomNumber;  
receive problemStatement problem;
```

It will print a message, wait for 5 seconds and print another message. The default graphic canvas to be displayed is `globalView` which has been defined in the previous construct. It will also receive, from the server, the assigned room number and the problem to be diagnosed.

The statement

```
deviceName = "notSelected";
```

is simply an assignment statement to set an initial value to `deviceName` to indicate that the user has not selected any network device for information or configuration. Selection of the network device is made by clicking on the icon in the `localView` mode.

The statements

```
define canvas localView {  
  if (roomNumber==1)  
    use canvas localView1;  
  else if (roomNumber==2)  
    use canvas localView2;  
  else  
    use canvas localView3;  
}
```

will restrict the user to a specific `localView` canvas based on the `roomNumber` assigned by the server.

Program 6.3f JACIE implementation of Network Troubleshooting (cont)

```

on session {
  if (currentView==1) {
    use canvas globalView;
    on MOUSECLICK {
      if (GETGRID==ViewIcons)
        if (GETGRIDX==0)
          print "You are in Global View";
        else
          if (GETGRIDX==1)
            currentView = 2;
          else
            print "[Problem] "+problem;
    }
  }
  else { //currentView = 2
    use canvas localView;
    on MOUSECLICK {
      if (GETGRID==ViewIcons)
        if (GETGRIDX==1)
          print "You are in Local View";
        else
          if (GETGRIDX==0)
            currentView = 1;
            // exit and refresh screen
          else
            print "[Problem] "+problem;
      // request information of the selected device from the server
      ...
      send checkDevice deviceName;
    }
    on TEXTENTERED {
      input instruction;
      clear localmessage;
      if (deviceName=="notSelected")
        print "Please select the device by clicking on its icon";
      else if (instruction=="switch on" || instruction=="switch off") {
        if (deviceType(deviceName)=="cable" || deviceName=="gateway")
          print "Invalid instruction-Cannot switch on/off a cable or a router";
        else
          send userRequest deviceName, instruction;
      }
      else // handle connect cable, disconnect cable
        ...
      else // handle change ipaddress, change netmask, change defaultgateway
        ...
      else if (extractWord(instruction,1)=="ping") {
        if (deviceType(deviceName)=="workstation"
            || deviceType(deviceName)=="server")
          if (deviceType(extractWord(instruction,2))=="cable"
              | deviceType(extractWord(instruction,2))=="hub")
            print "Invalid instruction - You may only ping to a remote "+
                  "workstation, server, printer or gateway";
          else
            send userRequest deviceName, instruction;
        else
          print "Invalid instruction-You may only ping from a workstation or "+
                "a server";
      }
      else
        print "["+deviceName+"] Invalid instruction entered";
    }
  }
  ...
}

```

Program 6.3g *JACIE implementation of Network Troubleshooting (cont)*

```

...
on NEWMESSAGE {
  if (MESSAGEID==deviceConfig) {
    receive deviceConfig deviceName, config;
    print servermessage "["+deviceName+"] "+config;
  }
  else if (MESSAGEID==problemSolved) {
    receive problemSolved;
    print servermessage "The network problem has been solved!";
  }
  else // new problem to be diagnosed
    ...
  else // exit
    ...
}
}
}

```

Programs 6.3f and 6.3g show the main construct of NTS. As shown in 6.3f, if the user's `currentView` is 1 (for global view), he or she can switch to local view by clicking on the local view icon (`GETGRIDX==1`) of the `ViewIcons` grid. He or she can also display the problem by clicking on the problem icon and the problem will be displayed on local message bar (as depicted by the statement `print "[Problem] "+problem;`)

Three events are captured within the `on session` construct if the user is in a local view – `on MOUSECLICK`, `on TEXTENTERED` and `on NEWMESSAGE`. If the event is `on MOUSECLICK`, depending on the clicked icon, it will respond by displaying the problem statement (if it is the problem icon) or toggling to global view from local view, or sending a request to the server for information of the selected network device (a workstation, a printer, a server, a hub, a gateway or a cable). The selected device will be the active device and can be reconfigured through one of the NTS command instructions. The operations are shown by the following statements:

```

on MOUSECLICK {
  if (GETGRID==ViewIcons)
    if (GETGRIDX==1)
      print "You are in Local View";
    else
      if (GETGRIDX==0)
        currentView = 1;
        // exit and refresh screen
      else
        print "[Problem] "+problem;
  // request information of the selected device from the server
  ...
  send checkDevice deviceName;
}

```

A command instruction for configuring the network device is entered on the text input bar.

It is handled by the `on TEXTENTERED` statement:

```
on TEXTENTERED {
  input instruction;
  clear localmessage;
  if (deviceName=="notSelected")
    print "Please select the device by clicking on its icon";
  else if (instruction=="switch on" || instruction=="switch off") {
    if (deviceType(deviceName)=="cable" || deviceName=="gateway")
      print "Invalid instruction-Cannot switch on/off a cable or a router";
    else
      send userRequest deviceName, instruction;
  }
  else // handle connect cable, disconnect cable
    ...

  else // handle change ipaddress, change netmask, change defaultgateway
    ...

  else if (extractWord(instruction,1)=="ping") {
    if (deviceType(deviceName)=="workstation"
        || deviceType(deviceName)=="server")
      ...
  }
  else
    print "["+deviceName+"] Invalid instruction entered";
}
```

As shown, with the help of `extractWord` and `deviceType` methods, it will check the syntax of the instruction and the validity of the operation. It will send the `userRequest` message to the server, together with the currently active `deviceName` and the instruction to be performed.

The code also demonstrates another event control statement, namely `on NEWMESSAGE`. The statement handles all incoming messages from the server and executes different statements depending on the message identifiers (as shown by the system variable `MESSAGEID`):

```
on NEWMESSAGE {
  if (MESSAGEID==deviceConfig) {
    receive deviceConfig deviceName, config;
    print servermessage "["+deviceName+"] "+config;
  }
  else if (MESSAGEID==problemSolved) {
    receive problemSolved;
    print servermessage "The network problem has been solved!";
  }
  else // new problem to be diagnosed
    ...

  else // exit
    ...
}
```

Program 6.3h JACIE implementation of Network Troubleshooting (cont)

```
on session end {
    clear servermessage;
}
```

One of the messages is `deviceConfig`, which carries the information of the selected active device or the current reconfigured information of the device. The information is then displayed on the server message display area.

The last construct of the `client` implementation of NTS is `on session end`. Program 6.3h shows the code. The only statement in use is `clear servermessage` to clear the server message bar.

The `server` implementation component specifies code for a server process. It starts with declarations of variables and methods. As mentioned in Chapter 4, JACIE variables and methods can be declared as `shared` – meaning that there will be only one copy referred to by all instances of remote users. The “unshared” variables, on the other hand, are local to each client connection. Program 6.3i and 6.3j are excerpts from the NTS script. Some of the variable declarations specified are as follows:

```
shared string problem;
shared int problemNumber;
shared string[31] netDevice = { ... };
    = {"WS11","WS12","WS13","P1","Hub1", // devices in room 1
      "Cable11","Cable12","Cable13","Cable14","Cable15",
      "WS21","WS22","WS23","P2", // devices in room 2
      "Cable21","Cable22","Cable23","Cable24",
      "Cable25","Cable26","Cable27","Cable28",
      "S31","S32","S33","Hub3", // devices in room 3
      "Cable31","Cable32","Cable33","Cable34",
      "Gateway"};

shared string[31][4] netConfig
    = { {"on","161.139.67.1","255.255.255.0","161.139.67.250"}, // WS11
      {"on","161.139.67.2","255.255.255.0","161.139.67.250"}, // WS12
      {"on","161.139.67.3","255.255.255.0","161.139.67.250"}, // WS13
      {"on","161.139.67.4","255.255.255.0","161.139.67.250"}, // P1
      {"on","na","na","na"}, // Hub1
      {"connected","na","na","na"}, // Cable11
      {"connected","na","na","na"}, // Cable12
      ...
      {"on","161.139.67.250","161.139.68.250","161.139.69.250"} }; // Gateway

shared int[31][31] reachability = 1; // 1=reachable
...
int roomNumber;
string instruction;
string deviceName;
string deviceStatus;
```

Program 6.3i JACIE implementation of Network Troubleshooting (cont)

```

server implementation {
  declaration {
    shared string problem;
    shared int problemNumber;

    shared string[31] netDevice
      = { "WS11", "WS12", "WS13", "P1", "Hub1",           // devices in room 1
          "Cable11", "Cable12", "Cable13", "Cable14", "Cable15",
          "WS21", "WS22", "WS23", "P2",                 // devices in room 2
          "Cable21", "Cable22", "Cable23", "Cable24",
          "Cable25", "Cable26", "Cable27", "Cable28",
          "S31", "S32", "S33", "Hub3",                 // devices in room 3
          "Cable31", "Cable32", "Cable33", "Cable34",
          "Gateway"};

    shared string[31][4] netConfig
      = { {"on", "161.139.67.1", "255.255.255.0", "161.139.67.250"}, // WS11
          {"on", "161.139.67.2", "255.255.255.0", "161.139.67.250"}, // WS12
          {"on", "161.139.67.3", "255.255.255.0", "161.139.67.250"}, // WS13
          {"on", "161.139.67.4", "255.255.255.0", "161.139.67.250"}, // P1
          {"on", "na", "na", "na"}, // Hub1
          {"connected", "na", "na", "na"}, // Cable11
          {"connected", "na", "na", "na"}, // Cable12
          ...
          {"on", "161.139.67.250", "161.139.68.250", "161.139.69.250"} }; // Gateway

    shared int[31][31] reachability = 1;

    shared void createProblem(int problemNumber) {
      // this method generates a problem from a predefined set
    }

    shared string extractWord(string theText, int pos) {
      // this method parses theText and returns the word at pos position
    }

    shared int findDeviceIndex(string deviceName) {
      // this method returns the index of deviceName as maintained by netDevice
      // or -1 if not found
    }

    shared string checkStatus(int deviceIndex) {
      // this method returns current information of the deviceIndex device
      int i = deviceIndex;
      if (i== -1)
        return "device not found";
      else if (i==0 || i==1 || i==2 || i==3           // workstations/printer in room 1
              || i==10 || i==11 || i==12 || i==13   // workstations/printer in room 2
              || i==22 || i==23 || i==24)           // servers in room 3
        if (netConfig[i][0] == "off")
          return "Status=off";
        else
          return "Status=on"
            +" IPAddress="+netConfig[i][1]
            +" netMask="+netConfig[i][2]
            +" defaultGateway="+netConfig[i][3];
      ...
    }

    shared string checkStatus(string deviceName) {
      return checkStatus(findDeviceIndex(deviceName));
    }
  }
}

```


Program 6.3j JACIE implementation of Network Troubleshooting (cont)

```

shared string reconfigure(string deviceName, string parameter, string value) {
// This method reconfigures the network device named deviceName with parameter
// and value specified. It will also check reachability of devices due to the
// new reconfiguration
int i = findDeviceIndex(deviceName);
int j;
string newStatus = "";
if (value=="on") {
    if (netConfig[i][0]=="on")
        newStatus = "Status=on (no change)";
    else {
        netConfig[i][0] = "on";
        newStatus = checkStatus(i);
        // reconfigure reachability
        if (deviceName=="Hub1") // if Hub1 is switched on
            for (j=0; j<31; j=j+1) {
                if (netConfig[j][0]=="on") {
                    if (netConfig[0][0]=="on") { // if WS11 is on,
                        reachability[j][0] = 1; // all online device
                        reachability[0][j] = 1; // can connect to it
                    }
                }
            }
        else if (deviceName=="Hub3") // if Hub3 is switched on
            ...
        else // if other device
            for (j=0; j<31; j=j+1) { // all online device
                if (netConfig[j][0]=="on") { // can connect to it
                    reachability[i][j] = 1;
                    reachability[j][i] = 1;
                }
            }
    }
}
else if (value=="off") {
    // handle reconfiguration and reachability
    ...
}
// handle other instructions
...
return newStatus;
}

void replyInstruction(string deviceName, string instruction) {
// this server method responds to the instruction given by the client
...
if (instruction=="switch on")
    // switch on the respective device
    config = reconfigure(deviceName,null,"on");
...
else if (extractWord(instruction,1)=="change"
        & extractWord(instruction,2)=="ipaddress")
    // reconfigure IPAddress for the respective device
    config = reconfigure(deviceName,"IPAddress",extractWord(instruction,3));
else
    ...
else if (extractWord(instruction,1)=="ping") {
    // ping remote device
    ...
    send deviceConfig deviceName, config;
    ...
}

int roomNumber;
string instruction;
string deviceName;
string deviceStatus;
}

```

As shown above, the shared variables are `problem`, `problemNumber`, `netDevice`, `netConfig` and `reachability`. They are common to all client connections. The “private” client-specific variables are `roomNumber`, `instruction`, `deviceName` and `deviceStatus`. This can be understood because each user is assigned a different room number and during the active session he or she is dealing with different command instructions, different device names, and different device statuses.

The array `netDevice` maintains the names of all the devices and the two-dimensional array `netConfig` maintains the current status of each device. The first value is the online/offline or, if cables, connected/disconnected status. This is followed by the IP address, the netmask and the default gateway. For the gateway, these are the IP addresses of the network interfaces. The two-dimensional array `reachability` simply determines if two devices are reachable from one another. Notice the simple way JACIE handles initialisation of arrays.

Program 6.3i and 6.3j also show some shared methods. They are `createProblem`, `extractWord`, `findDeviceIndex`, `checkStatus` and `reconfigure`. As its name implies, `createProblem` defines the problem to be solved by the communicating users. Three problems have been hardcoded for the users to troubleshoot. It is also possible (but not implemented in this version) for the problems are to be randomly generated from a set of common network problems. The method `extractWord` is the same as the one defined in client implementation and it serves the same purpose except at the server side. The two `checkStatus` methods are examples of how methods can be overridden. They simply check the `netConfig` array with the help of `findDeviceIndex` method. The method `reconfigure` reconfigures `netConfig` and redefines the reachability among devices.

Another method, `replyInstruction`, is “private” to each client. It is designed in such a way that it can send messages directly to the respective client (denoted by the use of `send` statement in the method). If it is made “shared” the server will not be able to know which client is to receive the message. The method updates `netConfig` and `reachability`

values accordingly by calling `reconfigure` method. This example also shows the importance of a correct choice of shared or “unshared” methods and variables.

Program 6.3k shows the two constructs that follow declaration in server implementation. They are `on server start` and `on session start`. In NTS, once the server first starts its process or after reinitialisation when all users have left the sessions, it will set the `problemNumber` to 1 and create the first problem. It stays alive, waiting for the new client to establish connection from then on.

Once a new client successfully established its connection to the server, the following statements in the `on session start` construct will be executed.

```
roomNumber = USERNUMBER;
send roomAssigned roomNumber;
send problemStatement problem;
```

`USERNUMBER` is a system variable that being assigned to each user upon connection. In this case, the `roomNumber` takes the `USERNUMBER` value so that the first remote user gets Room 1, and so on. With the `send` statements, the user is then notified of his or her assigned room and the problem to be solved. Since this is a multi-user application, the server will stay at this state until the specified number of users have been reached.

Program 6.3l shows the main session of the NTS, the `on session` construct. This construct mainly responds to the incoming message (denoted by the event control statement `on NEWMESSAGE`) depending on the `MESSAGEID`. As many of the operations have been defined in methods, the code has reasonably been simplified.

Program 6.3k *JACIE implementation of Network Troubleshooting (cont)*

```
on server start {
  problemNumber = 1;
  createProblem(problemNumber);
}

on session start {
  roomNumber = USERNUMBER;
  send roomAssigned roomNumber;
  send problemStatement problem;
}
```

Program 6.3l JACIE implementation of Network Troubleshooting (cont)

```
on session {
  on NEWMESSAGE {
    if (MESSAGEID==checkDevice) {
      receive checkDevice deviceName;
      deviceStatus = checkStatus(deviceName);
      send deviceConfig deviceName, deviceStatus;
    }
    else if (MESSAGEID==userRequest) {
      receive userRequest deviceName, instruction;
      replyInstruction(deviceName, instruction);
    }
  }
}
```

Program 6.3m JACIE implementation of Network Troubleshooting (cont)

```
on session end {
}

on server end {
  problemNumber = 1;
  createProblem(problemNumber);
}
}
```

The two types of messages are `checkDevice` and `userRequest`. On receiving a `checkDevice` message, the server will call the `checkStatus` method and will reply with the `deviceConfig` message. If a reconfiguration is to be made, as indicated by the `userRequest` message, the `replyInstruction` method will be called. As mentioned before, this method updates `netConfig` and `reachability` values accordingly and replies with the message.

The last two constructs of server body are shown in Program 6.3m. The constructs are `on session end` and `on server end`. No statement is required for the former. The latter, executed when all users leave the session, simply resets the `problemNumber` variable and calls the `createProblem` method so that the server is ready for the next action. This completes the NTS coding in JACIE.

6.2.2 Compilation and Execution

After compiling this script using the JACIE compiler, Java classes for the client program and the server program were generated. Table 6.2 and Table 6.3 list these Java classes. The many different classes are due to the fact that JACIE employs a composite design pattern for its standard user interface in the client program (refer section 5.4.2), while in the server program these classes represent different components that make up the server process (refer to sections 5.1.2 and 5.4.3).

Class Name	Description
NetTrouble.java	The main Java applet.
NetTroubleFrame.java	The window opened by the Java applet.
NetTroubleContainer.java	The container for the window.
NetTroubleMenuBar.java	The menu bar embedded in the container.
NetTroubleCanvas.java	The graphic canvas embedded in the container.
NetTroubleMessageBar.java	The message bar embedded in the container.
NetTroubleClientSessionAssistant.java	The client session assistant used by the container for communication.
NetTroubleUserList.java	The user list dialog box.
NetTroubleChatChannel.java	The chat channel dialog box.
NetTroubleAbout.java	The help dialog box
JACIEGrid.java	The class for grid handling in JACIE-generated programs.
JACIEImageButton.java	The class for custom-built image button in JACIE-generated programs.
JACIEMessage.java	The class for the client/server message in JACIE-generated programs.
JACIEMessageQueue.java	The class for queue handling of the JACIEMessage.
JACIEBox.java	The class for better visual effects of JACIE-generated programs.
NetTrouble.html	The Web page that called NetTrouble.java applet.

Table 6.2 *The Java classes for the client program of the Network Troubleshooting Application*

Class Name	Description
NetTroubleServer.java	The main server program.
NetTroubleServerSessionManager.java	The session manager of the server program.
NetTroubleServerSessionAssistant.java	The client-instance session assistant.
NetTroubleServerDeliveryManager.java	The delivery manager of the server program.
NetTroubleServerDeliveryAssistant.java	The client-instance delivery assistant.
JACIEMessage.java	The class for the client/server message in JACIE-generated programs.
JACIEMessageQueue.java	The class for queue handling of the JACIEMessage.

Table 6.3 *The Java classes for the server program of the Network Troubleshooting Application*

These classes were then compiled and deployed to the host machine, i.e., asahipc2. The host machine was also the Web Server. The server program was then activated at port 3333.

The NTS has been tested for execution. Three remote users have accessed the client program by means of the `NetTrouble.html` page generated by the JACIE compiler. Figures 6.6, 6.7 and 6.8 show screenshots of the three users on local view.

As mentioned earlier, diagnosing network problems in NTS is to be performed collaboratively. Utilising simple instructions to check and configure/reconfigure network devices under their control, users can converse among themselves in a process of finding the right solution to the given problem. Conversations are done through JACIE's standard chat channel.

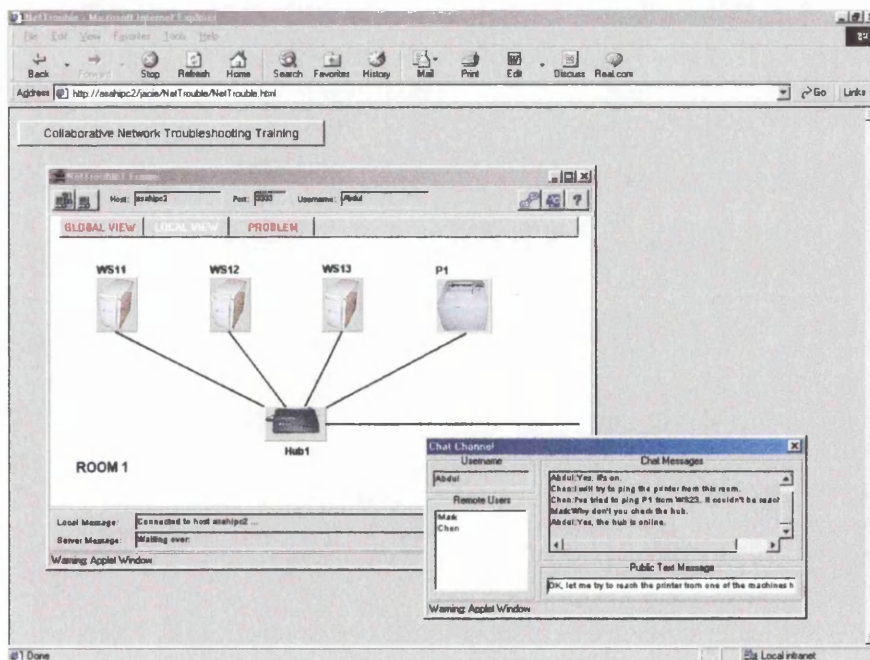


Figure 6.6 The Network Troubleshooting course in local view for Room 1 with a chat channel window

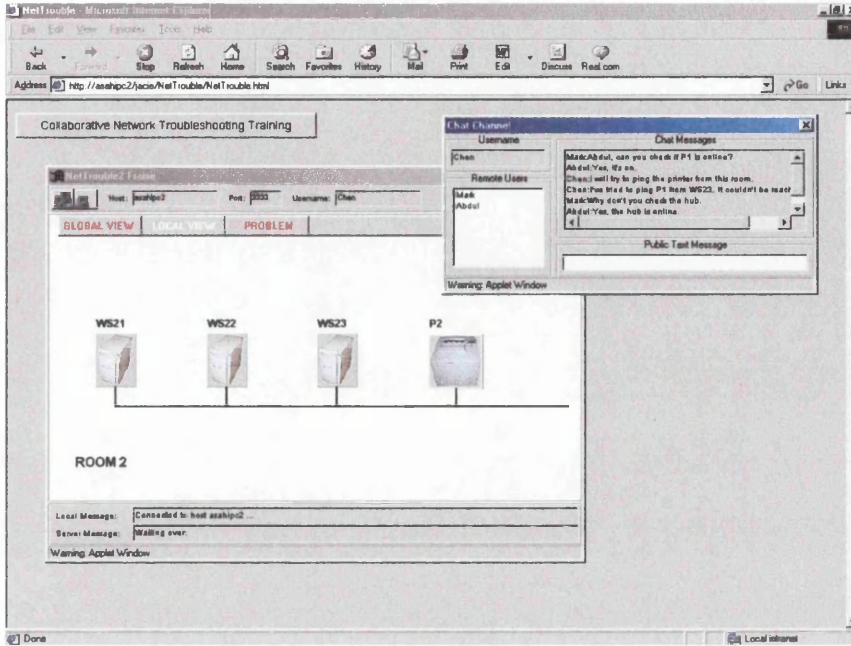


Figure 6.7 The Network Troubleshooting course in local view for Room 2 with a chat channel window

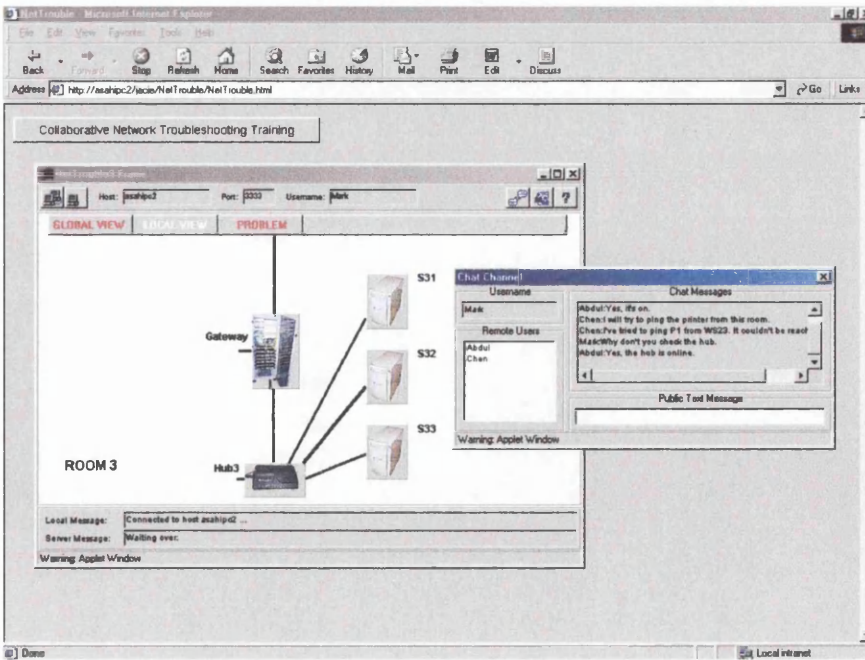


Figure 6.8 The Network Troubleshooting course in local view for Room 3 with a chat channel window

Figures 6.9 and 6.10 present the transcripts of the conversation along with the actions taken during a test run. The conversations are done through JACIE's built-in chat channel.

Problem Statement: Server S31 can't access Printer P1.

(Note that S31 is in Room 3 and P1 is in Room 1)

Captured conversation [and actions taken]

(Remote users' names are Abdul, Chen and Mark and have been designated Room 1, Room 2 and Room 3 respectively).

Mark: *Abdul, can you check if P1 is online?*

Abdul: [checking the status by clicking on P1; the device status is displayed stating the on/off status, together with the IP address, netmask, default gateway]
Yes, it's on.

Chen: *I will try to ping the printer from this room.*

Chen: [clicking on WS23 to make it the active device; the device status is displayed; issuing the ping instruction to P1]
I've tried to ping P1 from WS23. It couldn't be reached either.

Mark: *Why don't you check the hub?*

Abdul: [clicking on Hub1 to make it the active device; the device status is displayed]
Yes, the hub is online.

Abdul: *OK, let me try to reach the printer from one of the machines here.*
[clicking on WS13 to make it the active device; the device status is displayed; issuing the ping instruction to the printer]
Aha... I can't access either.

Chen: *Are you thinking what I'm thinking? Abdul, why don't you check the cable connection.*

Abdul: [clicking on Cable14; the connect/disconnect status shows that the cable is disconnected]
Yes, the cable is disconnected.
[issuing connect cable instruction]
OK, I've connected the cable.

Chen: [issuing the ping instruction to the printer]
OK, now I can ping to that printer.

Mark: [issuing the ping instruction to the printer]
I can reach the printer as well. Thank you Abdul.

Abdul: *You're welcome Mark. And thanks to you too, Chen.*

Figure 6.9 Transcript of the conversation for problem 1

Problem Statement: Workstation WS12 can't access Server S32.

(Note that WS12 is in Room 1 and S32 is in Room 3)

Captured conversation [and actions taken]

(Remote users' names are Abdul, Chen and Mark and have been designated Room 1, Room 2 and Room 3 respectively).

Mark: [checking the status by clicking on S32; the device status is displayed stating the on/off status, together with the IP address, netmask, default gateway]
I've checked S32. The server is on, the IP address is 161.139.68.12, the netmask is 255.255.255.0 and the default gateway is 161.139.68.250.

Abdul: *Let me try to ping S32 from WS12.*
[clicking on WS12 to make it the active device; the device status is displayed; issuing the ping instruction to the server]
Yes, it says unreachable.

Chen: *Abdul, why don't you try pinging S32 from other machines in your segment?*

Abdul: *I will...*

Abdul: [clicking on WS11 to make it the active device; the device status is displayed; issuing the ping instruction to server S32]
Yes, successful.

Chen: *So that means there is nothing wrong with your network segment and your hub.*

Mark: *Abdul, why don't you check the cable connection from WS12 to the hub?*

Abdul: [clicking on Cable12; the connect/disconnect status shows that the cable is connected]
Yes, the cable is connected.

Chen: [Toggling from local view to global view]
I realised you're using twisted pair cables unlike my segment which uses coax cables. Have you checked the hub, Abdul?

Abdul: *OK, let me try.*
[clicking on the hub to make it the active device; the device status is displayed]
Yes, there is nothing wrong.

Mark: *Abdul, can you check again the network configuration of WS12?*

Abdul: [clicking on WS12 to make it the active device; the device status is displayed stating the on/off status, together with the IP address, netmask, default gateway]
OK, the IP address is 161.139.67.2, the netmask is 255.255.255.0 and the default gateway is 161.139.68.250.

Figure 6.10a Transcript of the conversation for problem 2

Chen: [Toggling back from global view to local view; clicking on WS23 to make it the active device; the device status is displayed stating the on/off status, together with the IP address, netmask, default gateway]
The netmask and the default gateway are just like mine.

Mark: *Oh, that's the culprit. Segment in Room 1 shouldn't have the same default gateway as in Room 2, because they are in different subnets. Let me check the router configuration.*
[clicking on the Router (the Gateway); the device status is displayed, together with each NIC's configuration - the IP address, netmask, default gateway. Then, toggling to global view to double check which workstations belong to which subnets; and then toggling back]
Segments in Room 2 and in Room 3 are in the same subnet. The default gateway for all the workstations, servers and printers in this subnet is 161.139.68.250. But the workstations and the printer in Room 1 should have 161.139.67.250 as the default gateway. Abdul, you can check if the configuration is correct on other workstations in your segment.

Abdul: [clicking on WS11 to make it the active device; the device status is displayed, together with the IP address, netmask, default gateway – as this is the workstation which was able to access the server]
Yes, you're right Mark. The default gateway is 161.139.67.250. So I will change the configuration of WS12.
[clicking on WS12 to make it the active device; the device status is displayed; issuing change default gateway configuration instruction for this workstation]
OK, I did it.

Chen: *Abdul, why don't you try pinging S32 again?*

Abdul: [issuing the ping instruction to the server]
Yes, S32 is alive!
I can reach S32 now. We have successfully diagnosed and fixed the problem.

Mark: *One for all, all for one...*

Figure 6.10b Transcript of the conversation for problem 2 (cont)

The two transcripts show the steps taken by the collaborative users during troubleshooting process of the simulated network problem. By doing so they have successfully diagnosed, located and fixed the malfunctioning part.

6.2.3 Summary

Even though most of the collaborative activities in this network troubleshooting application were by means of a chat channel (which may not seem trivial) but a chat channel alone is inadequate as the interconnection and interdependency of network devices needs to be represented and logically maintained. JACIE with its standard chat

and canvas facilities, together with simple scripting language features, provides the mechanism for easy coding of the application.

Comparing with Java's implementation of this application (as produced by the JACIE compiler or if hardcoded by any experienced programmer), all the rich Java features used in the first type of interactive application (see 6.1.4) are required. In addition to these, multiple threads are used and they need to be carefully synchronised as to guarantee *fairness* as well as to prevent performance degradation. The Java-based JACIE language manages to handle this with a high degree of simplicity and versatility.

The NTS, which has been implemented in JACIE, is only an example of a server-mediated multi-user application. Practically many other server-based multi-user applications can be implemented using this scripting language. Among others are collaborative courseware and networked multi-user games.

6.3 A Group Collaborative Application: Collaborative Scrabble

The third example is a collaborative scrabble game. Unlike the second application described above, this Internet-based game has typical features of group collaborative applications. In this kind of application, users are organised into groups and they communicate with each other for the purpose of coordinating their activities as well as promoting a sense of community, friendship and group competition.

Different from the traditional scrabble game [126], this collaborative scrabble game has been designed for two groups with two players in each group. Each player (who is also a member of an assigned group) is given four tiles. The group competes for the highest score by collaboratively forming a word (or words) using members' available tiles and placing the tiles on the standard scrabble board during the group turn. Communications between group members (as well as among all players) are via a chat channel. In this version of the scrabble game, the server is responsible for "drawing" the tiles and "sending" them to the players, assigning turn, calculating the group score from the word

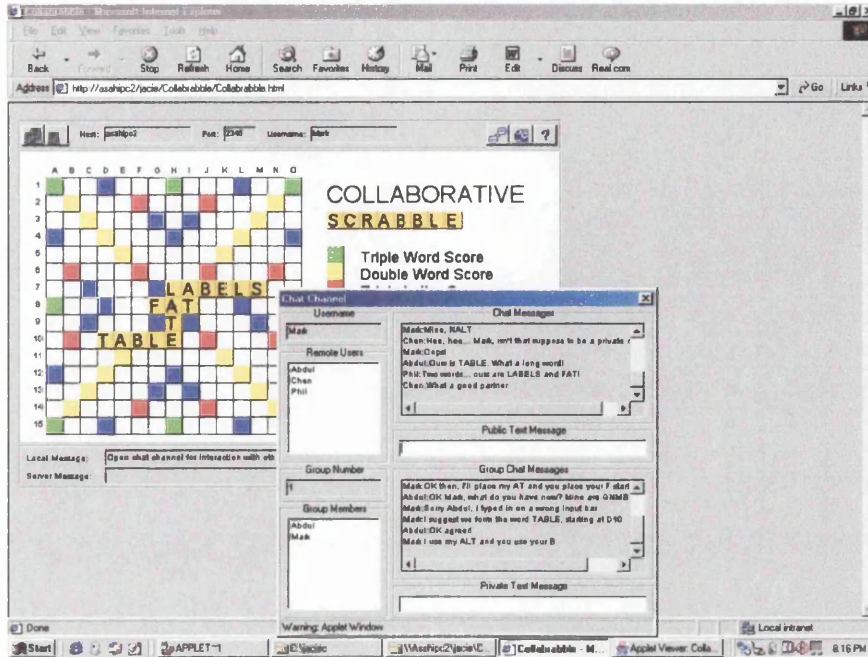


Figure 6.11 The Collaborative Scrabble game in action

(or words) formed upon completion of the group turn and informing all players of a new score. Figure 6.11 shows the screen capture of this game.

Program 6.4a JACIE implementation of Collaborative Scrabble

```
JACIE {
  applet name Collabrable;
  configuration {
    host "asahipc2";
    port 2345;
    username prompt;
    about "Collaborative Scrabble Game";
    channel canvas, chat;
    number of users 4;
    protocol roundrobin;
    number of groups 2;
    protocol of group alternate;
  }
  ...
}
```

6.3.1 Implementation

Program 6.4a shows JACIE script's configuration component of the collaborative scrabble game. As shown, this game is an applet named `Collabrabble` (after **Collaborative Scrabble**) which will be embedded directly in a Web page (unlike the previous example which is displayed on a separate applet window, activated by a text button embedded in a Web page). Like any other JACIE script, it uses `host`, `port` and `username` statements to specify the hostname, the port number and the username – the basic requirements for any client/server application. The applet uses `canvas` and `chat` channels. As expected, the `canvas` is where the game board will be displayed and also in which the user-system interactions will take place. The `chat` channel, on the other hand, is used for user-user communication and since groups are involved, the chat channel facility generated by the JACIE compiler reflects the requirement for public as well as private or group messages.

The inter-group interaction protocol (or floor management protocol) is round-robin (or in turn). As mentioned in Chapter 3, for intra-group protocol, the group members are to decide among themselves which floor control protocol is to be adopted. Negotiation is expected to be made through private chat channels. Also in this game, the players are grouped in an alternate order of his or her arrival – the first person to establish connection will be in group 1, second in group 2, third in group 1 and forth in group 2. The JACIE statements related to these requirements are:

```
number of users 4;  
protocol roundrobin;  
number of groups 2;  
protocol of group alternate;
```

Some of the message identifiers used in this game are as follows:

```
messages {  
    newTile, myMove, myTile, myPartnerMove, myPartnerTile,  
    myOpponentMove, myOpponentTile, score, opponentScore  
}
```

Program 6.4b JACIE implementation of Collaborative Scrabble (cont)

```

client implementation {
  declaration {
    string[15][15] board = "*"; // * = empty grid
    string[27] letter = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L",
                        "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X",
                        "Y", "Z", " "};
    string[15] num = {" 1", " 2", " 3", " 4", " 5", " 6", " 7", " 8", " 9",
                    "10", "11", "12", "13", "14", "15"};
    int[8][2] greenSquare = {{0,0}, {7,0}, {14,0}, {0,7}, {14,7}, {0,14},
                            {7,14}, {14,14}};
    int[24][2] blueSquare = {{3,0}, {11,0}, {6,2}, {8,2}, {7,3}, {0,3}, {14,3},
                            {2,6}, {6,6}, {8,6}, {12,6}, {3,7}, {11,7}, {2,8},
                            {6,8}, {8,8}, {12,8}, {0,11}, {7,11}, {14,11},
                            {6,12}, {8,12}, {3,14}, {11,14}};
    int[17][2] yellowSquare = {{1,1}, {2,2}, {3,3}, {4,4}, {13,1}, {12,2},
                              {11,3}, {10,4}, {7,7}, {4,10}, {3,11}, {2,12},
                              {1,13}, {10,10}, {11,11}, {12,12}, {13,13}};
    int[12][2] redSquare = {{5,1}, {9,1}, {1,5}, {5,5}, {9,5}, {13,5},
                           {1,9}, {5,9}, {9,9}, {13,9}, {5,13}, {9,13}};
    int[4] currentTile = -1; // -1 means not available
    boolean formingWord = false;
    string chosenTile;
    int letterIndex;
    ...
  }
  ...
}

```

To mention just a few, the `newTile` is associated with a message sent to each client containing the tile (or letter) “drawn” randomly by the server, the `myMove` and the `myTile` messages sent by the player to the server during his or her turn correspond to the board coordinate and the tile placed on the square respectively, whereas the `score` and the `opponentScore` messages are passed by the server to inform the group scores.

The client body component of the collaborative scrabble script starts with the declaration construct. Some of the variables used are shown in Program 6.4b.

The declaration statements, declare and initialise the variables used in the script. Just like the previous example application, the statements also show some distinctive features of the JACIE language in terms of initialising variables of an array type.

```

string[15][15] board = "*"; // * = empty cell
string[27] letter = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L",
                    "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X",
                    "Y", "Z", " "};
string[15] num = {" 1", " 2", " 3", " 4", " 5", " 6", " 7", " 8", " 9",
                 "10", "11", "12", "13", "14", "15"};
int[8][2] greenSquare = {{0,0}, {7,0}, {14,0}, {0,7}, {14,7}, {0,14},
                        {7,14}, {14,14}};
...

```

The declaration of the two-dimensional variable `board` shows how a single value can be initialised to each of the array element. The declarations of `letter`, `num`, `greenSquare` and other array variables, on the other hand, show how different values can be initialised to the array elements using curly braces (`{` and `}`). The asterisk assigned to `board` marks that the cell is empty. It will be used later to handle client-side processing which prevents the user from placing a tile in an occupied cell. The two-dimensional `greenSquare` (and other coloured squares, not shown) maintains the coordinates of all the green squares on board. This is important as each colour represent a different score value.

Following declaration is the `on canvas` construct. Like any other JACIE script, this construct specifies the default workspace canvas on which all user-defined interactions and collaborations take place. For the collaborative scrabble game, the statements in this construct basically display the board, the scores and the player drawn tiles. Most of these graphic primitives employed grid-based operations.

Program 6.4c JACIE implementation of Collaborative Scrabble (cont)

```

on canvas {
  // draw board
  foreground white;
  draw grid scrabbleBoard at 30,30 step 15,15 size 20,20 colour black width 1;
  for (int i=0; i<15; i=i+1) {
    draw string letter[i] at 37+i*20,25 font arial size 10;
    draw string num[i] at 15,42+i*20;
  }
  for (int i=0;i<8;i=i+1)
    paint grid scrabbleBoard
      at greenSquare[i][0],greenSquare[i][1] colour green;
  for (int i=0;i<24;i=i+1)
    paint grid scrabbleBoard
      at blueSquare[i][0],blueSquare[i][1] colour blue;
  for (int i=0;i<17;i=i+1)
    paint grid scrabbleBoard
      at yellowSquare[i][0],yellowSquare[i][1] colour yellow;
  for (int i=0;i<12;i=i+1)
    paint grid scrabbleBoard
      at redSquare[i][0],redSquare[i][1] colour red;
  ...
  // display the score values for premium squares (designated by different colours)
  ...
  // display the scores
  ...
  // display the tiles' rack
  draw string "YOUR TILES" at 360,300 size 18;
  draw grid myTile at 360,310 step 4,1 size 20,20 colour black width 1;
  draw string "click here to pass turn" at 495,318 size 10;
  ...
  // draw pass turn button
  draw grid passTurnPoint at 600,310 step 1,1 size 20,20 colour lightgray width 1;
  paint grid passTurnPoint at 0,0 colour lightgray;
}

```

Program 6.4c shows some statements used by the `on canvas` construct. Some of the graphics statements employed are:

```
foreground white;
draw grid scrabbleBoard at 30,30 step 15,15 size 20,20 colour black width 1;
for (int i=0; i<15; i=i+1) {
    draw string letter[i] at 37+i*20,25 font arial size 10;
    draw string num[i] at 15,42+i*20;
}
for (int i=0;i<8;i=i+1)
    paint grid scrabbleBoard
        at greenSquare[i][0],greenSquare[i][1] colour green;
for (int i=0;i<24;i=i+1)
    paint grid scrabbleBoard
        at blueSquare[i][0],blueSquare[i][1] colour blue;
...
```

The first graphics statement simply colours the canvas white. The second statement defines grid for the scrabble board. The scrabble board starts at absolute coordinate 30,30. It is a 20 by 20 grid with the size for each cell is 15 pixels by 15 pixels. One-pixel black lines are drawn to mark the grid.

The next statement is a `for` loop statement that displays letters and numbers across and downward the scrabble board. Even though there is no similar feature on the conventional game board, the letters and numbers are used for referencing purposes among remote players. The next `for` loops with `paint grid` statements simply colour the cells accordingly.

There are also other grids defined on the canvas. They are named `myTile` and `passTurnPoint`. They are used in the following context:

```
draw string "YOUR TILES" at 360,300 size 18;
draw grid myTile at 360,310 step 4,1 size 20,20 colour black width 1;
draw string "click here to pass turn" at 495,318 size 10;
...
draw grid passTurnPoint at 600,310 step 1,1 size 20,20 colour lightgray width 1;
paint grid passTurnPoint at 0,0 colour lightgray;
```

The `myTile` grid is defined at an absolute position (360,310). It consists of four cells with each one having the size of 20 pixels by 20 pixels. It is coloured light gray.

Program 6.4d JACIE implementation of Collaborative Scrabble (cont)

```
on session start {
  on WAITING {
    print "Waiting for other players";
  }
  print "Now we have enough people to play this game - enjoy!";
  wait 5;
  print "Open chat channel for interaction with other players";
  wait 5;
  for (int i=0; i<4; i=i+1)
    receive newTile currentTile[i];
}
```

As expected this `myTile` grid represents “the tile rack” that holds all the player’s current tiles. The `passTurnPoint` is a special grid that will be used later to indicate that the group has formed the word and ready to pass the turn to the other competing group.

The next construct, `on session start`, has the statements shown in Program 6.4d. Like any other JACIE script that supports multi-user operations, the `on WAITING` statement is used so that some text messages can be displayed to inform the player that the application is still waiting for more players before the game can start.

When enough players are online, each player will get the first four tiles to begin with. The players are also reminded to open a chat channel for private discussion among group members or public chat with other group members. The statements that handle these operations are:

```
print "Now we have enough people to play this game - enjoy!";
wait 5;
print "Open chat channel for interaction with other players";
wait 5;
for (int i=0; i<4; i=i+1)
  receive newTile currentTile[i];
```

The `on session` construct that follows is where other user interactions and client-server communications take place. Program 6.4e shows a code segment extracted from the game.

Program 6.4e JACIE implementation of Collaborative Scrabble (cont)

```

on session {
  // display tiles on the rack
  ...
  // display the current board
  for (int i=0; i<15; i=i+1)
    for (int j=0; j<15; j=j+1) {
      if (board[i][j] != "") {
        paint grid scrabbleBoard at i,j colour orange;
        draw string grid scrabbleBoard board[i][j] at i,j size 20;
      }
    }
  on TURN {
    print servermessage "It is your turn ...";
    on MOUSECLICK {
      ...
      gridX = GETGRIDX; gridY = GETGRIDY;
      if (GETGRID == myTile) {
        // forming a word
        formingWord = true;
        letterIndex = currentTile[gridX];
        chosenTile = letter[letterIndex];
        print "You've clicked on "+chosenTile;
        selectedTileLoc = gridX;
      }

      else if (formingWord && GETGRID==scrabbleBoard) {
        ...
        // update the board
        board[gridX][gridY] = chosenTile;
        // inform server of the move
        send myMove gridX, gridY;
        send myTile letterIndex;
        // update the rack
        ...
      }
      else if (GETGRID==passTurnPoint) {
        // pass turn
        print "passing turn to next player ...";
        pass turn;
        formingWord = false;
      }
    }
  }
  ...
}

```

As shown in the code, this JACIE script makes use of many graphics and event control statements. The grid-based operations of graphic primitives promote more readable and manageable code (as opposed to normal graphic primitives that used absolute positioning). The event handling mechanism of the JACIE language also simplifies user's interactions with the shared workspace.

```

on TURN {
  print servermessage "It is your turn ...";
  on MOUSECLICK {
    ...
  }
}

```

The captured on TURN event will execute all statements under it when the group gets its turn. This is the only time when the player can interact with the game board. However, the players can converse among themselves through the chat channel at any time.

During its group turn, the MOUSECLICK event is captured:

```
on MOUSECLICK {
  ...
  gridX = GETGRIDX; gridY = GETGRIDY;
  if (GETGRID == myTile) {
    // forming a word
    formingWord = true;
    letterIndex = currentTile[gridX];
    chosenTile = letter[letterIndex];
    print "You've clicked on "+chosenTile;
    selectedTileLoc = gridX;
  }

  else if (formingWord && GETGRID==scrabbleBoard) {
    ...
    // update the board
    board[gridX][gridY] = chosenTile;
    // inform server of the move
    send myMove gridX, gridY;
    send myTile letterIndex;
    // update the rack
    ...
  }
  else if (GETGRID==passTurnPoint) {
    // pass turn
    print "passing turn to next player ...";
    pass turn;
    formingWord = false;
  }
}
```

If the mouse click is on the rack, specified as myTile grid, a tile is selected and it is to be moved by the following mouse click. If the mouse position is in the board area (specified as scrabbleBoard grid) and as the second click, the selected tile will be placed at the appropriate grid. If the grid is passTurnPoint grid, this signifies that the group has completed its move and the turn is to be passed to the next group. All these grid-based operations are performed with the help of system variables GETGRID, GETGRIDX and GETGRIDY. If mouse click is on the position of the defined grid, GETGRID returns the name of the grid. At the same time GETGRIDX and GETGRIDY will hold the value of the relative coordinate.

Program 6.4f JACIE implementation of Collaborative Scrabble (cont)

```
on session {
  ...
  on NEWMESSAGE {
    if (MESSAGEID == myOpponentMove) {
      // update the board to reflect the opponent move
      receive myOpponentMove rgridX, rgridY;
      receive myOpponentTile remoteLetterIndex;
      board[rgridX][rgridY] = letter[remoteLetterIndex];
      refresh;
    }
    if (MESSAGEID == myPartnerMove) {
      // update the board to reflect the partner move
      ...
    }
    if (MESSAGEID == newTile) {
      // update the rack
      ...
    }
    if (MESSAGEID == score) {
      // update the group score
      ...
    }
    if (MESSAGEID == opponentScore) {
      // update the opponent score
      ...
    }
  }
}
```

Different actions are also taken if the client program receives new messages from the server which is reflected by the `on NEWMESSAGE` statement as shown in Program 6.4f. Some of the message types received from the server (as depicted by the system variable `MESSAGEID`) are `myOpponentMove`, `myPartnerMove`, `newTile`, `score` and `opponentScore`. As needed, the board and the canvas will be updated accordingly.

The final construct of the client body component (`on session end`) only prints the “The End” message. This is shown in Program 6.4g.

Program 6.4g JACIE implementation of Collaborative Scrabble (cont)

```
on session end {
  print "The End";
  wait 3;
}
```

Program 6.4h JACIE implementation of Collaborative Scrabble

```

server implementation {
  declaration {
    // boardScore: 4 = triple word, 3 = double word,
    //             2 = triple letter, 1 = double letter
    shared int[15][15] boardScore = {{4,0,0,1,0,0,0,4,0,0,0,1,0,0,4},
                                     {0,3,0,0,0,2,0,0,0,2,0,0,0,3,0},
                                     {0,0,3,0,0,0,1,0,1,0,0,0,3,0,0},
                                     {2,0,0,3,0,0,0,1,0,0,0,3,0,0,2},
                                     {0,0,0,0,3,0,0,0,0,0,3,0,0,0,0},
                                     {0,2,0,0,0,2,0,0,0,2,0,0,0,2,0},
                                     {0,0,1,0,0,0,1,0,1,0,0,0,1,0,0},
                                     {4,0,0,1,0,0,0,3,0,0,0,1,0,0,4},
                                     {0,0,1,0,0,0,1,0,1,0,0,0,1,0,0},
                                     {0,2,0,0,0,2,0,0,0,2,0,0,0,2,0},
                                     {0,0,0,0,3,0,0,0,0,0,3,0,0,0,0},
                                     {2,0,0,3,0,0,0,1,0,0,0,3,0,0,2},
                                     {0,0,3,0,0,0,1,0,1,0,0,0,3,0,0},
                                     {0,3,0,0,0,2,0,0,0,2,0,0,0,3,0},
                                     {4,0,0,1,0,0,0,4,0,0,0,1,0,0,4}};

    shared int[27] numberOfLetters = {9,2,2,4,12,2,3,2,9,1,1,4,2,6,
                                       8,2,1,6,4,6,4,2,2,1,2,1,2};
    shared int[27] letterScore      = {1,3,3,2,1,4,2,4,1,8,5,1,3,1,1,3,
                                       10,1,1,1,1,4,4,8,4,10,0};

    ...
    shared void initialiseGame() {
      ...
    }
    ...
    shared void updateBoard(int x, int y, int z) {
      ...
    }
    shared int calculateScore(int tilesUsed) {
      int score = 0;
      boolean doubleWord = false;
      boolean redoubleWord = false;
      boolean tripleWord = false;
      boolean retriipleWord = false;

      // calculate the main word's score
      // calculate other words' scores
      return score;
    }
    shared int drawTile() {
      int drawnTile = -1;
      // draw a tile randomly
      return drawnTile;
    }
    ...
    int moveX; int moveY;
    int letterIndex;
    int currentScore = 0;
    int moreTiles = 0;
  }
}

```

The server body component of the collaborative scrabble game performs many functions. For this game, the main functions of the server are to maintain the “bag” of available tiles, to randomly draw a tile at a time for all players, to calculate the group score and to mediate the communications among clients (be they inter-group or intra-group

communications). Program 6.4h is an extract of the first part of the server body component.

```
shared int[27] numberOfLetters = {9,2,2,4,12,2,3,2,9,1,1,4,2,6,
                                8,2,1,6,4,6,4,2,2,1,2,1,2};
shared int[27] letterScore     = {1,3,3,2,1,4,2,4,1,8,5,1,3,1,1,3,
                                10,1,1,1,1,4,4,8,4,10,0};
```

As shown, the tile bag is defined by two arrays, i.e., `numberOfLetters` and `letterScore`. The `numberOfLetters` basically maintains the distribution of each letter, whereas the `letterScore` keeps the letter values of the tiles. The tile can be drawn with the `drawTile()` method. As may be expected, the `numberOfLetters` will be updated accordingly when the `drawTile()` is called.

```
// boardScore: 4 = triple word, 3 = double word,
//             2 = triple letter, 1 = double letter
shared int[15][15] boardScore = {{4,0,0,1,0,0,0,4,0,0,0,1,0,0,4},
                                 {0,3,0,0,0,2,0,0,0,2,0,0,0,3,0},
                                 {0,0,3,0,0,0,1,0,1,0,0,0,3,0,0},
                                 {2,0,0,3,0,0,0,1,0,0,0,3,0,0,2},
                                 {0,0,0,0,3,0,0,0,0,0,3,0,0,0,0},
                                 ...
                                 {4,0,0,1,0,0,0,4,0,0,0,1,0,0,4}};
```

There is also an array that maintains the board score of the game, named `boardScore`. Like the conventional scrabble game [126], the `boardScore` has additional premium values used when placing tiles on premium squares, e.g., *double letter*, *triple letter*, *double word* or *triple word*. As can be seen, the JACIE's multi-dimensional array initialisation has been nicely used in this declaration statement.

Program 6.4i JACIE implementation of Collaborative Scrabble (cont)

```
on server start {
    initialiseGame();
}

on session start {
    for (int i=0; i<4; i=i+1)
        send newTile drawTile();
    moreTiles = 0;
}
```

Another important feature of the JACIE language to be noted is the use of shared modifier, where the single-copy shared variables and methods are to be referred to by all instances of remote players.

Some of the methods declared are `initialiseGame`, `updateBoard`, `calculateScore` and `drawTile`. As their names imply, these methods are to be used to perform server operations.

The next two constructs of the server body component are shown in Program 6.4i. The `initialiseGame()` method is invoked upon starting the server process. At the start of the game session four tiles will be drawn by the server and then sent to the individual players. This is done through the `drawTile()` method and the `send` statement.

Other constructs shown in Program 6.4j are the main features of the game. The code shows main message passing between the server process and the client process. Each move that the user made will be notified to other group members and opponents. The server updates the current status of the board with the `updateBoard` method.

Program 6.4j JACIE implementation of Collaborative Scrabble (cont)

```
on session {
  on TURN {
    receive myMove moveX, moveY;
    send myOpponentMove moveX, moveY to others;
    send myPartnerMove moveX, moveY to others;
    receive myTile letterIndex;
    send myOpponentTile letterIndex to group;
    send myPartnerTile letterIndex to group;
    updateBoard(moveX,moveY,letterIndex);
    moreTiles = moreTiles + 1;
  }
  currentScore = currentScore + calculateScore(moreTiles);
  send score currentScore;
  send opponentScore currentScore to others;
  for (int i=0; i<moreTiles; i=i+1)
    send newTile drawTile();
  moreTiles = 0;
}
on session end { }
on server end { }
```

Upon completing the turn the score will be calculated with the `calculateScore` method and all players will be informed of the new score. New tiles will be drawn and sent to the respective player. From this code a normal programmer can easily see the simplicity of the JACIE language as compared to other programming languages.

6.3.2 Compilation and Execution

The script has been successfully compiled with the JACIE compiler. It took less than ten seconds to generate the client and the server programs. The Java classes for the client and the server program are listed in Table 6.4 and Table 6.5. These Java classes were then compiled and deployed to the host machine, i.e. `asahipc2`. The server program was then activated at port 2345. The host machine was also the Web Server. Accessing the client program is by means of the Web page generated by the JACIE compiler.

Class Name	Description
<code>Collabrabble.java</code>	The main Java applet.
<code>CollabrabbleContainer.java</code>	The container for the Java applet.
<code>CollabrabbleMenuBar.java</code>	The menu bar embedded in the container.
<code>CollabrabbleCanvas.java</code>	The graphic canvas embedded in the container.
<code>CollabrabbleMessageBar.java</code>	The message bar embedded in the container.
<code>CollabrabbleClientSessionAssistant.java</code>	The client session assistant used by the container for communication.
<code>CollabrabbleUserList.java</code>	The user list dialog box.
<code>CollabrabbleChatChannel.java</code>	The chat channel dialog box.
<code>JACIEGrid.java</code>	The class for grid handling in JACIE-generated programs.
<code>JACIEImageButton.java</code>	The class for custom-built image button in JACIE-generated programs.
<code>JACIEMessage.java</code>	The class for the client/server message in JACIE-generated programs.
<code>JACIEMessageQueue.java</code>	The class for queue handling of the <code>JACIEMessage</code> .
<code>JACIEBox.java</code>	The class for better visual effects of JACIE-generated programs.
<code>Collabrabble.html</code>	The Web page in which <code>Collabrabble</code> applet is embedded.

Table 6.4 *The Java classes for the client program of the Collaborative Scrabble game*

Class Name	Description
<code>CollabrabbleServer.java</code>	The main server program.
<code>CollabrabbleServerSessionManager.java</code>	The session manager of the server program.
<code>CollabrabbleServerSessionAssistant.java</code>	The client-instance session assistant.
<code>CollabrabbleServerDeliveryManager.java</code>	The delivery manager of the server program.
<code>CollabrabbleServerDeliveryAssistant.java</code>	The client-instance delivery assistant.
<code>CollabrabbleServerFloorManager.java</code>	The floor manager of the server program.
<code>CollabrabbleServerGroupManager.java</code>	The group manager of the server program.
<code>JACIEMessage.java</code>	The class for the client/server message in JACIE-generated programs.
<code>JACIEMessageQueue.java</code>	The class for queue handling of the JACIEMessage.

Table 6.5 *The Java classes for the server program of the Collaborative Scrabble game*

6.3.3 Summary

This example application has illustrated many graphic features offered by the JACIE language. It is also a good example of how group collaborations operate and are handled. Essential issues relating to group collaboration, like display synchronisation and communication synchronisation, are handled without much effort and consciousness. Representation of inter-group and intra-group communication management has been realised at a very high level of programming abstraction. This example application also shows, using JACIE, the only major obstacle of designing this kind of applications is at its application logic, not on the communication and collaboration sides.

6.4 Result Analysis

The three examples of networked interactive and collaborative applications presented above provide some ground for comparison. While at first look, these applications may be too difficult to develop, the JACIE codes, in fact, are much simpler to understand than any other programming code in different programming languages. As a basis for comparison, Table 6.6 lists all the required language constructs used by the applications if they are to be developed in the Java language.

Applications	Required Java Features
Networked Knock Knock Jokes	<ul style="list-style-type: none"> • Server Socket • Socket • Exception handling • Threads • Event handling • Various input/output streams • Various class imports • Various classes • Various member modifiers
Multi-user Network Troubleshooting	<ul style="list-style-type: none"> • Server Socket • Socket • Exception handling • Multiple synchronized threads • Event handling • Various input/output streams • Various class imports • Various classes including abstract window toolkit • Various member modifiers • New semantic events • Double buffering
Collaborative Scrabble	<ul style="list-style-type: none"> • Server Socket • Socket • Exception handling • Multiple synchronized threads • Event handling • Various input/output streams • Various class imports • Various classes including abstract window toolkit • Various member modifiers • New semantic events • Thread grouping

Table 6.6 *The Java constructs required for developing example applications*

From Table 6.6, we can conclude that many advanced Java features are required to develop these applications. Highly technical knowledge is exceptionally required. Fortunately, none of these features are transparent from the JACIE programmers' point of view.

6.5 Chapter Summary

Three example networked interactive and collaborative applications are described in this chapter. The first application represents applications which employ *server-based interactive mode*. As mentioned in Chapter 3 this type of mode, where information or an application is placed on the server and to be accessed and executed by any user on the network, is the simplest of all. This is due to the fact that there is no direct interaction between concurrent executing client programs. Nonetheless, looking at the Java code, this type of application still requires some “low-level” programming. The Java and JACIE codes compared, clearly shown how JACIE’s implementation managed to hide many advanced, but complicated, features of the Java language.

The second application represents applications of type *server-mediated interactive mode* where multiple users are interacting with each other in a process of collaborative learning and working. As illustrated, things are becoming more complicated as the server, which acts as the mediator, needs to handle user-user communication as well as to manage common resources. This is additional to managing the flow and the state of the running application. JACIE, developed in consideration of different transition states that take place throughout the active client-server connection, has encapsulated much communication process handling behind the scene and, thus, hiding the very detail of multi-user network programming as opposed to Java or any other programming languages.

The third example application which implements *group collaborative mode* adds another programming complexity, as its support an intra-group communication in addition to an inter-group communication. The conventional way of programming this kind of application is unquestionably challenging. JACIE handles it elegantly. As this application also requires some graphic facilities, the built-in graphic features have simplified further the programming process.

All in all, the three example applications demonstrated above have shown the simplicity and the versatility of JACIE language.

Chapter 7

Conclusion

The programmer community has changed. In the era of personal computers and the Internet, more and more novice and casual programmers have joined the all-time sophisticated programmers who are armed with high system programming skills and sophisticated programming tools. This research recognises the increasing need of novice and casual programmers for tools that can simplify their programming tasks. Scripting languages can be the answer for their desire.

This research has proposed a novel scripting language named JACIE that is specifically designed for developing net-centric, multimedia collaborative applications. Focusing on the management of interactions and communications, JACIE offers a practical solution to programming difficulties involved in the implementation of this kind of applications.

7.1 Work Summary

The design and the proposed JACIE language specifications have been through many stages of work. The work done can be summarised as follows:

- The background study has provided a good idea of what interactive and collaborative applications are, the benefits they offer and the cost of developing these applications. Available technologies for developing these kind of applications have been studied and evaluated.
- Many interactive and collaborative applications have been investigated. Their different modes of interactions, their general features and their distinctive programming requirements have been determined. Some design principles have been stated so as to guide towards a better programming alternative as compared to the available solutions.
- The programming language of Java that has been chosen for the new tool to be built upon natively supports Internet protocols, is robust and it promotes platform-independence. The low-level networked multi-user programming requirements, which typically involved sockets handling, network input/output streams, multi-thread scheduling, shared resource management and event management, have been studied but at the same time taking into consideration these requirements are to be made hidden from programmers.
- The scripting language of JACIE has been formally specified. Guided by the particular design principles, the language constructs are then proposed which featured, among others, high-level abstractions of many interaction and communication tasks.
- In a process of constructing the JACIE compiler, several application frameworks and programming interfaces have been studied and adopted. The constructed prototype compiler recognises about 80 percent of the JACIE language features.
- Several example applications have been implemented to test and verify the viability and versatility of the language as well as the correctness of the compiler. The JACIE's implementations of the applications have become the basis for comparisons.

7.2 Result and Contribution

The two design principles laid out at the beginning of this research, namely *special purpose* and *programming efficiency*, have been the main focus of the work and eventually

have become the main contributing factor to the features of the language. The main features of the language are as follows:

- *Scripting* — The scripting programming style suits the target programmers who wish to develop the target applications. These programmers may appreciate the short learning cycle of a scripting language, in contrast to a general-purpose language, a software library, a framework or a preprocessor. Programming efficiency is achieved by reducing the complexity and inflexibility in function calls and parameter passing with traditional software libraries. Unlike some scripting languages that use cryptic characters and justifying them by “economy of expression”, JACIE encourages programmers to write readable (and thus maintainable) code by providing neat and not overly cryptic notations.
- *Template-based* — The template-based programming style suits the target applications which share a set of common features as identified in this research. It also promotes a “single program” code for the development of client/server applications, thus, suits the target programmers many of whom may lack network programming experience. From the perspective of programming efficiency, this eliminates the needs for constructing server and client programs separately. It also facilitates better correlation and coordination between server and client functionality and promotes easier software maintenance. To a certain degree, programming becomes form-filling.
- *Interaction Protocols* — JACIE introduces many high-level interaction protocols that have been designed to emulate various natural interaction and collaboration techniques. Two types of these protocols, which are floor management protocols and group management protocols, have been the main features highlighted in this language. These innovative features, unique to the JACIE language, suit the target applications that normally require one or both of the facilities. Programming-wise, these facilities reduce substantially the complexity and potential programming errors in specifying and implementing interaction protocols using low-level language constructs.
- *Built-in Channels* — JACIE supports five communication channels, namely canvas, chat, whiteboard, voice and video. These built-in channels provide the basic

communication needs in the target applications. Adopting one or more channels is just a matter of a simple specification. This reduces the complexity in programming with different APIs for requirements in communication and user interface design.

- *Event-driven* — While the common user-interface events may not be unique to the JACIE language, the high-level events are exclusive to it and will help the target programmers considerably. Programming efficiency is achieved by reducing the complexity of defining, sensing and handling of events, which typically require considerable knowledge and expertise.
- *Limited Graphics* — This feature suits the target applications for which 3D graphics play an insignificant role. This also prevents JACIE from becoming a complex and extensive language.
- *Web-ready* — JACIE also recognises the fact that the Web has become an integrative technology. Thus, JACIE-created interactive and collaborative applets can run within any Java-capable Web browser. The choice of developing either application or applet client is a matter of one keyword.
- *Java-based and Java Extensibility* — Being Java-based, JACIE is portable. JACIE-created interactive and collaborative applications can run on any platform containing the Java Virtual Machine (JVM). The feature that allows Java codes to be incorporated in the JACIE script promotes extensibility.

Many of the typical interactive and collaborative applications previously mentioned in earlier chapters could be implemented in JACIE. In general, these applications have been classified into three types — a server-based interaction type, a server-mediated interaction type and a group collaboration type. Some of the applications of the server-based interaction type are on-demand slide presentations, server-based interactive courseware and online testing, server-interaction computer games, group calendaring and online voting systems. Some examples of the server-mediated interaction type applications are text, voice and video conferencing systems, multi-user slide presentation, public chat place, shared whiteboard, multiplayer games and decision support systems. Many group collaborations can also be implemented using JACIE. They are, among others, teamwork

courseware, team-based games and other collaborative applications with structured image-based shared space or multi inter-related displays.

The three example applications reported in the last chapter are among the applications implemented in JACIE. They were selected to represent the three classifications of interactive and collaborative applications — the target applications for this scripting language. The example applications have demonstrated how the high-level abstractions of JACIE successfully handle low-level programming tasks. Typical low-level programming tasks, such as socket handling, exception handling, multiple threads synchronisations, event handling, message passing through networked input/output streams and graphics primitives including double buffering, are concealed from the eyes of the programmers. In return, JACIE provides simpler language constructs that can accomplish remarkable results in only a few lines of code. The syntax is also designed to be logical and readily understood.

Despite its strength, the current implementation of the JACIE language is not without limitations. The first limitation is that even though the programmer may specify multiple communication channels, the language does not allow the use of more than one of the same communication channels in one application, e.g. two whiteboard channels or two independent canvas channels. For the canvas channel, even though one may define multiple canvases, they can only be switched back and forth and manipulated individually within the same standard window. Another limitation of JACIE relates to graphic images to be incorporated in the script. These image files have to reside in the same directory as the JACIE script. Then and only then the image files can be copied to the target directory of the generated Java codes. Also, for the canvas, the JACIE language constructs only provide some basic graphic primitives. It does not support high-level GUI components such as buttons, check boxes, lists, text fields, text panels, graphic panels, etc. Thus, JACIE is not meant for the development of applications that are highly laden with GUI components. One such example application is a collaborative authoring editor.

The JACIE language has no native support of file handling. Hence, while it is possible to maintain the state of the running applications in memory, it is not possible to save the state permanently on a secondary storage medium for future callbacks. Multiplayer games created in JACIE, for example, are those that are to be played in one running session, as the games will be restarted at every session opening. Also, JACIE does not support the interface to external databases. There is no language construct that can be used for developing collaborative applications that need to make query to a database server. Alternatively, accessing to external files and databases can be made possible indirectly with the help of the JACIE's Java integration feature. Finally, JACIE does not have a facility that enables running applications on one machine to be shared and manipulated by collaborating remote users. This handy facility that allows meeting participants to view and work on documents of common interest simultaneously can be a new communication channel for the future implementation of JACIE.

The three types of interactive and collaborative applications can also be developed using conventional system programming languages, such as C, C++ and Java. Among these three languages, Java may be the most popular choice for its architecture-neutral, secure, network-centric and native Web browser support features. Java's support for many GUI components and its support for external file handling as well as database connectivity overcome all the limitations of JACIE mentioned above. Unfortunately, as discussed earlier, the rich language of Java is not within the grasp of many novice programmers. Alternatively, available collaborative frameworks, e.g. JCE, Habanero, JSMT, TANGO and Promondia, can also be used side-by-side with the Java language for developing interactive and collaborative applications. While it may remove some programming tasks, knowledge of the programming language where these software frameworks would be embedded is still required.

Study has also been done to investigate if the available scripting languages can provide a better alternative to normal programming languages in the development of interactive and collaborative applications. This research has established the fact that the currently available scripting languages, i.e. JavaScript, VBScript, Perl, Tcl and Python, are not

suitable development tools for the target applications. As general-purpose “glue” languages, they are not well-suited for developing real-time, complex multithreaded shared-memory applications — the very nature of the synchronous collaborative applications. Nevertheless, the philosophy of scripting languages, which is simple and direct to the point, has become the essence of the JACIE language.

Development Tools	Advantages	Disadvantages
Programming languages prior to Java (e.g. C, C++, etc.)	Used by many old-time programmers; compile to native machine code (thus, faster).	Platform-dependent; developing interactive and collaborative applications requires very low-level system programming knowledge.
Java	A programming language designed for the Internet; very rich in features; platform-independent.	Requires a good working knowledge of object-oriented programming.
Java collaborative frameworks (e.g. JCE, Habanero, JSDT, TANGO, etc)	Provide software components for collaborative tasks; ready to be integrated with custom-built applications.	A good working knowledge of Java programming language is still required so as to integrate the framework adopted and to use other language features.
Scripting languages (e.g. JavaScript, VBScript, Perl, Tcl and Python, etc)	Their programming style suits especially novice programmers.	They are not suited for developing collaborative applications — they are meant to complement system programming languages with which software components are built.
JACIE	A Java-based scripting language that combines the simple style of scripting languages and the strength of Java; provides many high-level abstractions for the handling of common interactive and collaborative processes; has built-in support of various communication channels that can be used for collaboration tools.	The basic language constructs lack strong graphic support and file/database integration (but can be implemented via the Java integration feature); the communication channels only provide some basic features.
Collaboration tools (e.g. Microsoft NetMeeting)	Commercial strength tools that allow users in the Internet or intranet to communicate via chat window, shared whiteboard, audio and video facilities; some tools also allow application sharing and file transfer among remote computers.	Requires highly-skilled programmers to integrate the facilities in custom-built collaborative applications (i.e. through COM/DCOM programming); platform-dependent.

Table 7.1 *The advantages and the disadvantages of available software tools for the development of interactive and collaborative applications*

Table 7.1 compares the advantages and the disadvantages of various available software tools, including JACIE, which can be used for the development of interactive and collaborative applications. The table also lists some popular collaboration tools — even though they are not software development tools per se. They are included only for comparison purposes.

This research has also proposed a programming model for developing interactive and collaborative applications. Extended from the normal socket-programming technique, the model utilises threads and queues for an effective, efficient, secure and manageable result.

Two conference papers, which are the outcome of this research, have been presented in two international conferences [71,72]. The second paper has received a worthy acknowledgment from the international community and has been published in a special volume *Journal of Annals of Software Engineering (ASE) on Multimedia Software Engineering* [73].

7.3 Future Work

After the period of research on the construction of JACIE, a software tool for the development of interactive and collaborative applications, some further research prospects are anticipated. Future research work can be continued based on the achievements of JACIE.

7.3.1 A Full Implementation of a Fully Reliable Compiler

Any programming language is insignificant without a fully effective and efficient compiler. Currently not all the features in the JACIE language are implemented by the compiler prototype. Full compiler development allows greater selection of applications. The codes generated should be fully optimised. An optimised Java code will provide a

better performance as opposed to a “baggy” code. The compiler should also support extensive error checking that simplifies the script debugging process. A syntax-directed editor can also be built so that a syntax error checking can be done even during the script editing process.

7.3.2 Exploring the Possible Benefits of Visual JACIE

The textual scripting language of JACIE can be awkward for some programmers and requires some programming experience to code significant functionality. Programming some features of JACIE, graphics in particular, can be made more intuitive and easy to use with a visual scripting language. A Visual JACIE development environment may help bridge the usability gap between the graphical programming environment and its script language by cognitively simplifying the scripting task. It can promote easier and faster way of doing programming and visual code can be easier to understand.

7.3.3 Extending the Web Server Feature with Collaborative Engine

The World Wide Web has become an integrative technology. More and more asynchronous collaborative applications are having new face lift with the Web. This is possible through the adoption of many scripting languages used within the Web server programming interface and as back-end gateway to other services. The time has come for the Web server to extend its feature to support synchronous collaborative application natively. A standard web server collaborative engine can be proposed. The engine should not only provide the communication and interaction infrastructure for common electronic collaboration tools, but also any domain-specific collaborative application. Software tools for collaborative applications can be used and JACIE can be one of them.

Appendix

The JACIE Language Specifications

Data Types and Operators

Primitive Types	Size/Format	Description
int	32-bit 2's complement	Integer
float	32-bit IEEE 754	Single-precision floating point
boolean	true or false	Boolean
image	gif or jpeg typed images	Image
string	a series of characters between double quotation marks ("...")	Character string; a character is represented by a string of a single character
Compound Types	Description	
array	array for managing and collecting primitive types	

Arithmetic Operators			
+	addition	/	division
-	subtraction/negation	%	modulus
*	multiplication		
Relational and Conditional Operators			
>	greater than		or
>=	greater than or equals to	&	and
<	less than	^	exclusive or
<=	less than or equals to		logical or
==	equals to	&&	logical and
!=	not equals to	!	not

Comments		
<one line comment>	// comment text	Implemented
<multiple line comment>	/* comment text */	Implemented

Language Constructs

Configuration Statements		
<JACIE program>	JACIE { <create application> <create applet> }	Implemented
<create application>	application name <identifier> ; configuration { <program configuration> } messages { <message definition> } client implementation { <client program implementation> } server implementation { <server program implementation> }	Implemented
<create applet>	applet name <identifier> ; [<create applet option>] configuration { <program configuration> } messages { <message definition> } client implementation { <client program implementation> } server implementation { <server program implementation> }	Implemented
<create applet option>	appletlauncher <text button launcher> <image button launcher> ;	Implemented
<text button launcher>	text <string>	Implemented
<image button launcher>	image <string>	Implemented
<program configuration>	{ <configuration statement> }	Implemented
<configuration statement>	<specify hostname> <specify port number> <specify username> <specify channel> <specify about> <specify number of users> <specify number of observers> <specify protocol> <specify number of groups> <specify group protocol>	Implemented
<specify hostname>	host <string> prompt ;	Implemented
<specify port number>	port <integer number> prompt ;	Implemented
<specify username>	username <string> prompt ;	Implemented

<specify channel>	channel <channel name> { , <channel name> } ;	Implemented
<channel name>	canvas chat whiteboard voice video	Partially implemented
<specify about>	about <string> <about file> ;	Implemented
<about file>	file <string>	Implemented
<specify number of users>	number of users <integer number> <specify minimum users> <specify maximum users> <specify range number of users> ;	Partially Implemented
<specify minimum users>	minimum <integer number>	Not implemented
<specify maximum users>	maximum <integer number>	Not implemented
<specify range number of users>	<specify minimum users> <specify maximum users>	Not implemented
<specify number of observers>	number of observers <integer number> ;	Not implemented
<specify protocol>	protocol contention roundrobin reservation random tapping <timed token> ;	Partially implemented
<timed token>	token <integer number>	Not implemented
<specify number of groups>	number of groups <integer number> ;	Implemented
<specify group protocol>	protocol of group userdefined random alternate ;	Partially implemented
<message definition>	[<identifier> { , <identifier> }]	Implemented

Client Implementation and Server Implementation Constructs		
<client program implementation>	declaration < variable and method declaration list> on canvas <compound statement> on session start <compound statement> on session <compound statement> on session end <compound statement>	Implemented
<server program implementation>	declaration < variable and method declaration list> on server start <compound statement> on session start <compound statement> on session <compound statement> on session end <compound statement> on server end <compound statement>	Implemented
<variable and method declaration list>	{ <variable declaration list> <method declaration list> }	Implemented

Variable Declaration Statements		
<variable declaration list>	{ [shared] <data types> <variable declarator> ; }	Implemented
<data types>	<primitive type> <compound type>	Implemented
<primitive type>	int float boolean image string	Implemented
<compound type>	<primitive type> [<expression>] <compound type> [<expression>]	Implemented
<variable declarator>	<identifier> [= <variable initialiser>]	Implemented
<variable initialiser>	<expression> <array initialiser>	Implemented
<array initialiser>	{ <variable initialiser list> }	Implemented
<variable initialiser list>	{ <variable initialiser>, }<variable initialiser>	Implemented

Expressions		
<expression>	<assignment expression>	Implemented
<assignment expression>	<conditional expression> <assignment>	Implemented
<conditional expression>	<conditional or expression>	Implemented
<conditional or expression>	<conditional and expression> <conditional or expression> <conditional or expression>	Implemented
<conditional and expression>	<inclusive or expression> <conditional and expression> && <conditional and expression>	Implemented
<inclusive or expression>	<exclusive or expression> <inclusive or expression> <inclusive or expression>	Implemented
<exclusive or expression>	<and expression> <exclusive or expression> ^ <exclusive or expression>	Implemented
<and expression>	<equality expression> <and expression> & <and expression>	Implemented
<equality expression>	<relational expression> <equality expression> == <equality expression> <equality expression> != <equality expression>	Implemented
<relational expression>	<additive expression> <relational expression> < <relational expression> <relational expression> > <relational expression> <relational expression> <= <relational expression> <relational expression> >= <relational expression>	Implemented

<additive expression>	<multiplicative expression> <additive expression> + <additive expression> <additive expression> - <additive expression>	Implemented
<multiplicative expression>	<unary expression> <multiplicative expression> * <multiplicative expression> / <multiplicative expression> % <multiplicative expression>	Implemented
<unary expression>	+ <unary expression> - <unary expression> <unary expression not plus minus>	Implementer
<unary expression not plus minus>	<postfix expression> ! <unary expression>	Implemented
<postfix expression>	<primary> <name> rnd (<expression>) <system variable>	Implemented
<primary>	<literal> (<expression>) <method invocation> <array access>	Implemented
<literal>	<int literal> <float literal> <boolean literal> <string literal> <>null literal>	Implemented
<method invocation>	<name> ({<argument list>})	Implemented
<argument list>	{ <expression> , } <expression>	Implemented
<array access>	<name> { [<expression>] }+	Implemented
<name>	<identifier> java_<identifier>	Partially implemented

Method Declaration		
<method declaration list>	{ [shared] <method header> <method body> }	Implemented
<method header>	<data type><identifier> ([<formal parameter list>]) void <identifier> ([<formal parameter list>])	Implemented
<formal parameter list>	{ <formal parameter> , } <formal parameter>	Implemented
<formal parameter>	<data type> <identifier>	Implemented
<method body>	<compound statement>	Implemented

Basic Statements		
<compound statement>	{ [<statement list>] }	Implemented
<statement list>	{ <statement> }+	Implemented
<statement>	<expression statement> <if then statement> <if then else statement> <for statement> <while statement> <return statement> <exit statement> <compound statement> <text input statement> <print text statement> <clear message bar statement> <canvas size statement> <foreground colour statement> <refresh screen statement> <clean canvas statement> <move to statement> <draw grid statement> <paint grid statement> <draw line statement> <draw image statement> <draw string statement> <canvas definition statement> <specify canvas statement> <event control statement> <pause statement> <send statement> <pass turn statement> <abort session statement>	Partially implemented
<expression statement>	<statement expression> ;	Implemented
<statement expression>	<assignment> <method invocation>	Implemented
<assignment>	<assignee> = <assignment expression>	Implemented
<assignee>	<name> <array access>	Implemented
<if then statement>	if (<expression>) <statement>	Implemented
<if then else statement>	if (<expression>) <statement> else <statement>	Implemented
<for statement>	for ([<for init>] ; [<expression>] ; [<for update>]) <statement>	Implemented
<for init>	<statement expression list> <local variable declaration>	Implemented
<local variable declaration>	<data type> <variable declarator>	Implemented
<for update>	<statement expression list>	Implemented
<statement expression list>	{ <state expression> , } <statement expression>	Implemented
<while statement>	while (<expression>) <statement>	Implemented
<return statement>	return [<expression>] ;	Implemented
<exit statement>	exit ;	Implemented

Input-output Statements		
<text input statement>	input <receiver list> ;	Implemented
<receiver list>	{ <assignee> , } <assignee>	Partially implemented
<print text statement>	print [<message bar>] [<expression list>] ;	Implemented
<clear message bar statement>	clear <message bar> ;	Implemented
<message bar>	servermessage localmessage	Implemented

Graphics Statement		
<canvas size statement>	canvas size <pair expression> ;	Not implemented
<foreground colour statement>	foreground <colour name> ;	Implemented
<background colour statement>	background <colour name> ;	Implemented
<colour name>	black blue green cyan red magenta yellow white gray darkgray lightgray orange pink	Implemented
<refresh screen statement>	refresh ;	Implemented
<clean canvas statement>	clean ;	Not implemented
<draw grid statement>	draw <grid name> [<draw at>] [<draw step>] [<draw size>] [<draw colour>] [<draw width>] ;	Implemented
<grid name>	grid <identifier>	Implemented
<draw at>	at <pair expression>	Implemented
<draw step>	step <pair expression>	Implemented
<draw size>	size <pair expression>	Implemented
<draw colour>	colour <colour name>	Implemented
<draw width>	width <expression>	Implemented
<paint grid statement>	paint <grid name> [<draw at>] [<draw colour>] ;	Implemented
<move to statement>	move to [<grid name>] <pair expression> ;	Not implemented
<pair expression>	<expression> , <expression>	Implemented
<draw line statement>	draw line [<grid name>] [<draw from>] to <pair expression> [<draw colour>] [<draw width>] ;	Implemented
<draw image statement>	draw image [<grid name>] <expression list> [<draw at>] [<draw size>] [<draw flip>] ;	Implemented

<draw string statement>	draw string [<grid name>] <expression list> [<draw at>] [<draw font>] [] [] ;	Implemented
<draw from>	from <pair expression>	Implemented
<draw flip>	flip <flip choice>	Not implemented
<flip choice>	horizontally vertically diagonally	Not implemented
<draw font>	font 	Implemented
	arial courier times	Implemented
	<expression>	Implemented
	plain bold italic bolditalic	Implemented
<canvas definition statement>	define canvas <identifier> <compound statement> ;	Partially implemented
<specify canvas statement>	use canvas <identifier> ;	Partially implemented

Event-control Statements		
<event control statement>	on <event> <statement>	Implemented
<event>	WAITING OBSERVERCONNECTION TURN GROUPTURN REQUESTCONTROL RESERVATION SERVERABORT CLIENTABORT NEWMESSAGE MOUSECLICK MOUSEPRESS MOUSERELEASE TEXTENTERED	Partially implemented
<pause statement>	wait <expression> ;	Implemented
<system variable>	USERNAME USERNUMBER GROUPNUMBER CURRENTTURN CURRENTGROUPTURN MESSAGEID GETX GETY GETGRID GETGRIDX GETGRIDY GETTEXT	Partially implemented

Communication Statements		
<send statement>	send <identifier> [<expression list> [to <send choice>]] ;	Implemented
<send choice>	server all others group	Partially implemented
<receive statement>	receive <identifier> [<receiver list>] ;	Partially implemented
<pass turn statement>	pass turn [<expression>] ;	Implemented
<abort session statement>	abort ;	Implemented

Interfacing to Java language		
<embedded java code>	Java { <java codes> }	Partially implemented

Glossary

Alternate group protocol A group management protocol that defines grouping according to an alternate order of the participants' arrival time.

Applet A small *Java* program that can be embedded in an *HTML* page.

Asynchronous collaboration A form of collaboration that does not require participants to be available at the same time.

Authoring language A high level programming language which requires less technical knowledge to master and are used exclusively for applications that present a mixture of textual, graphical and audio data

Browser A program run on a client computer for viewing World Wide Web pages.

Chat channel A communication channel that allows online text communication between remote users.

Client A computer or a program that connects to and requests information from a server.

Client-Server protocol A communication protocol between networked computers in which the services of one computer (the server) are requested by the other (the client).

Collaboration (or Cooperation) Working together in a coordinated way in the pursuit of shared goals.

Common User Access A familiar and universal "look and feel" of an application's user interface layout.

Communication channels The media through which user-user and user-server communications take place.

Compiler A program that translates a source code written in one programming language to a target code of a particular machine language or other intermediate language.

Contention protocol A floor management protocol that imposes no turn control among participants.

CSCW (acronym) Computer-Supported Cooperative Works. The field concerned with the design of computer-based systems to support and improve the work of groups of users engaged on common tasks or objectives.

CVE (acronym) Collaborative Virtual Environment. A virtual environment that supports multiple interacting users.

Cyberspace Currently used to describe the whole range of information resources available through computer networks.

Delivery Assistant A JACIE software component that assists the delivery manager for handling client-specific resources and application logics.

Delivery Manager A JACIE software component that handles shared resources and shared application logics.

Delivery protocol A low-level protocol adopted by the JACIE architecture to handle the delivery of messages from clients to a server and vice-versa.

Domain name The unique name that identifies an Internet site in a hierarchical system of delegated authority – the Domain Name System, without requiring to know the true numerical address.

Email Electronic Mail. Messages, usually text with or without attachments, sent from one person to another via computer.

Event An action or occurrence detected by a running program.

Event-driven programming A kind of programming that offers built-in support of various event handlers.

Floor Control Manager (or Floor Manager) A JACIE software component that imposes the adopted floor management protocol.

Floor management protocol A protocol of coordinating participants' turns during the running session and synchronizing access to shared resources.

Gateway Computer hardware and software that allow users to connect from one network to another.

GIF (acronym) Graphics Interchange Format. A common format for image files.

Graphics Two- or three-dimensional images, typically drawings or photographs.

Group collaboration (or Teamwork) A type of interactivity where the server is not only managing shared application and resources but also responsible for coordinating inter-group communication as well as intra-group communication and coordination.

Group Manager A JACIE software component that imposes the adopted group management protocol.

Group management protocol A protocol for grouping participants.

Host Any computer on a network that is a repository for services available to other computers on the network.

HTML (acronym) HyperText Markup Language. The coding language used to create *Hypertext* documents for use on the *World Wide Web*.

HTTP (acronym) HyperText Transfer Protocol. A set of instructions for communication between a server and a World Wide Web client.

Hypertext A document that contains links to other documents that can be chosen by a reader and which cause another document to be retrieved and displayed.

In turn protocol (refer round robin protocol)

Information superhighway The term to describe a possible upgrade to the existing Internet through the use of high speed data transmission.

Interaction protocol A protocol that defines the rules that govern the means of interactions between user-user and user-server in a collaborative environment.

internet (Lower case i) A network of networks - internetworking.

Internet (Upper case I) A global network of networks through which computers communicate by sending information in packets using the *TCP/IP* protocols.

Internet Explorer A browser developed by Microsoft.

Intranet A part of the Internet used internally within a company or organisation.

IP address (acronym) Internet Protocol Address. A unique number consisting of 4 parts separated by dots that identifies every computer on the Internet.

IRC (acronym) Internet Relay Chat. The system allowing Internet users to conduct online text based communication with one or more other users.

Java A programming language created by Sun Microsystems for developing applets and applications that are capable of running on any computer regardless of the operating system.

JavaScript A programming language that is mostly used in web pages, usually to add features that make the web page more dynamic.

JDK (acronym) Java Development Kit. A software development package from Sun Microsystems that implements the basic set of tools needed to write, test and debug Java applications and applets.

JPEG (acronym) Joint Photographic Experts Group. A common format for image files.

LAN (acronym) Local Area Network. A network of computers confined within a small area, such as an office building.

Listserv An electronic mailing list typically used by a broad range of discussion groups.

Message channel A communication channel for channeling system messages and user-defined messages.

MOO (acronym) Mud Object Oriented. One of several kinds of multi-user role-playing environments.

MUD (acronym) Multi-User Dungeon or Dimension. A (usually text-based) multi-user simulation environment in which users can create things that stay after they leave and which other users can interact within their absence, thus allowing a world to be built gradually and collectively.

Multimedia A combination of media types on a single document, including text, graphics, animation, audio and video.

MUSE (acronym) Multi-User Simulated Environment. One kind of MUD - usually with little or no violence.

Netscape Navigator A browser developed by Netscape Communication.

Newsgroup The name for discussion groups on *USENET*.

Newsreader A program designed for organizing the discussion threads received from a mailing list or newsgroup.

NNTP (acronym) Network News Transfer Protocol. A protocol that defines how news articles are passed around between computers.

Observer A remote user that has no control over the running application except as a viewer.

Packet A unit into which information is divided for transmission across the Internet.

Ping A program for determining if another computer is presently connected to the Internet.

Pixel Short for picture element - the smallest unit of resolution on a monitor. Commonly used as a unit of measurement.

Port A number representing a service that an Internet server listens on.

Protocol An agreed upon set of rules by which computers exchange information.

Random-directed protocol (or Random protocol) A floor management protocol that assigns participants' turn in random order.

Random group protocol A group management protocol that defines grouping through a random order.

Reservation protocol A floor management protocol that allows participants to reserve their turn in a queue and allow them to participate on first-come-first-served basis.

Round robin protocol A floor management protocol that allocates users' accessibility one after the other (a.k.a *in turn* or *token passing* protocol)

Session Assistant A JACIE software component that assists the session manager by handling one particular active session.

Session Manager A JACIE software component that manages the establishment of new connections.

Session protocol A low-level protocol adopted by the JACIE architecture to handle client/server connections.

Scripting language A relatively simple programming language which provide greater functionality through simple constructs and normally emphasises on "gluing" external applications written in other languages.

Shared workspace channel (or workspace channel) A special communication channel on which collaborative application can be designed and user-user interactions can take place.

Server A computer or a program that provides a service to another client computer or program.

Server-based interactions A type of interactivity where information or an application is placed on a server to be accessed and executed by independent users on the network.

Server-mediated interactions A server-based interaction but the users are interacting with one another and are coordinated by the server.

Synchronous collaboration A form of collaboration that occurs in real-time.

Tapping protocol A floor management protocol that allows the participant that is having the turn to decide (or tap) who gets the next turn.

TCP/IP (acronym) Transmission Control Protocol/Internet Protocol. A suite of protocols that defines the Internet.

Telnet The command, program and protocol used to login from one Internet site to another.

Template-based language A programming language which imposes a set of standard components.

Timed token protocol A variation of a token passing protocol with time restriction.

Token passing protocol (refer round robin protocol)

URL (acronym) Uniform Resource Locator. The standard way to give the address of any resource on the Internet that is part of the World Wide Web (WWW).

USENET A world-wide bulletin board system of discussion groups or newsgroups.

User-defined group protocol A group management protocol that defines grouping through mutual agreement among participants.

Video channel A communication channel which allows online video-based communication.

Virtual environment A virtual world that is dynamically controlled by actions of the individual in a way that it appears real to the user.

Voice channel A communication channel which allows online voice-based communication.

VR (acronym) Virtual Reality. A computer-based application (usually in 3D graphics) that provides a human computer interface so that the computer and its devices create a sensory environment.

VRML (acronym) Virtual Reality Modelling Language. A language to represent 3D compositional graphics on the Web.

Whiteboard channel A communication channel where a dedicated shared canvas is used on which arbitrary drawings are made by communicating users.

WWW (acronym) World Wide Web, or simply Web. A service on the Internet which uses a combination of text, graphics, audio and video (multimedia) to provide information on any subject.

WYSIWIS (acronym) what you see is what I see. The notion of display synchronization such that all remote participants are viewing the same thing.

Bibliography

- [1] H. M. Abdel-Wahab. Using Java for multimedia collaborative applications. In *Proc. 3rd International Workshop on Protocols for Multimedia Systems (PROMS'96)*, pages 49-62, Madrid, Spain, Oct. 1996.
- [2] H. M. Abdel-Wahab and M. A. Feit. XTV: A Framework for sharing X window clients in remote synchronous collaboration. In *Proceedings of IEEE TriComm 91: Communications for Distributed Applications & Systems*, pages 159-167, Chapel Hill, North Carolina, April 1991.
- [3] D. Adams. *WTV: An MS Windows based Collaborative System*, Master 's Project Report, Department of Computer Science, Old Dominion University, Dec. 1995.
- [4] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, MA, 1986.
- [5] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, 1998.
- [6] Apple Computer. *HyperCard Script Language Guide: The HyperTalk Language*. Addison Wesley, Reading, Massachusetts, 1990
- [7] M. Argyle. *Cooperation: The Basis of Sociability*. Routledge, London, 1991.
- [8] D. Arnold, A. Bond and M. Chilvers. Hector: Distributed Objects in Python. *Dr. Dobb's Sourcebook*. Jan/Feb 1997. [<http://www.ddj.com/articles/1997/9713/9713b/9713b.htm>]
- [9] L. Beca, G. Cheng, G. C. Fox, T. Jurga, K.Olszewski, M. Podgorny, K. Walczak. Web technologies for collaborative visualization and simulation. In *Proc. of the 8 th SIAM Conf. On Parallel Processing*, Minneapolis, March 1997.
- [10] S. Benford, J. Bowers, S. Gray, D. Leever, T. Rodden, M. Rygol, V. Stanger. The VirtuOsi Project. In *Proceedings London Virtual Reality Expo 1994 (VR'94)*, Meckler, London, Feb. 1994.

- [11] J. P. Bennett. *Introduction to Compiling Techniques – A First Course Using ANSI C, Lex and Yacc*. McGraw Hill, 1990.
- [12] R. Bentley, T. Horstmann and J. Trevor. The World Wide Web as enabling technology for CSCW: The case of BSCW. *International Journal of CSCW Special Issue on CSCW and the Web*, 6(2-3):111-134, 1997.
- [13] D. J. Berg and J. S. Fritzinger. *Advanced Techniques for Java Developers*. Wiley Computer Publishing, 1997.
- [14] E. Berk. *JLex: A lexical analyzer generator for Java*, 2000. In [<http://www.cs.princeton.edu/~appel/modern/java/JLex/manual.html>]
- [15] T. Berners-Lee. *Hypertext Transfer Protocol: A Stateless Search, Retrieve and Manipulation Protocol*, Internet draft, CERN, November 1993.
- [16] T. Berners-Lee. *Hypertext Markup Language Specification - 2.0*, Internet draft, CERN, November 1994.
- [17] T. Berners-Lee. *Weaving the Web – The Past Present and Future of the World Wide Web by its Inventor*. Orion Business Books, 1999.
- [18] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, A. Secret. The World Wide Web. *Communications of the ACM*, 37(8):76-82, 1994.
- [19] G. Blair and T. Rodden. The challenges of CSCW for open distributed processing. In *Proc. International Conference on Open Distributed Processing 1993*, pages 99-112, 1993.
- [20] W. Broll. VRML – From the Web to interactive multi-user virtual reality. In D. W. Fellner, editor, *Modeling – Virtual Worlds – Distributed Graphics, Proceedings of the International Workshop MVD'95*, pages 191-200, Bad Honnef, Germany, Nov. 27-28, 1995.
- [21] A. Bruckman and M. Resnick. Virtual Professional Community: Results from the MediaMOO project. In *Proc. The Third International Conference on Cyberspace (3CyberConf)*, May 1993.
- [22] J. Byous. Java™ Technology: An Early History. In [<http://java.sun.com/features/1998/05/birthday.html>], 1998.
- [23] M. Campione and K. Walrath. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet*. Addison-Wesley, Reading, MA, 1998.

- [24] C. Carlson and O. Hagsand. DIVE: A platform for multi-user virtual environments. *Computers and Graphics*, 17(6):663-669, 1993.
- [25] A. Chabert, E. Grossman, L. Jackson, S. Pietrowicz, and C. Seguin. Java object sharing in Habanero. *Communications of the ACM*, 41:69-76, June 1998.
- [26] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond, Washington, 1996.
- [27] A. Cockburn and T. Dale. CEVA: A tool for collaborative video analysis. In *Proc. International ACM SIGGROUP Conference on Supporting Group Work 1997 (GROUP '97)*, pages 47-55, Nov. 1997.
- [28] A. Cockburn and S. Greenberg. The design and evolution of TurboTurtle, a collaborative microworld for exploring Newtonian physics. *International Journal of Human Computer Studies*, 48(6):777-801, Academic Press, 1998.
- [29] A. Colebourne. *AC3D Manual*. Lancaster University, 2000. In [<http://www.comp.lancs.ac.uk/computing/users/andy/ac3d/man/ac3dman.html>].
- [30] *Collins English Dictionary and Thesaurus*. Diamond Books, London, 1992.
- [31] D. E. Comer. *Internetworking with TCP/IP: Principles, Protocols and Architecture*. Prentice-Hall, Singapore, 1988.
- [32] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP Volume II: Design, Implementation and Internals*. Prentice-Hall, New Jersey, 1991.
- [33] D. E. Comer and D.L. Stevens. *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications*. Prentice-Hall, New Jersey, 1994.
- [34] G. Coularis, J. Dollimore and T. Kindberg, *Distributed Systems: Concepts and Design Second Edition*. Addison-Wesley, 1994.
- [35] CSR. *Eventware Classroom*. Collaborative Systems Research Inc. Oct. 1997. In [<http://eventware.com>]
- [36] P. Curtis. Mudding: Social Phenomena in text-based virtual realities. In M. Stefik, editor, *Internet Dreams: Archetypes, Myths, and Metaphors*, pages 265-292, MIT Press, Cambridge, 1996.
- [37] P. Curtis and D. Nichols. MUDs grow up: social virtual reality in the real world. In *Digest of Papers, Spring IEEE Computer Conference 1994 (COMPCON 94)*, pages 193-200, IEEE Computer Society Press, Los Alamitos, 1994

- [38] G. DeSanctis and B. Gallupe. A foundation for the study of group decision support systems. *Management Science*, 33(5):589-609, May 1987.
- [39] A. Dieberger. Browsing the WWW by interacting with a textual virtual environment - A framework for experimenting with navigational metaphors In *Proc. Hypertext'96*, pages 170-179, Washington DC, 1996.
- [40] S. Diehl. *Distributed Virtual Worlds*, Springer-Verlag, 2001.
- [41] A. Dix, J. Finlay, G. Abowd and R. Beale. *Human-Computer Interaction*. Prentice Hall, 1993.
- [42] T. Dorsey. CU-SeeMe Desktop Video Conferencing Software. *Connexions* 9(3), March 1999.
- [43] D. Dougherty. Interview: Larry Wall and Tom Christiansen. In R. Khare, editor, *Scripting languages: Automating the Web. World Wide Web Journal*, 2(2), O'Reilly & Associates, Spring 1997. [<http://www.w3j.com/6/s1.interview.html>]
- [44] EA Sport. *FIFA2000 Online - Soccer Gaming*. In [<http://www.fifa2000.net/>]
- [45] ECMA. *Standard ECMA-262: ECMAScript Language Specification, 3rd Edition*. Dec. 1999. [<http://www.ecma.ch/ecma1/stand/ecma-262.htm>]
- [46] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, Washington, 1998.
- [47] D. C. Engelbart. A research centre for augmenting human intellect. In *Proc. The 1968 Fall Joint Computing Conference*, San Francisco, CA, 33(1):395-410, AFIPS Press, Montvale, NY, Fall 1968.
- [48] R. Evard. Collaborative networked communication: MUDs as systems tools. In *Proc. The Seventh Systems Administration Conference (LISA VII)*, pages 1-8, Monterey, CA, Nov. 1993
- [49] T. L. Fanderclai. MUDs in Education: New environments, new pedagogies. *Computer Mediated Communication Magazine*, 2(1):8, January 1995.
- [50] P. Ferris. What is CMC? An overview of scholarly definition. *Computer-Mediated Communication Magazine*, 4(1), January 1997.
- [51] G. Fitzpatrick, S. Kaplan and T. Mansfield. Physical spaces, virtual places and social worlds: A study of work in the virtual. In M. S. Ackerman, editor, *Proc. ACM CSCW'96 Conference on Computer-Supported Cooperative Work*, pages 334-343, Boston, MA, Nov. 16-20, 1996.

- [52] D. Flanagan. *JavaScript: The Definitive Guide, 3rd Edition*. O'Reilly And Associates, 1998.
- [53] F. Fluckiger. *Understanding Networked Multimedia*. Prentice Hall, London, 1995.
- [54] J. Fox. Collaborative Applications and the Java Shared Data Toolkit: What the JSDT can and can't do for you. *Dr. Dobb's Journal*, Feb. 2000.
- [55] L. Fuchs, U. Pankoke-Babatz and W. Printz. Supporting cooperative awareness with local event mechanisms: The GroupDesk system. In H. Marmolin, Y. Sundblad and K. Schmidt, editors, *The European Conference on Computer-Supported Cooperative Work (ECSCW'95)*, pages 245-260, 10-14 Sept. 1995, Stockholm, Sweden, Kluwer Dordrecht, 1995.
- [56] T. A. Furness, W. Winn and R. Yu. *The Impact of Three-Dimensional Immersive Virtual Environments on Modern Pedagogy: Global Change, VR and Learning*. National Science Foundation (NSF) Workshop Report R-97-32, Human Interface Technology Laboratory, University of Washington, 1997.
[<http://www.hitl.washington.edu/publications/r-97-32/>]
- [57] U. Gall and F. J. Hauck. Promondia: A Java-based framework for real-time group communication. In *Proc. 6th International World Wide Web Conference (WWW6)*, Santa Clara, CA. April 1997.
- [58] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [59] G. F. Gargiulo and O. R. Carmandi. *REXX with OS/2, TSO, & CMS Features: Quick Reference Guide*. Mass Market, 1999.
- [60] D. M. Geary. *Graphic Java 1.2 Mastering the JFC 3rd Edition Vol. I: AWT*. Sun Microsystems Press, Palo Alto, CA, 1999.
- [61] GNU. *Guile – Project GNU's Extension Language*. 2000.
[<http://www.gnu.org/software/guile/resources.html>].
- [62] L. M. Gomez and D. N. Gordin. Establishing project-enhanced classrooms through design. In D. Jonassen and G. McCalla, editors, *Proc. of the International Conference on Computers in Education*. Charlottesville, VA, 1996.
- [63] S. Greenberg and M. Roseman. GroupWeb: A WWW browser as real time groupware. In *ACM SIGCHI'96 Conference on Human Factors in Computing System, Companion Proceedings*, pages 271-272. 1996.

- [66] R. E. Griswold and M. T. Griswold. History of the Icon programming language. *Preprints of the 2nd ACM SIGPLAN History of Programming Languages Conference (HOPL-II)*, pages 359-360, Cambridge, Massachusetts, April 20-21, 1993. SIGPLAN Notices 28(3), March 1993.
- [67] J. Grudin. Computer-Supported Cooperative Work: History and focus. *IEEE Computer*, 27(5):19-26, May 1994.
- [68] O. Hagsand. Interactive Multiuser VEs in the DIVE system. *IEEE Multimedia*, pages 30-39, Spring 1996.
- [69] H. Hahn and R. Stout. *The Internet Complete Reference*. Osborne McGraw-Hill, Berkeley, California, 1994.
- [70] A. S. Haji-Ismail. *JACIE – Java-based Authoring language for Collaborative Interactive Environments*. Technical Report, Department of Computer Science, University of Wales Swansea, 1998.
- [71] A. S. Haji-Ismail, M. Chen, P. W. Grant. Managing interactions and communications in collaborative multimedia applications: The JACIE Way. *Proc. Second International Conference on Information, Communications and Signal Processing (ICICS'99)*, Singapore, Dec 1999.
- [72] A.S. Haji-Ismail, M. Chen, P.W. Grant and M. Kiddell. JACIE - an authoring language for rapid prototyping net-centric, multimedia and collaborative applications. In *Proc. IEEE International Symposium on Multimedia Software Engineering*, pages 385-392, Taipei, IEEE Computer Society, December 2000.
- [73] A. S. Haji-Ismail, M. Chen, P. W. Grant and M. Kiddell. JACIE – an authoring language for rapid prototyping net-centric, multimedia and collaborative applications. In J. J. P. Tsai, editor, *Annals of Software Engineering (ASE) on Multimedia Software Engineering*, pages 47-76, Kluwer Academic Publishers, 2001.
- [74] G. Hall, J. Vaisey, S. Shirmohammadi and N. Georganas. *A Survey of Web-based Telecollaboration Tools*. In [http://www.telelearn.ca/g_access/news/tools.htm]

- pages 25-30, GMD, Sankt Augustin, Germany, 1999.
- [78] S. E. Hudson. *CUP: LALR Parser Generator for Java – User’s Manual*, 2000
[<http://www.cs.princeton.edu/~appel/modern/java/CUP/manual>]
 - [79] M. Hughes, M. Shoffner, D. Hamner and U. Bellur. *Java Network Programming: Complete Guide to Networking, Streams and Distributed Computing*. Manning Greenwich, CT, 1999.
 - [80] ICQ. *ICQ Inc – (I SEEK YOU) – What is it*. In [<http://www.icq.com/products/whatisicq.html>], 2000.
 - [81] Intel. *Intel ProShare Conferencing Products*. In [<http://support.intel.com/support/proshare/>]
 - [82] H. Ishi. TeamWorkStation: Towards a seamless shared workspace. In *Proc. Conference on Computer-Supported Cooperative Work 1990 (CSCW’90)*, pages 13-26, Los Angeles, CA, 7-10 Oct. 1990.
 - [83] International Organization for Standardization (ISO). *ISO/IEC 10746-1 Information technology – Open Distributed Processing – Reference Model: Overview*, 2001. In [<http://www.iso.ch/iso/en/ittf/PubliclyAvailableStandards/C020696e.pdf>]
 - [84] ISOC. A brief history of the Internet. In *Internet Society (ISOC): All About the Internet*, [<http://www.isoc.org/internet/history/brief.html>], 2000.
 - [85] J. Jaworski and J. Jaworski. *Mastering JavaScript and JScript*. Sybex, 1999.
 - [86] R. Johansen. *Groupware: Computer Support for Business Teams*. Macmillan New York, 1988.
 - [87] D. W. Johnson and R. T. Johnson. Cooperative learning and achievement. In *Cooperative Learning: Theory and Research*. Praeger, New York, 1990
 - [88] S. C. Johnson. *Yacc – Yet Another Compiler Compiler*. CS Technical Report Bell Telephone Laboratories, New Jersey, 1975

- [91] R. Khare. Ed. Scripting languages: Automating the Web. *World Wide Web Journal*, 2(2), O'Reilly & Associates, Spring 1997.
- [92] A. A. Khwaja and J. E. Urban. Syntax-directed editing environments: Issues and features. In E. Deaton, G.H. Berghel and G. Hedrick, editors, *Applied Computing, States of the Art and Practice*, pages 230-237, ACM Press, New York, 1993.
- [93] T. Kindberg. Mushroom: a framework for collaboration and interaction across the Internet. In U. Busbach, D. Kerr and K. Sikkell, editors, *Proc. CSCW & the Web, 5th ERCIM workshop*, pages 43-53, GMD, Bonn, 1996.
- [94] G. Klein. *JFlex: The Fast Lexical Analyser Generator for Java - User's Manual* 2001. In [<http://home.in.tum.de/~kleing/jflex/manual.html>]
- [95] S. G. Kochan and P. H. Wood. *Unix Shell Programming*. Sams Publishing, 1997.
- [96] T. D. Koschmann, (Ed). *CSCL: Theory and Practice of an Emerging Paradigm*, Lawrence Erlbaum Associates, Mahwah, NJ, 1996.
- [97] C. Laird and K. Soraiz. Choosing a scripting language – Perl, Tcl and Python: they're not your father's scripting language. *SunWorld*, Oct. 1997. [http://www.sunworld.com/sunworldonline/swol-10-1997/swol-10-scripting_p.html].
- [98] L. D. Lejter. *A Collaborative Environment for Algorithm Animation on the WWW* Technical Report CS-97-07, Department of Computer Science, Brown University, May 1997. [<http://www.cs.brown.edu/publications/techreports/reports/CS-97-07.html>]
- [99] D. McConnell *Implementing Computer Supported Cooperative Learning*. Kogan Page, Logan, 1994.
- [100] C. McManis. *Looking for lex and yacc for Java? You don't know Jack*. Java World, Dec. 1996. In [<http://www.javaworld.com/javaworld/jw-12-1996/jw-12-jack.htm>]
- [101] Metamata. *JavaCC Feature Highlights*, 2000. In [<http://www.metamata.com/JavaCC/features.html>]

- [105] L. Neal. Virtual classrooms and communities. In *Proc. International ACM SIGGROUP Conference on Supporting Group Work 1997 (GROUP'97)*, page 55, Nov. 1997.
- [106] E. Nemeth, G. Snyder, S. Seebass and T. H. Hein. *UNIX System Administration Handbook*. Prentice Hall, 2000.
- [107] Netscape Communications. *Netscape Collabra*, 2000. In [<http://home.netscape.com/communicator/collabra/v4.0/index.html>].
- [108] Netscape Communications. *Netscape Conference*, 2000. In [<http://home.netscape.com/communicator/conference/v4.0/index.html>].
- [109] J. Nielsen. *Multimedia and Hypertext: The Internet and Beyond*. Academic Press, Boston, 1995.
- [110] Novell. *GroupWise*, 2000. In [<http://www.novell.com/products/groupwise/index.html>].
- [111] G. Nutt. *Operating Systems – A Modern Perspective*. Addison-Wesley, Reading, MA, 1997.
- [112] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, California, 1997.
- [113] J. Oikarinen and D. Reed. Internet Relay Chat Protocol: Request for Comments #1459, May 1993. In *Internet RFC/STD/FYI/BCP Archives* [<http://www.faqs.org/rfcs/rfc1459.html>], 2000.
- [114] OSF. *Introduction to OSF DCE*. Prentice Hall, 1995.
- [115] J. K. Ousterhoot. *Tcl and Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.
- [116] J. K. Ousterhoot. Scripting: Higher-level programming for the 21st century. *Linux Computer Magazine*, 31(3), March 1998.
- [117] Oxford. *A Dictionary of Computing*. Oxford University Press, Reading, UK, 1996.

- [121] Praxistech. *Hesse Collaborative Framework Overview*. Praxis Technical Group Jan. 1998. In [<http://www.praxistech.com/hesse/hesse.html>]
- [122] D. Rizzetto, and C. Catania. A Voice over IP service architecture for integrated communications. *IEEE Internet Computing*, 3(3):53-62, 1999.
- [123] T. Rodden, L. Bannon and K. Kuuti. The COMIC research project. In *Proc. CHI'94*, ACM Press, 1994.
- [124] M. Roseman and S. Greenberg. TeamRooms: Network places for collaboration. M. S. Ackerman, editor, *Proc. ACM CSCW'96 Conference on Computer-Supported Cooperative Work*, pages 325-333, Boston, MA, Nov. 16-20, 1996.
- [125] M. Roseman and S. Greenberg. Building groupware with GroupKit. In M. Harrison, editor, *Tcl/Tk Tools*, pages 535-564, O'Reilly Press, 1997.
- [126] Scrabble. J. W. Spear & Sons Plc, England, 2000.
- [127] S. Shirmohammadi and N. D. Georganas. Collaborating in 3D virtual environments: A synchronous architecture. In *Proc. IEEE 9th Inter. Workshops Enabling Technology Infrastructure For Collaborative Enterprises (WETICE)*. Knowledge Media Networking Workshop, NIST, Washington DC, June 2000.
- [128] P. G. Shotsberger, K. B. Smith and C. G. Spell. Collaborative distance learning the World Wide Web: Would that look like? In *Proc. Computer Support for Collaborative Learning '95*, Bloomington, Indiana, Oct. 1995.
- [129] J. Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley and Sons, 1997.
- [130] J. L. Simon. *VBScript Superbible: The Complete Reference to Programming in Microsoft Visual Basic Scripting Edition*. Waite Group, 1997.
- [131] M. Slater, M. Usoh, S. Benford, D. Snowdon, C. Brown, T. Rodden, G. Smith, S. Wilbur. Distributed Extensible Virtual Reality Laboratory (DEVRL). In M. Goebel, P. Slavik and J. J. van Wijk, editors, *Virtual Environments and Scientific Visualisation '96*, pages 137-148. Springer Computer Science, 1996.

- [147] Unix Systems Laboratories. *Unix SVR4.2: User's Guide*. Prentice Hall, New Jersey, 1992.
- [148] K. W. van den Herik and G. de Vreede. GSS for cooperative policymaking: No trivial matter. In *Proc. International ACM SIGGROUP Conference on Supporting Group Work 1997 (GROUP '97)*, pages 148-157, Nov. 1997.
- [149] L. Wall, T. Christiansen and R. L. Schwartz, *Programming Perl 2nd Edition*. O'Reilly and Associates, Inc., 1996.
- [150] A. Watters, G. v. Rossum and J. C. Ahlstrom. *Internet Programming with Python*. MIS Press/Henry Holt publishers, 1996.
- [151] WebGain. *Java Compiler compiler (JavaCC) – The Java Parser Generator*, 2001. In [http://www.webgain.com/products/java_cc/]
- [152] D. Willis. The 10 most important products of the decade – number 10: Lotus Notes. *Network Computing*, TechWeb, Oct. 2000.
- [153] T. K. Woo and M. J. Rees. A synchronous collaboration tool for World Wide Web. In *Electronic Proceedings of the Second World Wide Web Conference '94: Merging the Web and the Web*, 1994. (<http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/CSCW/woos/SynColTol.html>).
- [154] J. A. Wood. Work in progress at the Virtual Monash: the development of online learning across the University and beyond. *Active Learning*, 1 (2):13-18, 1995.
- [155] WorldsAway FAQ, 2001. In [<http://www.digitalspace.com/avatars/book/fullbook/chwa/chwafaq.htm>]
- [156] YAHOO. *Yahoo! Games*, 2001. In [<http://games.yahoo.com>]