Swansea University E-Theses

Interaction and interest management in a scripting language.

Abidin, Sita Zaleha Zainal

How to cite:

Abidin, Sita Zaleha Zainal (2006) *Interaction and interest management in a scripting language..* thesis, Swansea University.
http://cronfa.swan.ac.uk/Record/cronfa42324

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/

Interaction and Interest Management in a Scripting Language

Siti Zaleha Zainal Abidin BSc.(Michigan) MSc. (Illinois)

A thesis submitted to the University of Wales in
candidature for the degree of Philosophiae Doctor

Department of Computer Science
University of Wales, Swansea

August 2006

ProQuest Number: 10798032

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

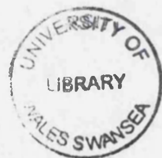
In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10798032

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


UNIVERSITY OF WALES SWANSEA
LIBRARY

Summary

Interaction management is concerned with the protocols that govern interactive activities among multiple users or agents in networked collaborative environments. Interest management is concerned with the relevance-based data filtering in networked collaborative environments. The main objective of the former is to structure interactive activities according to the requirements of the application concerned, while the main objective of the latter is to provide secured data transmission of a subset of information relevant to each recipient. The research in these two important aspects of networked software has largely been carried out in specific application domains such as online meetings, online groupware and online games.

This thesis is concerned with the design and implementation of high-level language constructs for interaction and interest management. The work that has been undertaken includes

- an abstract study of interactive activities and data transmission in networked collaborative environments through a large number of variations of the noughts and crosses game;
- the design of a set of language constructs for specifying a variety of interaction protocols;
- the design of a set of language constructs for specifying secured data sharing with relevance-based filtering;
- the implementation of these language constructs in the form of a major extension of a scripting language JACIE (*Java-based Authoring Language for Collaborative Interactive Environments*);
- the development of two demonstration applications, namely e-learning on Simulation of Network Trouble Shooting and online Bridge, using the extended JACIE for demonstrating the technical feasibility and usefulness of the design.

These high-level language constructs support a class of complicated software features in networked collaborative applications, such as turn management, interaction timing, group formation, dynamic protocol changes, distributed data sharing, access control, authentication and information filtering. They enable programmers to implement such features in an intuitive manner without involving low-level system programming directly, which would otherwise require the knowledge and skills of experienced network programmers.

Some parts within this thesis have been presented at the IEEE International Symposium on Multimedia Software Engineering 2004, Miami, Florida, USA. The parts on interaction management is published in the Elsevier Journal of Network and Computer Applications, Volume 30, Issue 2, April 2007.

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed .... (candidate)

Date 02/04/2007

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (candidate)

Date 02/04/2007

Statement 2

I hereby give my 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.

Signed .... (candidate)

Date 02/04/2007

Acknowledgements

I would like to express my sincere gratitude to both of my supervisors, Professor Min Chen and Dr. Phil W. Grant for their patient guidance, encouragement and advice they have provided throughout my PhD program. Without their guidance, this thesis would not have been possible.

I would like to take this opportunity to thank Universiti Teknologi MARA (UiTM) for their generosity in funding my study and enable my family to stay with me in the UK. Without them, my PhD life would be very difficult. Receiving this scholarship also motivates me to give my very best in achieving my goals. In particular, my appreciation must go to the UiTM management staff whose confidence and willingness to support has helped me to move forward in many aspects.

I would like to thank all my family members, here or abroad for the moral support. My special thanks to all my children, Hazirah, Hazmi Afandi and Haidah Dayini, and also to my husband, Mohamad Zailani. Their patience and sacrifices meant a great deal.

Last but not least, thanks to all my friends in the postgraduate lab and colleagues in UiTM who constantly give me support and ideas.

To all of these people, I am deeply grateful.

Contents

1 Introduction 1

1.1 Project Background 3
1.2 Aims and Objectives 6
1.3 Thesis Outline 7
1.3.1 Chapter 2: Networked Collaborative Systems 7
1.3.2 Chapter 3: Programming Languages and Tools for Developing Networked Applications 8
1.3.3 Chapter 4: JACIE Overview and Enhancements 8
1.3.4 Chapter 5: Interaction Management 8
1.3.5 Chapter 6: Interest Management 9
1.3.6 Chapter 7: JACIE Applications 9
1.3.7 Chapter 8: Conclusion 10
1.3.8 Appendices 10

2 Networked Collaborative Systems 11

2.1 Introduction 11
2.2 Collaborative Environments 12
2.2.1 Networked Collaboration 14
2.2.2 Collaborative Management 15
2.2.2.1 Centralised Management 15
2.2.2.2 Distributed Management 15
2.2.3 Collaborative Applications 17
2.3 Interaction Management 19
2.3.1 Control Methods 21
2.3.2 Management Strategies 22
2.3.3 Implementation Techniques 24
2.3.3.1 Database System Support 25
2.3.3.2 Agent System Support 25
2.3.3.3 Algorithm Based Design 26
2.4 Interest Management 26
2.4.1 Data Filtering 27
2.4.1.1 Generic Filtering Strategies 27
2.4.1.2 Filtering Factors Based on User Interest 30
2.4.1.3 Filtering Implementation 32
2.4.2 Access Control 33

2.4.2.1 Access and Security Factors 34
2.4.2.2 Management Strategies 36
2.4.2.3 Sharing Factors 37

3 Programming Languages and Tools for Developing Networked Applications 40
3.1 Introduction 40
3.2 Network Programming Languages and Tools 41
3.2.1 Web Technology 42
3.2.2 Networked Applications Development 43
3.2.2.1 Method of Communication 44
3.2.2.2 The Challenges 46
3.2.2.3 The Techniques 47
3.2.2.4 The Characteristics 49
3.2.3 Java-based Collaborative Framework 50
3.2.4 Scripting Languages 52
3.3 Interaction Management 54
3.3.1 Implementation Tools 54
3.3.1.1 Using Software Toolkits 55
3.3.1.2 Using Programming Languages 55
3.3.1.3 Using Component-Based Approach 56
3.4 Interest Management 56
3.4.1 Filtering Issues 57
3.4.2 Data Sharing 57
3.4.2.1 Sharing Factors 58
3.4.2.2 Security and Privacy 61

4 JACIE Overview and Enhancements 63
4.1 Introduction 63
4.2 Overview of JACIE 64
4.3 JACIE Compiler 65
4.4 Main Features of JACIE 67
4.5 Managing Collaboration 69
4.6 Race Condition 73
4.6.1 The Causes 73
4.6.2 The Detection 74
4.6.3 The Solution 75
4.7 Improvements to the JACIE Language 79
4.7.1 Enhancement of Data Types 79
4.7.2 Supporting Code Optimisation 81
4.7.3 Compilation Errors 82
4.8 Summary 88

5 Interaction Management 89
5.1 Introduction 89
5.2 Related Work 90
5.2.1 Interaction Protocols 90
5.2.2 Temporal Coordination 91

5.3 The Noughts and Crosses Game and Its Variations . . . 92
5.3.1 History of the Noughts and Crosses . . . 92
5.3.2 Definitions . . . 93
5.3.3 Traditional Noughts and Crosses . . . 96
5.3.4 Summary of Variations . . . 97
5.4 Interaction Management in JACIE . . . 100
5.4.1 Round Robin . . . 100
5.4.2 Contention . . . 103
5.4.3 Reservation . . . 106
5.4.4 Master . . . 107
5.4.5 Tapping . . . 108
5.4.6 Group Protocols . . . 109
5.4.6.1 Protocol Group Userdefined . . . 110
5.4.6.2 Protocol Group Roundrobin . . . 110
5.4.6.3 Protocol Group Random . . . 110
5.4.6.4 Protocol Group Master $\eta_{grp,mem}$ . . . 110
5.5 Language Enhancements . . . 111
5.5.1 The Software Architecture for Managing Collaboration . . . 111
5.5.2 Additional Tokens and Productions . . . 112
5.5.3 Additional Codes and New Java Classes . . . 115
5.6 Other Protocol Design Issues . . . 116
5.6.1 Static and Dynamic Interaction Protocol Settings . . . 116
5.6.2 Timer Implementation . . . 117
5.6.2.1 Server Based Timer . . . 117
5.6.2.2 Client Based Timer . . . 119
5.6.2.3 Timer Interrupt . . . 119
5.7 Summary . . . 121

6 Interest Management **123**
6.1 Introduction . . . 123
6.2 Related Work . . . 124
6.2.1 Programming Data Sharing in Distributed Systems . . . 124
6.2.2 Interest Management and Filtering Methods . . . 125
6.2.3 Access Control and Data Security . . . 126
6.3 Interest Management in JACIE . . . 127
6.3.1 Shared Variables and Attributes . . . 127
6.3.2 Management Framework . . . 128
6.3.3 Assigning Value to Shared Variable . . . 132
6.3.4 Access Control and Filtering Framework . . . 133
6.3.4.1 The Access Control . . . 135
6.3.4.2 The Filtering . . . 137
6.4 Language Constructs for Interest Management . . . 138
6.4.1 *Statement:* `use` . . . 139
6.4.2 *Statement:* `set` . . . 140
6.4.3 *Statement:* `check` . . . 141
6.4.4 *Statement:* `filter` *and* `interest set` . . . 141

6.5 Language Enhancements 142
6.5.1 Additional Tokens and Productions 142
6.5.2 Additional Code and New Java Classes 144
6.6 Technical Considerations 145
6.6.1 Mutual Exclusion 145
6.6.2 Permission List Management 146
6.6.3 Access List Management 147
6.7 Secret Switch 147
6.8 Summary 153

7 JACIE Applications and Performance Analysis 154
7.1 Introduction 154
7.2 Bridge Game 155
7.3 Implementation of the Bridge Game 157
7.3.1 Program Flow and Game Layout 157
7.3.2 Dynamic Protocol Changes 163
7.3.2.1 Method One: Protocol Round Robin Only 163
7.3.2.2 Method Two: Protocol Round Robin and Protocol Master User 164
7.3.2.3 Method Three: Protocol Round Robin and Protocol Group 165
7.3.2.4 Summary on the Bridge Game Interaction Protocol Implementations 166
7.4 E-learning on Simulation of Network Trouble Shooting 166
7.5 Performance Analysis 177
7.5.1 JACIE vs. Java Translated Program 177
7.5.2 Preliminary User Study on JACIE 179
7.5.3 Interaction vs. Transmission Delay 184
7.6 Summary 186

8 Conclusion 188
8.1 Summary of Contributions 189
8.1.1 The Collection of Interaction Protocols 189
8.1.2 Interaction Management Implementation 189
8.1.3 Interest Management Implementation 190
8.1.4 Major Language Enhancements 191
8.1.5 Demonstration Applications 191
8.1.6 Minor Improvements on the Language and Compiler 191
8.2 Future Work 192

A Variations of the Noughts and Crosses Games 194
A.1 Five-in-a-line 194
A.2 Connect-4 194
A.3 Three Stones 195
A.4 Hasty Battle 195
A.5 Vicious Battle 196
A.6 Gentlemen's Battle 197
A.7 Dictator's Entertainment 197

A.8 First-Come, First Served 197
A.9 Opportunity Knocks 198
A.10 Secret Switch 198
A.11 Group Games 199

B Nomenclature **200**

C The JACIE II Language Specifications **202**
C.1 Token Specifications 202
C.2 Syntax Specifications 203
C.2.1 JACIE Program Body 203
C.2.2 JACIE Configuration Section 204
C.2.3 Message Definition 205
C.2.4 Client Implementation and Server Implementation Sections 205
C.2.5 Variable Declaration 206
C.2.6 Method Declaration Statements 206
C.2.7 Basic Statements 206
C.2.8 Expression Statements 207
C.2.9 Control Statements 207
C.2.10 Iteration Statements 208
C.2.11 Input Output Statements 208
C.2.12 Graphics Statements 208
C.2.13 Event Control Statements 209
C.2.14 Communication Statements 209
C.2.15 Interface Statement 210
C.2.16 Interaction Management Statements 210
C.2.17 Interest Management Statements 210

Bibliography **212**

List of Figures **236**

List of Tables **238**

Chapter 1

Introduction

Contents

1.1 **Project Background** 3
1.2 **Aims and Objectives** 6
1.3 **Thesis Outline** 7

Since the Internet has became a huge information highway, many researchers are interested to explore and enhance its capabilities. It consists of complex entities involving many activities. Different types of computers that use different software systems are integrated in one system. Electronic mail, chat and electronic-commerce are now routine activities on the network. On the whole, people from all over the world benefit greatly as they can share information, communicate and interact very easily [99, 120, 164, 181].

The network connection can be in the form of telephone line, cable, satellite or wireless. Among the earliest interactive communication system is the telephone with only analog voice as data. This has been superseded by interactive communication as in the *networked collaborative systems* which is no longer limited to one type of data. This can be in various form of multimedia elements, such as analog voice converted into digital form, text, images and video. In this way, people can share their work remotely as if in a face-to-face environment.

In a communication system, the user or node involved in an activity can be called a *process*. The data or any element required by the activity can be called a *resource*. When communication occurs in a collaborative system interactively, there must be more than one process involved. As communication can occur concurrently, there must be rules, called *protocols* which govern the processes [141, 310, 287]. The need of protocols is to ensure that the communication between all processes is proper ordered. It is highly likely that some data will be shared among the processes. Therefore, there is a strong requirement for mechanisms to ensure that the data is in the correct form, consistent and secure. Such mechanisms are referred to as *data access and protection.*

Since there may be many users involved and numerous resources are available, it is a challenge to design such systems, especially *interactive networked collaborative systems* where many people can interact and communicate at the same time. Therefore, a networked collaborative application usually has a finite number of users working together with some common or similar objectives [202]. All the users are in the *tightly coupled* context while they are collaborating. The interaction protocol to determine the users' turn and secured data sharing and transmission are among the most common issues within these systems.

Software systems are very important in making the Internet successful. Software exists to support the hardware technologies in many forms from the low level system program, to the high level application software. Examples of such software include Web browsers, network languages and various software tools. However, software for managing interaction among the remote users usually relies on low level system and network programming.

To design a structured interaction in a high level language usually requires an *application program interface* (API) [70] that enables the design of high level interaction protocols to be implemented using low level programming utilities and network protocols. Some software systems can handle interaction protocols in the context of various networked applications [78, 260, 107]. Several network languages can also be used in implementing such applications. Java [152, 105, 51] and C# [84] can be considered among the popular network languages. However, since both are general purpose languages, program codes tend to be large to implement an application. Scripting languages such as Perl [269], Python [325] and VBScript [161], have been used for network applications as these languages usually can provide simple programming [24]. These languages are also general purpose and require networked programming skills in writing a networked application. There are also several languages such as logic programming languages [271] and concurrent languages [106] that give support for networked environments, such as agent systems [166, 204, 298] and parallel systems [180]. As most of the existing programming languages mentioned above can be used to build applications in networked system in general, we specifically focus on interactive networked collaborative systems. Furthermore, in most circumstances, programming using these languages requires specific algorithms for implementing the interaction protocol that limits the flexibility of having various protocols in developing applications. Such protocols have not yet been provided in the form of language constructs.

In managing data sharing, system level programming also provides an important role to ensure the data consistency. This job is usually performed by the operating system to guarantee the *mutual exclusion* condition. At the high level, some programming languages provide the sharing of memory in most concurrent or parallel applications. The languages used in these systems, such as Orca [20] and APL [129], facilitate variable sharing through *message passing* or *remote procedure call (RPC)* [216]. Our main concern is not only the secured data sharing mechanisms, but also the relevant-based data filtering in accordance to the users' interest.

Managing interest can ensure that the selected users, who have the access rights on a certain resource, will get the shared resource only if they are interested [230]. In this case, the

shared data is handled securely and effectively. This interest management is mostly found in distributed and collaborative systems [231, 217, 229]. Such systems often involve broadcasting large amount of data, and therefore, it is necessary to filter the data according to the users' need. Simulation is often carried out to provide the most effective methods of filtering [230, 312, 323].

As mentioned above, there are several network languages available for designing and implementing networked applications. In most cases, the languages provide the sharing of components, object or memory through message passing, remote procedure calls or application program interfaces. Normally, to implement an application, an experienced network programmer is required. Hence, building such challenging tasks in the form of language constructs can reduce the effort of writing long and complicated program codes.

In most existing networked applications, the common way of handling interest management is by a centralised control [53]. Here, the server plays the important role of managing the shared data among the remote users. Therefore, it is highly desirable to have distributed and flexible control of shared data at the client end without putting all the burden on the server and provide a set of high level language constructs for handling shared data. The shared data used in a high level language is usually a variable, object or program component [117, 285].

Another popular way of implementing shared data is through a database system [53, 217]. However, in most circumstances, this method requires the support of a sophisticated system to determine secure access and the filtering needs. While this approach is reliable, it is not practical to depend on such system in implementing an application that requires only limited amount of data with basic interest management.

The lack of easy programming techniques for developing interactive networked collaborative systems leads this research to provide programmers of such systems with high level language constructs. Figure 1.1 shows some example applications for Internet collaborative activities. An online meeting permits many users to have discussions similar to the face-to-face environment. This application requires a structured order of turn control for proper sharing of information and determining the outcome. In an online form filling application, a user must enter appropriate answer according to the specified questions. In most cases, the form is designed in such a way that the user is guided throughout the question answered session which allows the user to manipulate only specific domains. For an online game application, such as card game, several users play their cards in turn according to the game rules. Some cards may be shown to all the players for determining their strategies. Hence, it is crucial that the interaction protocol and access rights are managed correctly and efficiently.

1.1 Project Background

The overall research covers a very broad area in several different fields of computer science. The main study concerns Internet computing and networking with the focus on interaction

Online Meeting [75]

Online Form Filling [250]

Card Game [309]

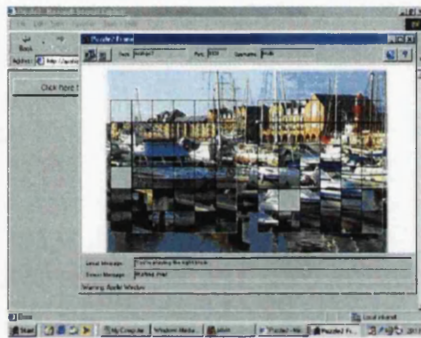
Figure 1.1: Internet Collaborative Activities.

protocols for turn control in collaboration and the control of secured shared data. Other aspects of computer science that support this research are operating system concepts and design, compiler construction, programming languages concepts and database management.

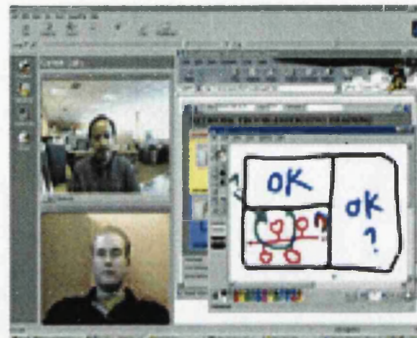
We have made a comprehensive study of interaction activities and secure data transmission. Interaction management is concerned with the protocols that govern structured interactive activities among multiple users or agents in networked collaborative environments, while interest management is concerned with the secure relevant-based data filtering. We propose the design of new high level language constructs and report our efforts for incorporating these new constructs into JACIE (*Java-based Authoring Language for Collaborating Interactive Environments*) [139], an existing scripting language designed to support rapid prototyping and implementation of networked collaborative applications. We demonstrate the usefulness of these language constructs through variations of the noughts and crosses game, an on-line bridge game and an e-learning application.

Figure 1.2 represents the output of some applications written in JACIE. The jigsaw puzzle game shows JACIE canvas channel for two players game with contention protocol. Both

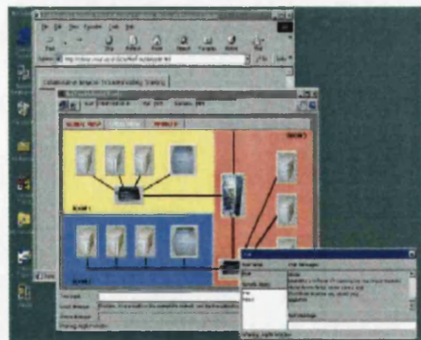
players share a common canvas to rearrange the correct order of grid square to complete the whole picture. Another example is the communication between two people through a shared whiteboard supported by *Microsoft Netmeeting* [73]. They can draw or sketch the board with the common sketching tools. The network trouble shooting example shows an e-learning application with several views of a working canvas that provides global or local view of rooms involved in the discussion. This application shows how three people in three different rooms have a discussion using a chat channel to find out the cause of network problem given by the server. One person in a specified room can view and change the status of all the devices located in the room represented in his/her local view. Other users can only give suggestions on finding out the solution based on the given global view. The group scrabble game presents a simple group collaboration with supportive private chat channels among group members. At any turn, any member can represent the group after some agreements have been made among them through private discussions.



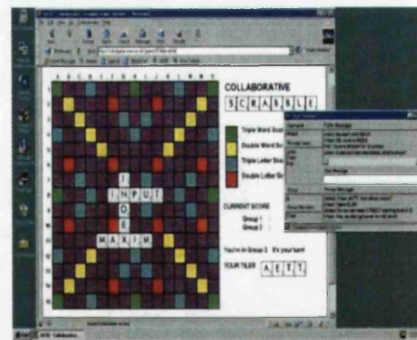
Jigsaw Puzzle



Shared Whiteboard



Network Trouble Shooting



Group Scrabble

Figure 1.2: JACIE Sample Output

JACIE I was designed originally with a simple built in interaction feature to handle interactive collaboration among users or groups [140]. The collaboration supporting multimedia activities can be achieved through several built in communication channels. Although JACIE I provided a small set of interaction protocols, it was not broad and flexible enough to cater for many desirable networked collaborative applications. For example, all the mentioned

protocols was *contention*, *round robin*, *reservation*, *random*, *tapping*, *token* and *group*, and from these, protocol contention was fully implemented and tested, while protocol round robin was implemented partially, and most of other mentioned protocols were not yet implemented. For the group protocol, it was developed to select group members instead of determining group turn protocol. The proposed selections were *alternate*, *random* and *userdefined* with full implementation and testing in the *alternate* protocol.

In managing data sharing, a programmer had to design and write a specific algorithm. JACIE I provided the programmer with message passing facilities that enable any type of variables to be exchanged with the server. In this way, data sharing between clients could be made through server. In addition, JACIE I allowed both the client and server program to reside in one file for easy managing identifiers that involved in the communication. A programmer could instruct the server to send any message either to all, some or only one of the existing clients. However, there was no sharing nor security access of variables in the form of language constructs.

1.2 Aims and Objectives

The main objectives of this work are:

1. To conduct an abstract study on the interaction activities and data transmission that can provide a variety of interaction protocols. We aim to show that within this study, we can conceptualise the interaction management, and thus, help us in the design of comprehensive collection of protocols.

2. To design a set of high level language constructs for specifying structured interaction. This design should focus on the management of floor control for user and group collaborations. The purpose is to provide the design of various protocols commonly used in interactive networked collaborative application.

3. To design a set of high level language constructs for specifying secured data sharing with relevant-based filtering. The purpose of this design is to provide a high level programming interface for managing shared data among server and users in an intuitive manner.

4. To include these language constructs in a high level programming language. With the proposed comprehensive design incorporated into an existing programming language, we can implement interactive collaborative applications.

5. To develop some applications to demonstrate the technical feasibility and the usefulness of the design of these language constructs.

6. To make some alterations and enhancements to the vehicle language in order to accommodate these new design and features. These changes may include providing some error messages for debugging purposes, adding new supportive statements or rearranging some elements in the existing statements and adding more special identifiers or *keywords*.

In general, several requirements related to this work include:

- Networked collaborative environment — In this environment, users usually rely on the provided software system to engage in a collaboration. There is no limitation on the physical distance between users.
- Interactive system — Communication between users can occur immediately that enable several users to work as a team.
- Scripting language — Language is the main vehicle to achieve our objectives. This language enables a programmer to develop application faster than the traditional methods [24].
- Compiler construction — The scripting language must be translated into another language that can be used for programming network applications.
- Network programming language — Java language is chosen to be the translated language so that the work can be executed on any platform of the Internet [162].

1.3 Thesis Outline

The research areas in this thesis is spread over eight chapters. Chapter 2 and Chapter 3 presents the investigation and overview of previous research work in networked systems, tools and languages. The detailed discussion on the implemented language is in Chapter 4, while Chapter 5 and 6 covers the main research work. We demonstrate the example applications in Chapter 7 before the concluding remarks in Chapter 8.

1.3.1 Chapter 2: Networked Collaborative Systems

As interaction and interest management are common issues in networked collaborative systems, it is important to review these systems by looking into their concepts, structures and system management. User collaboration can occur at the same or different time and interactive collaboration requires more challenge in managing users activities that often need scheduling so that the work performed can achieve its objectives. Several objectives and example applications are also reviewed.

We examine the interaction management in some existing systems by discussing on its methods, management strategies and implementation techniques. Similar discussion is made on the interest management by looking into several filtering methods that mostly be found in *collaborative virtual environments* [153, 203, 25] and simulation systems [187, 323, 312, 230]. The access control and security issues are also reviewed by discussing on their factors and management strategies.

1.3.2 Chapter 3: Programming Languages and Tools for Developing Networked Applications

Software supports a programmer or system designer to develop networked applications. There are several ways and techniques that can be performed such as the use of software products, software toolkits or through programming. In order to understand the networked environments that can facilitate collaboration, web technology and its related software components are reviewed.

Then, this chapter discusses the development of networked applications that includes communication methods, challenges that need to be considered, available implementation techniques, characteristics of the required environment for application development and type of resources that may be needed. Since JACIE is translated into Java language, the examples of various Java-based collaborative frameworks are included before brief discussions on the scripting languages.

Interaction and interest management in some software tools and programming languages are reviewed on their implementation techniques. Common factors in sharing resources using these tools are also discussed.

1.3.3 Chapter 4: JACIE Overview and Enhancements

This chapter begins by giving an overview of the JACIE language especially its software architecture, main features, and the language structure for managing collaboration. Since JACIE is built on top of Java programming language, the code translation in made using Java compiler tools, *JFlex* for lexical analysis, and *JCup* for parsing, with many Java classes and methods to support the language translation process. In managing collaboration through message passing, the overview of a client/server communication is given with a flow diagram of the states of collaborative sessions and some code segments on the message passing activities.

The enhancements on interaction and interest management as the major extension to the language are given in details in Chapter 5 and 6, respectively, so this chapter discusses on the enhancements made to improve the language on other related issues such as race condition, supporting code optimisation and printing compiler messages on detecting illegal actions.

1.3.4 Chapter 5: Interaction Management

This chapter provides an extensive study on interaction management based on the noughts and crosses game and its variations. These variations include some other existing board games and several versions of the traditional noughts and crosses games with some modifications on the rules of the game. The main focus is on the interaction and work control access. These games provide a significant comparative study of the interaction protocols

commonly used in networked applications. In addition, a set of formal notations for modelling the spatio-temporal activities in a generalised noughts and crosses game is presented.

This chapter then describes the language constructs on interaction management in JACIE. Types of protocols can be defined in one statement that consists of several option tags. Detailed explanations are provided on all of the options, including several supportive statements together with their respective protocols. All of the protocols can be statically defined, as well as changed dynamically during a session. This chapter also discusses the enhancement made to this language and implementation of timers that are included in the protocol options. A comparative study on the server based and client based timer is also presented.

1.3.5 Chapter 6: Interest Management

This chapter begins with a brief review on the issues of data sharing, filtering methods and data access and security techniques. Then, the interest management in JACIE is presented by discussing the *shared variable* in JACIE II and its management framework. Several examples of networked applications are mentioned to help in designing interest management in JACIE II. These factors include determining the *access rule* and *user list.*

The chapter continues by presenting the language constructs that handle shared variables for the interest management. Several types of statement are presented that consists of the owner's permission statement, read statement with assignment and selection control, a write operation and interest filtering statement. The description on the language enhancements is also presented. Some technical issues in this new design are also considered which include mutual exclusion and the management of the user *permission list* and the *access list.* Then, this chapter provides the implementation of *Secret Switch* (one of the noughts and crosses game) as an example to show how these language designs are tested and implemented.

1.3.6 Chapter 7: JACIE Applications

This chapter gives the example applications that have been implemented for both the interaction and interest management designs. An online Bridge game shows the usefulness of the interaction protocols, while for the interest management, an e-learning group exercise on Simulation of Network Trouble Shooting application is presented. The descriptions include some screenshots and code segments on the testing and implementation of the new language constructs.

Several experiments have been undertaken to show the significance of this research work. Some results on JACIE and its equivalent Java program are obtained to give the overall JACIE performance. A small case study is also conducted that involves several experienced and non-experienced people in programming to test their learning abilities on JACIE and its interaction management features. The experiments on determining the cause of delay in some of the example programs are also provided.

1.3.7 Chapter 8: Conclusion

The concluding chapter presents an overview of the achievements that this research work has contributed to the scientific community. It also provides the author's suggestions for future work.

1.3.8 Appendices

Appendix A describes in detail the variations of noughts and crosses type games that support the discussion on the interaction protocol design in Chapter 5. A collection of the common terms and mathematical symbols that has been used throughout this thesis is presented in Table B.1 and Table B.2 in Appendix B. Appendix C contains token and syntax specifications, which support Chapter 4, 5 and 6.

Chapter 2

Networked Collaborative Systems

Contents

2.1 Introduction 11
2.2 Collaborative Environments 12
2.3 Interaction Management 19
2.4 Interest Management 26

2.1 Introduction

A network is a collection of computers connected by a medium to facilitate information exchange. It consists of more than one computer that normally supports many collaborative activities such as communication, information sharing, discussions and so on. Even though each entity in the system is distributed throughout the network, they are in some way coordinated and appear to work together as a single system.

The technology of computer network was first introduced by the ARPAnet project in the 1969 [330]. The main purposes of its existence were as follows.

- Resource sharing — Distributed resources, either hardware devices or files, can be shared among users.
- Super computing — Multiple computers can be combined efficiently to form a parallel system for solving large computational problems.
- Computer reliability — Data or information can be replicated on several computers for avoiding a total lost in the case of any computer failure.

The networked communication connection started with the telephone line and optical cables are now a common place. In recent years, wireless technology emerged as the underlying technology for ubiquitous systems. Much research has been undertaken to improve the use of network systems in many applications. Communication between users is one of the important issues and interactive collaborative systems enhance the network capabilities. The

interactive networked collaborative system was developed in the early 1990's and it has become very popular since 1997 [172].

Within the networked system, system management is important in providing users the ability to handle their tasks properly and achieve their goals. In this chapter, we review the features and management of the networked collaborative systems and discuss research works that have been done in the area of interaction and interest management.

2.2 Collaborative Environments

The word *collaborate* is defined as 'to work with someone else for a special purpose' [50]. In this research work, *collaborating* is a situation of having several users to work together interactively and remotely via computer networks. Hence, a networked collaborative system is a computer mediated system for interaction between users.

The terms, *collaborative workspace*, *networked virtual environment*, *collaborative virtual environments*, *computer supportive cooperative work* (or CSCW) and many others, have been used to describe a collaborative environment in a networked system. It has also been referred to as *multimedia system* since communication is achieved mostly using multimedia. Whatever terms are used for these systems, all of them perform similar network tasks and only differ in terms of their specific environments [172].

Collaborative virtual environment (or CVE) is a type of networked collaborative environments that gains popularity in much research. It is a system that allows multiple users to engage in a common activity in a networked environment, where users are represented graphically within the environment and allow others to interact through their graphical representation [174, 172]. It is defined by Oliveira *et al.* [81] as the 'virtual reality spaces that enable participants to collaborate and share objects as if physically present in the same place'. They also divided systems into *rendering and graphics* and *communications middleware*. Rendering and graphics cover the display aspect and the communication middleware concentrates on the issues of the user connections and communications.

It is also common that the words *collaborative* and *cooperative* are used interchangeably. According to Panitz [248], collaboration is more on the philosophical aspect of interaction while cooperation is the structure of interaction that accomplishes the goal. In our context, there is no distinction between them as *cooperate* means 'to act or work together for a particular purpose' [50] and it has the same meaning as *collaborate*.

According to the space-time matrix [43] that denoted the characteristics of the cooperative work environment (in CSCW), the working space can be classified by

- Time — The form of interaction whether or not depend on time. The term *synchronous* refers to work that happens immediately as in interactive system, while *asynchronous* illustrates the action that happens at different time or time independent. It can also be named as a *temporal factor*.

- Space — The distance between users whether they are in the same place (*co-located*) or at different places geographically (*remote*). It can also be named as a *spatial factor* [151].

Table 2.1 shows the matrix with example applications that relate to the situations described in [43]. When work is done at the same time and at the same place, it is usually referred to as a *face-to-face* environment. The rest of the environments may require some materials or devices to support the collaborative works [151] and a networked system uses the *different place* environment. In particular, the work in this thesis is concerned only with building *synchronous* applications in networked systems.

	Same time (synchronous)	**Different time (asynchronous)**
Same place (colocated)	presentation support	shared computers, bulletin board
Different Place (distance)	videophones, chat	email, workflow

Table 2.1: CSCW Environment.

In general, people engage in a collaboration whether in face-to-face or networked environments for a mutual goal. There are several advantages of collaboration, which includes

- Increase productivity — *Productivity* is the ratio of output to input [147]. When a collaboration occurs, the users, represent the input, can work together in order to produce a piece of work, an output. In this way, the more users engage in collaborative works, the more works can be produced, as pointed out by Havemann [147]. He undertook research on the effectiveness of collaboration among scientists in their productivity outcome in the period of 14 years, which was measured by the number of research papers publication. The study had indicated that their productivity had increased.

- Gain knowledge — Collaboration allows people to share and exchange information. For example, scientists and engineers can have conferences and collaborative meetings to get news and up to date information for their research [67]. In this way, people learn from each other, share information and gain more knowledge.

- Enjoyment and entertainment — People also get together to play games or be entertained individually or in groups. This is another way of socialising, making friends and enjoying themselves.

Therefore, networked systems, which do not limit the distance between users, increase the opportunity for people to collaborate. They serve the need of people who are looking for a fast, efficient and effective way to reach others or to get information.

Research in the effectiveness of networked collaboration is also carried out to compare face-to-face and networked collaboration [296] and how some devices can influence collaborative activities [318, 154] but the most important factors are how user interaction can be conducted while keeping shared data among participants precise and consistent.

2.2.1 Networked Collaboration

Communication in networks involve many processes (either clients or servers) that need some way to be coordinated. The coordination is usually dependent on the application. In recent years, most applications in networked collaborative systems not only facilitate ordinary communication, but also provide collaborative work that requires users to interact. Time is the main factor that influences the collaborative activities. When collaboration occurs at the same time, we refer to it as *interactive* collaboration. For collaboration at a different time, the term *non-interactive* is used.

Interactive collaboration usually consists of concurrent users whose works may affect others in the system. The main advantage of this system is that the output of the work can be achieved immediately. In sharing data that requires mutually exclusion [287, 310], a control protocol is needed to ensure users' activities are in the correct work flow and all users are treated appropriately to achieve the objectives of their work. A mutual exclusion is a condition where a shared resource can be used by only one user at a time and all other users are excluded from accessing the same resource [310]. Several issues may occur in managing interactive collaboration such as race condition, number of users in the system, type of shared resources used, handling users interaction and shared resources.

In implementing most collaborative applications, the system usually has a distinct number of users in accordance to the system requirements and limited number of resources for reducing the complexities of the system design [260]. However, research has also been done to allow an unlimited number of users to collaborate by introducing *scalability* [223] factors. Interaction management is needed for user coordination while interest management is to ensure data security and consistency.

It is possible to have interactive collaboration that does not require mutual exclusion condition. Some facilities such as whiteboard in e-learning and video in video conferencing do not require any sequence of user controls. With a whiteboard, users can freely sketch or write on the board at the same time. Similar to video conferencing, people can listen or talk at any time they like, since they can see and hear all participants. Therefore, in these environments, the sharing facilities are not limited to any control sequence.

In non-interactive collaborative environment, the medium for collaboration may be the main issue instead of the users coordination since people can collaborate regardless of time. A simple example is electronic mail. The communication does not happen at the same time and a user sends messages to others where the messages are kept in storage for later retrieval and response.

Besides the communication medium, users' activities are usually supported by software systems that are particularly concerned with user connection, interface and security [125] without the need to be concerned particularly with the interaction management, however, interest management is still required.

2.2.2 Collaborative Management

In a collaboration, communication can be between client and server, server to server, and client to client through servers. In this client/server connection, the control of the collaborative activities can be either *centralised* or *distributed.*

Figure 2.1 demonstrates the client/server connection with server mediated interaction and group collaboration. For server mediated interaction in (a), several users can communicate individually with the server in an independent activity or collaboration among users can occur through the server. For group collaboration in (b), users are divided into groups that allow them to work individually and also interact between groups or within the group. The communication between group members must be made through the server regardless of their grouping.

2.2.2.1 Centralised Management

Centralised management is 'concentration of decision-making power' in a single administration [182]. It is the server which plays the most important role in keeping all shared information and managing activities. As the server has full control over the whole system, the data and security control can be monitored accordingly. Its advantages include

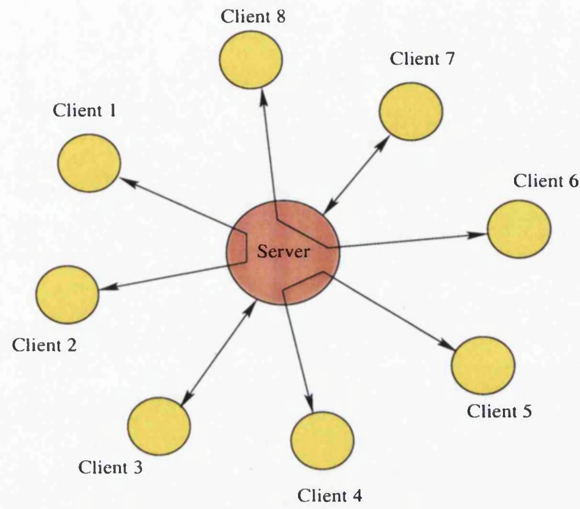
- Data consistency — The management relies only on one source, so data can be kept consistent as data duplication rarely occurs [182].
- Process coordination — It is simpler to coordinate processes when all are under one management. Such a system is less complex compared to having processes under multiple administrators [114].

In the case where there are a large number of processes in a system, such as in a database system, the server is loaded with too many jobs and if the server fails to function, the whole system will have problems [114]. Therefore, centralised control is usually suited well on small scale systems and research on distributed computing has emerged to find the best solution for data management control.

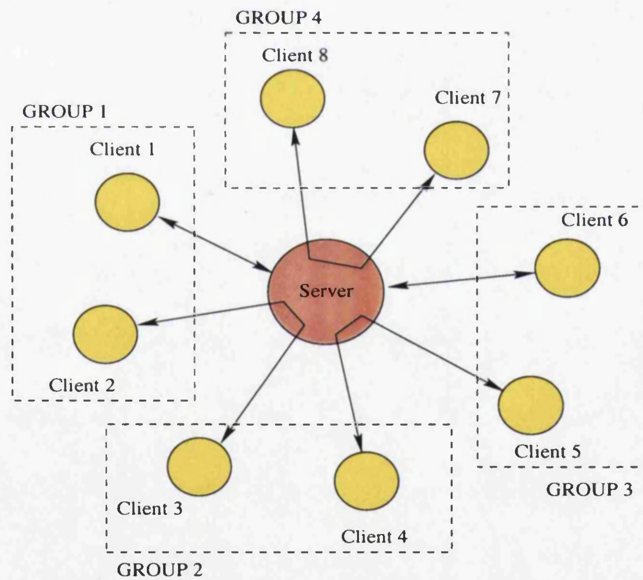
2.2.2.2 Distributed Management

In distributed control, the data and management control can be passed to clients. *Replication* is the term usually referred to data that is copied to a client and the system is called a *distributed system.* Although these systems require critical data communication [93], there are a lot of advantages that include

- Reliability — When data has multiple copies over several components in a networked system, system reliability can be achieved. In this way, if one of the components fails to function, others can take over the responsibilities of the failing component. Therefore, the tasks performed in the system still continue to be processed.



(a) Server mediated multi-user interaction



(b) Group collaboration

Figure 2.1: Client/Server Connections that Allow Users Collaboration.

- Storage — When data and work are increasing, more storage capacity is needed. With distributed computing capability, data can also be kept at clients' nodes without losing the computing power.
- Performance — As most data is kept at the client, the data transfer activities are minimised and the network traffic is reduced. A good system performance can be achieved when the amount of data on every client is equally distributed and no client is loaded heavily or lightly.

Due to these advantages, a lot of research has been undertaken in various systems such as agent systems [308], 3D collaborative systems [135] and database systems [56, 104, 264]. For example, Park *et al.* [251] introduced a scalable data management scheme to replicate a data based on *priority transfer*. Priority is determined by the data location, user interest and the popularity of the data among users. Their work gives positive results that show their mechanisms reduce communication time.

2.2.3 Collaborative Applications

There are a large number of collaborative applications in areas such as e-commerce [305, 210], e-learning [307, 40], entertainment [297, 212, 299], software development [338, 83] and much more. Basically, most collaborative applications support users to serve the following purposes:

- To make decisions — People form a meeting to discuss issues and to reach decisions. Mark *et al.* [215] proposed a system called DOLPHIN that supported online meetings, to investigate the effects of meeting technologies in collaborative style. In the experiment, participants worked in parallel to produce a proposed document. During these collaborative meetings, a virtual chairman can be assigned in the form of an agent [238].
- To create or modify documents — Work can also be done on a shared document. At one time, users can read or write a document that is globally or privately accessed by users and proper access control [268] is needed to ensure all users are obtaining the correct updated documents.
- To share ideas — People can work together on sharing their ideas on a specific topic in various environments such as an educational discussion in e-learning [261] or sharing information using objects in virtual systems [124].
- To get information — Users can gather information online from several information sources, as free access or limited usage depending on the applications. An example of limited access can be in video conferencing where only one speaker transmits data using a media stream [143, 260] or allowing only one person to control a few web cameras for viewing several online videos [78] .

However, most of them involve in non-interactive collaborations or interactive with unstructured activities. The most common goal of collaboration is probably to get or share information among users or groups of users. There are few platforms for structured collaborative activities since they are challenging and require proper management.

With the capabilities of the networked systems to allow immediate response between users at different places, much work can be accomplished and probably some critical problems can be solved in many different areas of everyday life. Such areas include

- Collaborative work — People get together to work on various fields using computers as the communication and workspace. For example, in healthcare collaboration [76], medical colleagues can work together in discussing any medical subject and their work is usually supported by various collaboration tools such as database, images and chat mechanisms. It is also possible to convert rapidly single user tools into shared versions with working back and forth between interactive and non-interactive as proposed in the *Integrated Synchronous And Asynchronous Collaboration (ISSAC)* project [169].

  In business environments where collaboration among people is usually significant, most activities use transactions through database and discussion through video or communicating through e-mail. It is rarely found a structured collaborative activity in e-commerce. However, Lei *et al.* [196] proposed a middleware platform to enable user-to-user collaboration from within *information technology* (IT) business operations that are concerned with data and services.

  There is also collaborative work carried out on document sharing, such as proposed by Sureswaren *et al.* [303] that enables people to share all web based and popular productivity application documents such as controlling power point slides and updating documents. The system adds a new server with unique server ID to accommodate several new users.

- E-learning — The development of networked based distance learning consists of several viewpoints [103] with several forms of learning environments, materials and methods. Hence, some of its applications require structured activities. According to Drira *et al.* [103], who distinguish the interaction levels in terms of cooperation, coordination and communication, believe that each interaction level can be seen as functional, architectural and technological. Functionally, the cooperation is a user-to-user interaction paradigm, while coordination refers to group collaboration at user-level and communication is information exchange between users.

  The interaction in this type of application is either student-student or student-teacher collaboration [261]. The structured activities must occur in the application such as discussion on specific learning content or one teacher to many students learning environment. In contrast, general purpose learning can occur at any time or at the same time but without any specific control on the materials or information access.

- E-meeting — Meetings can be conducted virtually using supportive software or through video with usually one person to chair and determine the floor control. Meetings mostly occur in a business environment [215] for decision making or e-learning where the environment consists of one teacher to many students [261]. Therefore, the turn control policy of all participants is fully dependent on the chair person and the mutual exclusion condition can be guaranteed.

It is also possible not to have a chair person to determine the floor control but rather use a *token-based* control. For example, Chang *et al.* [58] use a kind of data as a token to be passed among participants to indicate the turn control. Whoever holds the *token* is the one who is in control and it is allowable for participants to make token request. Their views on a modern net-centric meeting must contain the following criteria,

- Web-based
- Multimedia environment
- Session and floor control rules
- Formal language approach to defining formal meeting

• Online game — Multiplayer games are available on the internet [255, 163] that enable people to play online interactively. For more challenging and interesting games that require complex design and implementation, users subscription may be required.

There are wide selections of user-to-computer games as well as user-to-user collaborative games. In developing such games, many researchers have brought up issues that include the game presentation [42], user interface and supported device [63], game security [27, 48] and other network issues [126, 228, 256]. Interest management plays one of the important roles since the runtime performance of the game is very important [206].

Nowadays, many games are presented in a virtual world. From a limited number of users to a large scale networked system, there have been many outstanding issues such as scalability, transmission delay and implementation techniques for developing such applications [228]. As pointed by Cai *et al.* [48], interactive games on the Internet need the support from a scalable software architecture with the combination of centralised and fully-distributed architecture. This combination forms a distributed client-server architecture with multiple servers for many clients. With the server mediated services, the game state and communication transmission can be controlled securely.

In building such collaborative applications, several approaches have been presented that rely on agent systems [305, 256], networked programming [107, 46, 301] or middleware systems [199, 334] with the support of database systems for applications that require massive data manipulation [324]. Therefore, the implementation requires knowledge and skill especially on the technical design of system requirements and networked collaborative systems.

2.3 Interaction Management

Interaction is 'when two or more people or things interact' [50] or 'the influence of objects, materials, or events on one another' [100]. In network environments, several issues have been raised around the topic of interaction that include

- Social discourse — It is concerned with social and psychological aspects of communication that are incorporated into the internet design. The internet provides a technological environment in building a cyberspace venue as a medium to discuss social issues [148, 214].
- Human computer interaction — It is concerned with how human can interact easily with the computer, the design of the user interface and devices that people use to interact with the computer [10, 279].
- Work coordination — It is concerned with the handling of users' activities so that users can work together in an organised way [50].

Throughout this research work, the focus is on the work coordination for interactive collaborative environments where interaction is commonly referred to as a *floor control*. It manages user's control right over shared resources within a workspace [96], and allows remote users to share networked multimedia application with some of the following *floor* characteristics [93].

- Mutual exclusive permission
- Dynamically granted to the collaborating users
- Mitigating race condition
- Guaranteeing fair access
- Deadlock-free resource access

In [93], Dommel abd Garcia further pointed out that in managing collaborative activities, there were several required services such as session management, floor control, authentication and synchronisation among mixed media, and in having a *collaboration aware* application, the main focus is to integrate the floor control with session control. Session control supports users in coordinating activity based on a connection management protocol that mediates between upper application layer and relay requests down to end-to-end services. Such services include supporting users to establish a session, and helping users to join and withdraw from a session.

In a session, there are three time-based stages of user interaction that usually require the floor to take effect [93] that include

- Initialisation phase — The time when the floor is created and determines the online users.
- Flow of control — It is based on the designed control policy to determine who is in control of specified resources.
- Termination — When any user withdraws from the collaborative system, it may or may not influence the floor controller depending on the control policy or the application requirement.

In order to collaborate, a user must be *aware* of others in the system. As pointed out by Liechti [201], group awareness can be defined as the understanding of the activities of others within a team. In this way, a user knows what and when to take action [227].

Much research has been undertaken on networked systems such as in collaborative virtual environments [124] and agent systems [254, 85] to provide strategies and techniques in handling structured user interaction. Such research is to serve the following purposes:

- Efficient control — The work flow must follow the specific application demand while the consistency of the shared data is maintained.
- Various applications support — It is significant that a system that supports collaborative applications is flexible in concurrency control. In this way, various applications can be implemented.

As the main purpose is to achieve the efficient control in user interaction, many existing systems have proposed several ways and strategies in different application domains. Although most of them concentrate on one specific application, there are also some systems, which rely on *agents* [70], that are capable of supporting various applications [86], and middleware platforms for system extensibility [124].

2.3.1 Control Methods

A protocol is 'a convention or standard that controls or enables the connection, communication, and data transfer between two computing endpoints' [223]. It has a similar meaning to *floor control policy* or *floor allocation* that establishes clear rules for assigning a user's turn to control a computer resource [95]. Several methods have been introduced that can be categorised into the following.

- Centralised control — Much research uses this policy especially when users collaborate with the support of database systems. With this control, there are two ways users' activities are usually allowed. They are
  - free access : Users are free to access any resource they want, however, a locking protocol is used to block any user if the requested resource is in use.
  - single controller : With a single controller, users can request to have a *floor* and it is up to the controller to grant the request.

  Free access is widely used in most collaborative virtual systems [138, 211, 25, 135] and several other methods and algorithms are proposed on top of the policy to achieve efficient system performance. For the single controller, it is totally dependent on the controller to decide on the control policy such as implemented in e-meeting applications [58].
- Token passing — A *token* is used to determine the user in control and at any one time, there is only one user who can hold the token. The token is usually passed around among users so that every user can have the control over the resources that they want. This scheme appears to be fair to all users and it is possible to add a timer option to this policy to avoid users having a long waiting time.

There are many names referring to interaction protocols such as first come first served [78], round robin [177, 78], free floor [177] or contention [94], central moderator [177] or reserva-

tion [94]. These protocols are either controlled centrally or are token passing as mentioned above. More policies are proposed by Kausar and Crowcroft [177] that include *explicit release* and *pause detection*, which all of these are the customisations of the original turn taking that require user in control to react upon event or time. They also propose another policy called *pre-emptive scheme* with the priority factor being enforced as one of floor control policies.

2.3.2 Management Strategies

Since managing the interaction protocol is challenging, several approaches have been introduced. Some systems propose architectural-based design while others use models, both approaches usually are considered to have several components with specific functions to guarantee the flow of work is smooth and communication between processes is properly handled.

There are several system architectures proposed to provide collaborative frameworks. In order to handle interactive structured collaborative work, most systems depend on the server to handle the floor control activities. Thus, with the centralised control, proper management can be achieved. For example, Wang *et al.* [324] propose a client-server architecture where all user events must be sent to the server first. At the server side, it has *virtual world manager* that includes a virtual 3D world, scene manager, object manager, consistency manager and event manager. Besides the virtual world manager, there are also a session manager and server manager. Therefore, it can give scalable, persistent and consistent control that results in excellent performance for 3D collaboration.

Sureswaran *et al.* [303] also introduce a client/server model for distributed network architecture that has three characteristics, *chairman*, *presenter* and *participants*. The *chairman* controls the overall session, which consist of all users (*participants*), and the user in control is the *presenter*. The system provides document sharing to users. The collaborative work takes place with inter server communication while the system allows the growth of users by adding extra servers for several new users.

In particular, to maintain the collaborative session interactively, processes are 'tightly coupled', which have explicit member registration and are governed by a formal agenda [94]. The services provided by the system components can be

- Layered — A layered host-based service in an architecture allows the dynamic organisation of users in a multilevel control tree [95]. Within this architecture, the system is not only able to support interactive user-to-user collaboration, but also allows more users to enter the system. The newly added users can form a group under the control of one new host [303, 95].
- Integrated — In managing user interaction, the system is divided into several components in order to organise the system to control user coordination and the communication mechanisms. Each component has some distinct functions that can be integrated and linked during the on-going session. For example, Abdel-Wahab *et al.*

[2] introduced a system architecture that was divided into three major components namely *session server*, *session control manager* and *event controller*. The *session server* handled the floor control that offered several policies that consist of *request-and-get*, *request-and wait* and *no-floor policy*. With the *session control manager*, a user could request and release a floor, call, join or leave a session. The *event controller* controlled the collaboration between a *sender* and one or more *consumers* that are associated with the server and one or more clients.

While the layered service is significant, the management of many users with a lot of data requires database support or file systems management. Some systems may provide only *short time* collaboration in conjuction with non-interactive collaboration [268]. In addition, the system can usually provide a limited choice of interaction protocols such as using prediction, filtering and reservation in a virtual system [94] or locking protocol in sharing documents [303].

It is common that most systems use integrated services, dividing the systems into at least two components that consist of

- Floor controller
- Communication handler

In the CWCE System [190], there are three main components, the job generator, environment coordinator and resource provider. The main aim of this system is to provide agent-based applications for distributed computing. For granting a resource, a resource provider will follow the *first-come-first-served* (FCFS) strategy. Therefore, a process must wait for its turn to get the resource. Another agent, the job manager uses a load distribution mechanism called *adaptive highest response ratio next* (AHRRN) to maintain a queue of all jobs waiting to be transfered to the resource provider.

Another system proposed by Lei *et al.* [196], called *contextual collaboration framework*, the architecture consists of a component namely *contextual collaboration platform* for managing structured activities in business applications. The component has three items; contexts, collaboration space and structured activity. Context is referred to as objects or any aspects related to business environment, while collaboration space is a container for business elements and structured activity specifies the roles in the collaborative processes. Within each item, there is a manager to fully control all activities.

A system implemented by Joslin *et al.* [173] allows multiple users to collaborate using 3D graphics presentation based on loading and managing scenes for connected users. They proposed an architecture for special communication and task management that relies on *threads*. Therefore, a *thread manager* is one of the important components of its architecture. The characteristics of its management system include

- Buffer management — It allows unlimited data transfer for communication but the data flow control is done using *first in first out* buffer.
- Thread management — The threads are created not according to the number of connecting clients, but rather according to the task performed. Hence, a priority value

can be set to the tasks for faster processing.

With the *thread manager* that acts like the floor controller, this system also introduces a buffer manager as part of the communication handler.

There are several models proposed for interaction management strategies with several ways for handling group collaboration. Pinelle and Gutwin[253] developed a model called *Collaboration Usability Analysis(CUA)*, a task analysis technique designed to represent collaboration in shared tasks for the purpose of carrying out usability evaluation of groupware. Although their work focused more on the groupware usability factors, the designed model was based on the mechanics of collaboration that were the basic operations of teamwork. The mechanics covered two general types of activity that were

- Coordination — It concerns the *shared access* that includes tools, objects, space and time, and *transfer* specifically for handling objects.
- Communication — It handles *explicit communication* and *information gathering* during the collaboration such as the communication interface and awareness of participants.

Other work by Shih *et al.* [283], proposed a multimedia presentation system based on *Petri Nets Model* [233], that used the relation between places, transitions, input and output places of transitions. Its floor control provided *free access* where users can freely send their messages to the server, *equal control* that limits only one user with specified timed token to have control, *group discussion* to allow users to create new discussion group and *direct contact* that allowed two people to have private collaboration. These protocols were used for distance learning application that allowed senders to share a whiteboard, perform general discussions, tools sharing and private communications. It is quite similar to a model presented by Chang *et al.* [58] that used Coloured Petri Nets [189] and focused on the hierarchical order system with similar floor control policies to apply to collaborative meetings.

Other models for handling interaction management were proposed by Fantar *et al.* [113] that introduced floor control algorithm on *Session Initiation Protocol* (SIP) [223]. Their work focused on managing access to multimedia data streams in a video conferencing system. In the model, each client SIP was connected to a server proxy SIP which had connection to all other servers' proxy SIP in the system. The floor was a token attributed to participants for holding the control on resource access and also handling other participant's floor request.

2.3.3 Implementation Techniques

Many of the above networked collaborative systems are implemented mainly for specific applications. While some systems rely on sophisticated database or agent systems, there are also systems built upon programming languages and tools. Details on these languages and tools are discussed in Chapter 3.

2.3.3.1 Database System Support

Database systems usually handle massive amounts of data. Collaborative virtual environments where collaboration is usually reperesented by graphic images, require a lot of data to be managed. For example, a system called *Windows Virtual Life Network* (WVLNET) uses a server database for clients to upload their information. In this way, the server manages the clients' activities [173] and data consistency can be guaranteed.

Although such collaborative systems can support many users at any one time, in managing interactive collaboration, only a few selected users can form a group collaboration [135] with the issues of scalability and concurrency control. The concept of scalability 'is to reduce the message exchange as much as possible in terms of number and size without harming the shared context and interactive performances' [191] while concurrency is to maintain synchronisation on replicated data on clients. According to Yang and Lee [332], in distributed virtual environment that concern sharing *objects* between users, concurrency control is categorised into three schemes,

- Pessimistic — A user is blocked until a lock request for an object is granted before the object can be manipulated. It is the simplest mechanism that guarantees data consistency can be achieved. The disadvantage of this scheme is that users may suffer from a long response time when the number of users increase.
- Optimistic — This scheme allows users to update objects and interact naturally, but a repair must be done when conflicts occur. Therefore, the system is more complex and users have to undo and redo some actions [191].
- Prediction — It uses the optimistic scheme with the attempt to eliminate the need to repair by having the owner to predict who the next user for ownership transfer request.

In general, database systems do not provide any specific interaction protocols but rather use the mentioned concurrency control schemes to ensure data consistency on mutually exclusive condition.

2.3.3.2 Agent System Support

In multiagent systems, *agents* are used to perform the coordination of the involving parties. *Agent* is a 'software program that intelligently performs its duties without human interaction' [244]. It is also defined as 'a piece of autonomous, or semi-autonomous proactive and reactive computer software' [223]. As pointed out by Bergenti and Ricci [34], due to the challenges of providing the proper interaction protocols, it is a long standing issue in the agent community to come up with the distinct and proper solution. By far, most agents are involved in non-interactive communication or providing a specific protocol based on some model [34, 190]. In addition, they also pointed out that the meaning of interaction in agent's coordination activities seldom mixed the role of coordinating with computing on performing certain tasks. Hence, they further proposed some approaches to clearly distinguish the agents' roles.

Theoretically, Demazeau *et al.* [85] have proposed basic interaction protocols between agents in controlling vision systems based on several models, where the agents must be capable of reacting to incoming messages they received and also know what to expect after sending a message in an activity. In this way, the global system control can be achieved in terms of its possible and desired behaviours.

Practically, the *Augmented Multi-party Interaction* (AMI) project, is an example of research on multimodal interaction that supports online virtual meetings [238]. The meeting is conducted in a smart environment where agents are used to help in handling information for each participant and there is also a *virtual chairman* for coordinating the meeting session.

2.3.3.3 Algorithm Based Design

Interaction protocols can be programmed using any general purpose language to build specific applications such as multi-user multi-camera environments [78], interactive video with media channel rotation schemes [260], collaboration with java applets [107] and distributed multimedia in distance learning [283].

Fantar *et al.* used algorithms to handle floor control in a video conferencing system to work on the *Session Initiation Protocol* (SIP) that provided mechanisms to initiate sessions, allow other users to join and leave or ask third parties to join. SIP also supported session management and data exchange capabilities with minor help in access control. Therefore, the proposed algorithm would extend coordination between participants for video conferencing [113].

Most of the work supported by algorithms rely on the server to control user coordination. In implementing such systems, a programmer must have good skills in network programming and excellent computer science knowledge to design such challenging work.

2.4 Interest Management

The secured sharing of information in networked systems must follow some rules to ensure that the shared data is precise and consistent. In a collaboration, some of the shared data are only relevant to some people. For example, in sharing a database system, a few users can read or view the information, and only a certain number of them can do the updating. At any one time, only one user is allowed to do any changes to the particular data. Within such systems, a user may not want to know all the information kept in the system. Therefore, there is a need to filter the data for clients and such a process is called *data filtering*. As pointed out by Barrus *et al.* [25], the reasons for having data filtering are to serve the following purposes

- Reduce the system complexities.
- Increase system performance.

While there are many ways that data can be filtered, user interest is the most commonly found in networked collaborative systems. Many of these systems handle massive amount of data that some way is needed to scale it down [191]. Most systems focus only on the filtering mechanisms without concern for user access control or data protection since the systems can rely on the database system that supported them or the system can use simple locking protocols for concurrency control in updating the database [138, 211, 25, 135]. Therefore, these systems usually serve for specific application such as multiplayer online games [336, 206], teacher to students e-learning [124] and training or experimental testing using simulation [230, 224, 312]. In these systems, people also share data or information during collaboration that shows the need to include some mechanisms for determining user access for data consistency and security for user privacy.

2.4.1 Data Filtering

In networked systems, data filtering refers to reducing the amount of irrelevant or unwanted data transfered to clients [5, 217]. In group collaboration, the frequency of message exchange is less which results in minimising network traffic and reducing the burden on clients [217] in receiving unnecessary messages. Data filtering is also implemented in parallel systems to reduce data transfer for efficient computing power [180]. Other networked systems such as collaborative virtual systems [203, 205, 153], agent systems [298, 263] and database systems [53, 217], also implement data filtering based on user interest.

2.4.1.1 Generic Filtering Strategies

There are several strategies proposed to handle interest management in large networked systems. They include the division of

- Region — Filtering of messages can be performed by the users location that can be divided into several regions. In this way, messages are sent to the specified regions which are referred to as a group of users with the same interest. Since the network covers a very wide area, it is also possible to divide the region logically for not only achieving efficient communication, but also for reducing the complexities in managing a lot of users and many activities as implemented in Spline (Scalable Platform for Large Interactive Networked Environments) [25]. Several terms have been used to describe the region division such as locales [25], sub-region or sub-domain [217].
- User — The user interest can also be determined by grouping all the users in the system into several groups. The users are usually allowed to acknowledge the server their existence and can make known their interest so that they can be grouped according to the information given to the server [217, 53]. It is also possible to group them according to their role/identities or the tasks that they are working on. In this way, users are grouped and work with others of the same interest.

Spline [25] is one of the leading collaborative networked systems that supports a very large virtual world. It is implemented by Mitsubishi Electric Research Laboratories (MERL). It is a 3D graphics presentation containing objects that are divided into 62 locales. Diamond

Park is one of the virtual landscapes implemented using Spline. Users or park visitors can interact while undertaking some activities in the park. They can even converse verbally when they are near each other. Therefore, although logically the system has many locales, it is hidden from the users and the system allows smooth transition within the application [25] .

In the research of Masa and Zara [217], spatial division is adopted allowing users to define their interests using the concept of room division in the application. The implemented system is aimed at social interaction in virtual environments with *avatars* used for communication and selection of activities.

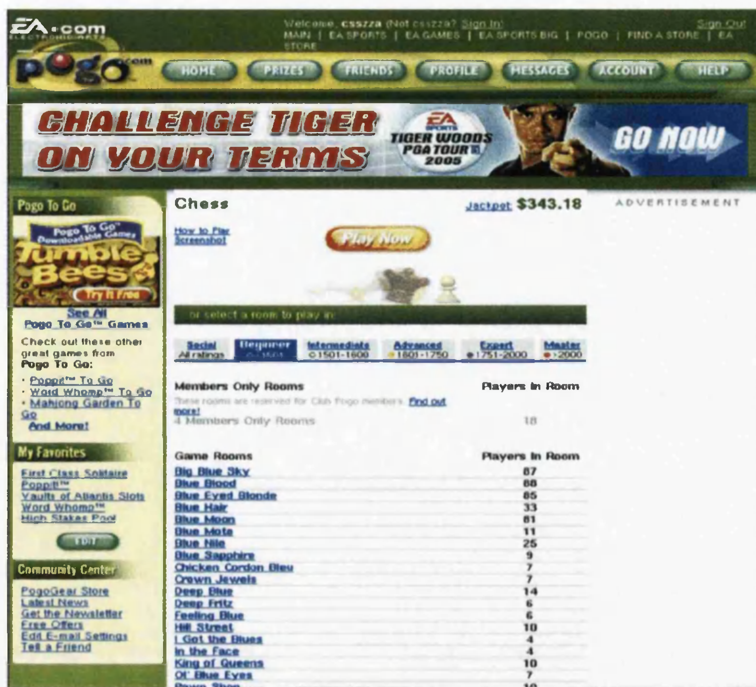
It is common in some Internet applications to use the concept of room to determine user interest. For example, at *www.pogo.com* as shown in Figure 2.2, many users are grouped by rooms. The rooms are originally divided according to the difficulty level of the match. Within a room, there are tables where players can select their opponents. In choosing the match, they can decide to play with or without timers. Users can also log into the system as observers. This example illustrates the need to divide users into smaller groups for managing structured interaction and interest management.

Another system called CLOVES (City-Level Optimisations for Virtual Environments) [53] groups users according to user selection on specified functional groups in an application interface. The system that serves as data, navigation and communication infrastructure for the City Scanning project, allows users to launch applications to join and leave functional groups determined by the application.

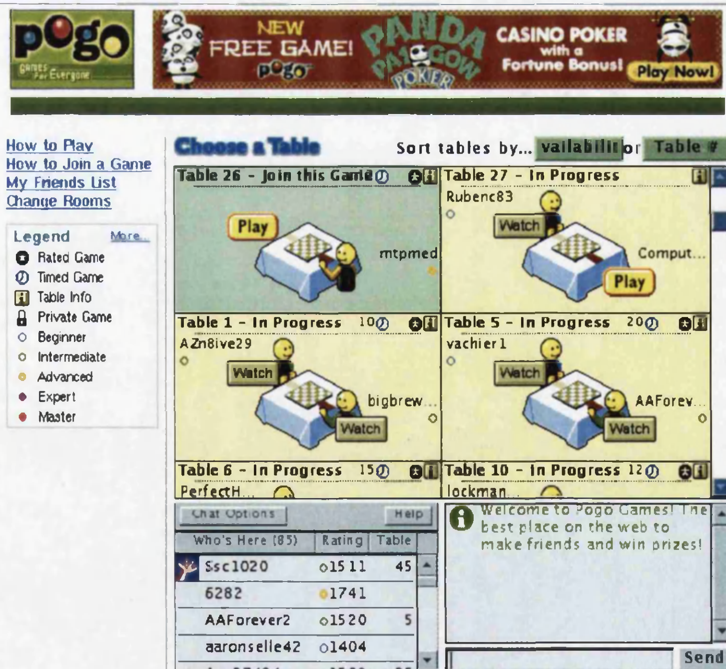
According to Zou *et al.* [341], in networked multi-user games, there are two basic grouping strategies,

- Cell-based — It is a region division based on any shape, normally squares, known as cells. Each cell is assigned a *multicast* [223] group address and there are two sets of cells associated with each entity for information sending and receiving sets.
- Entity-based — A single multicast group for an entity where the entity multicasts all information within this single group. Therefore, in order to collaborate with other entities, group intersection should be determined by the entity's coordinate and multicast group address.

With the assumption that the 'playing area' is presented in two dimensional space and divided into cells, an entity is only interested in receiving data from other entities within its vision domain as shown in Figure 2.3. These two grouping strategies are not particularly for game environments, but can also be applied to virtual environments in general. As pointed out by Liu *et al.* [205], the entity-based strategy requires distance comparison between avatar and it is adopted by small scale virtual environments while the cell-based strategy requires tracking the avatar's position and can be adopted by large scale systems.



(a) Rooms Division



(b) Player Selection

Figure 2.2: Pogo Game Collaboration Chess.

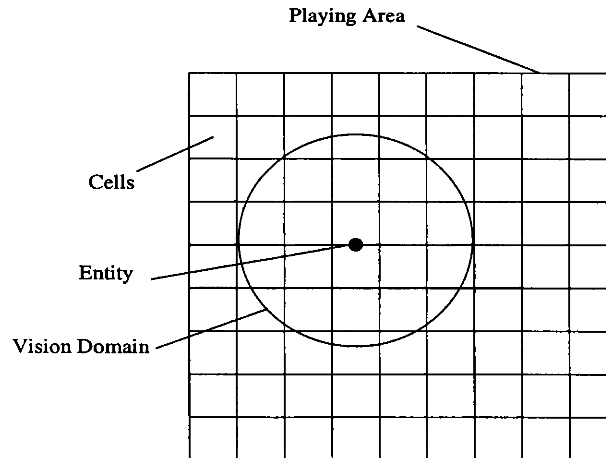


Figure 2.3: Entities, Cells and Vision Domains [341].

2.4.1.2 Filtering Factors Based on User Interest

Many filtering strategies are implemented based on users interest. *Area of Interest* (AOI) is the method used mainly on the server side to filter message [203]. AOI and *user awareness* in CSCW systems have similar approach that regulates the amount of information to be processed by each user. *Awareness* means an understanding of the activities of others, which provides a context for one's own activity [98]. In awareness management, suppressing the irrelevant information that must be processed by each user [13] shows the need to filter information in collaborative management.

User interest is also found in other research areas such as data mining [186] and *information retrieval* (IR) [66]. Data mining 'is the process of automatically searching large volumes of data for patterns' [223] while information retrieval is often related to a specific object searching and query [223]. IR is a broad interdisciplinary field and automated information retrieval (IR) systems were originally used to manage information explosion in scientific literature [223]. Information retrieval and *information filtering* (IF) concern on the same objective, that is to get information [29]. As pointed out by Raje *et al.*, recently information filtering has started to attract attention in information management research that involves repeated interaction over multiple sessions with users having long term goals compared to information retrieval which only satisfies the users' short term information needs [263].

There are many different ways to determine user interest that include the followings.

- Distance — Determining a user's *area of interest* (AOI) can be found by aura-based [217, 229, 228] or zone-based [336] approaches. In the aura-based, AOI basically consists of three elements, focus, aura and nimbus [134].
  - focus : A sub-region that defines the actual focal point of a user or the area of space that a client is interested for a certain medium.
  - nimbus : A region where a client can be detected or known by others.

- aura : A region where the awareness of an object becomes enabled and usually contains both the focus and nimbus for purposes of determining interaction.

In this approach, either the server or client itself can determine the aura based on data position. Several approaches have been introduced to determine a client's aura such as predictive modelling of object movements [229], algorithm-based AOI calculations (i.e. collision detection algorithm [211, 206] or clients' interest matching [188]) and visibility-based awareness measurement [153].

While aura-based approaches use clients' awareness and focus, zone-based approaches raise several issues, particularly on the number of users in each partition [172]. For example, in a networked virtual system, where the virtual area is very large, it is important to divide this workspace into several smaller zones because at one moment in time, a user usually focuses on a certain zone in the system. While the user is working, he/she can move from one zone to another. In this way, a user interest can be changed dynamically. Therefore, it is important to manage the number of users and related objects in a zone so that none of the zone is suffered from neither work overloading nor idle [82]. Thus, it influences the system performance.

- Frequency — In data mining [186] and *information retrieval* (IR) systems [66], user interest, usually referred to as *user preference*, can be indicated by the frequency of information search. In a system such as SHARK (SHARing Knowledge), a document-sharing multi-agent application [298], agents are used to identify and group users before analysing and categorising them according to the documents that they match the most. In this way, such agents not only help in determining user interest, but also the user's future document searching and downloading can be speeded up. These agents can also promote collaboration among users with similar interest.

  Another system, COBioSIFTER [263], has similar approaches to determine user interest based on information search frequencies for biological data. With the similar concept of frequent user activities, interest can also be based on the activities that occur through user interaction in a system.

  Ding and Zhu [91] describe a 3D virtual environment, called MultiVR, where user interest is represented by user's interaction with several objects. In such a system, some algorithms are proposed to analyse the behaviour of any interaction activity that results in the achievement of finding the focus of interest.

- Predefined setting — User interest in an application can also be determined by allowing the user to provide a certain value. For example, in CLOVES system [53], a user is allowed to join or leave a certain functional group that is previously set according to the grouping procedure. Selecting to join a particular group means that the user has determined the interest value.

Almost all of the related work presented here that concerns the data filtering based on user interest show the significance of interest management. However, as most work applies in large scale systems, interest management issues often focus on only the reduction of data exchange between clients or message passing in the communication systems. For interactive

group collaboration, such as collaborative games, interest management can allow users to engage in such preference activities in the form of small groups. In these environments, although some data or information must be shared, there is little focus on integrating interest management with user access and data protection except that mentioned by Belkin and Croft [29] on information sharing that is mainly supported by agents and database systems but without any consideration of user collaboration.

2.4.1.3 Filtering Implementation

As there are many strategies and approaches in filtering of data in network systems, these filtering schemes are implemented in several ways that include the followings.

- Simulation technique — Much research on interest management is performed through simulation. Concern is mainly with system efficiency and model testing. In fact, interest management is one of the most significant concepts in distributed simulation [187] with the aim of reducing the number of messages passed in various applications such as military games training [230], software design [187] and mobile objects or agents [312, 323].

  Liu *et al.* [205] propose grouping strategies using algorithms that apply to entity-based and cell-based groupings. Their work focuses on avoiding the calculation of finding visible sets and reducing the complexity of updating groups, when an entity changes its position or AOI. As a result, the grouping scheme overhead can be reduced and only necessary messages are sent to the certain groups.

  In the use of *Distributed Interactive Simulation* proposed by Messina *et al.* [224], interest was managed, either server-based or router-based, where system entities had to define their presence and activities to their local CPUs via messaging. Their work had concluded that interest management was very significant in achieving system performance since one critical issue within the system was the growing number of entities.

  In agent-based distributed simulation as proposed by Wang *et al.* [323], the interest management for *High Level Language Architecture* (HLA) [223] supports two types of filtering

  - Class-based filtering : A service that allows a user to update and receive updates to object attributes based on object class.
  - Value-based filtering : A service that extends the update service using routing and regions.

  With routing spaces or multidimensional coordinate systems, users' interest are determined by subscription or updated regions. Using algorithms to calculate the intersection between these two, connectivity and efficient data transfer can be established.

- Virtual Environments — Interest management is also one of the important issues in virtual systems that are mainly concerned with data transmission delay in user motion, action and communication [53], minimising network traffic and reducing the burden

on clients [217]. These systems are usually supported by database systems for users' information that apply to various applications such as online games [341, 336, 228], e-learning [124] and shared virtual worlds [53, 217].

For example, a net-VE system for social interaction and culture content dissemination called e-Agora [217], combines spatial models and functional filtering interest management and comes up with a formalised storing and distributing updates namely *General Variable* (GV) concept. The GV for a particular user is kept in the GVs database and this system also uses *Virtual Reality Modelling Language* (VRML) technology. In another shared virtual world system, CLOVES [53], the spatial subdivision is used a database with centralise information storing and the system introduces a *Graduated Visibility Set* (GVS) for its interest management techniques that also combines the spatial and functional filtering.

- Agent-based data filtering — Although agents are usually required for information retrieval in information management systems, it is also possible for agents to filter data according to users' interest in order to help people in obtaining the information they need in a faster and more efficient way. SHARK [298] and COBioSIFTER [263] are example of such systems.

Although much research has been undertaken on interest management and several techniques for data filtering have been implemented, there is no attempt to provide basic and simple interest management for small scale interactive group collaboration through high level language constructs that can filter message exchanges between users for a general approach in various networked collaborative applications.

2.4.2 Access Control

Access means 'the right to use' [50] that refers to mechanisms and policies to restrict the use of computer resources [70]. Upon being granted system access, particular data or information may be protected for privacy. *Security* refers to the 'techniques for ensuring that data stored in a computer cannot be read or compromised by any individuals without authorisation. Most security measures involve data encryption and passwords. Data encryption is the translation of data into a form that is unintelligible without a deciphering mechanism, while a password is a secret word or phrase that gives a user access to a particular data' [70]. It may occur that there are several access levels for sharing data or information in collaborative systems.

Access control on shared data needs to be enforced for consistency and sensitivity. Tolone *et al.* [317] proposed access control requirements that contain several elements as follows:

- Distributed platform — The control access platform must support distributed features as resources can reside in distributed places.
- Generic models — The model can cover the needs of a wide variety of tasks and models and be able to support various information backgrounds.

- Greater scalability — This refers to the number of operations supported. The greater number would be better.
- Strong protection on information and resources — With massive data sharing, the need for strong data security with different levels of access can be achieved.
- Flexible authorisation — A clear distinction for authorisation must be supported for easy data manipulation and consistency.
- High-level specification — The better managing of access in conjuction with the complexity of the applications.
- Dynamic models — Access policies can be changed and managed at run-time.
- Reasonable costs and performance — The costs would be kept at a reasonable level as well as achieving good system performances.

The above requirements are based on models that are supported by software tools for *Grid environments* that are dynamic, distributed and usually only support a short duration of group collaboration [92]. As *Grid computing* is more concerned with speed and storage, it is not particularly good for handling interactive group collaboration, but rather for coordinating the sharing of resources among a massive number of users [236, 292]. Therefore, security is one of its important issues.

In small scale interactive group collaboration, all the elements in the access control requirements listed above can benefit the design of interest management that integrates access control with data filtering. Apart from the 'greater scalability' factor, high level systems such as middleware systems or special purpose network programming languages are capable of providing generic and dynamic models with flexible authorisation and high level specification for information protection in distributed platforms at reasonable cost and performance. However, by far, there is no such tools to provide these features.

2.4.2.1 Access and Security Factors

In some systems, access control can be viewed as the same as security control while other systems differentiate these considering access control as a more general system access while security applies to specific data or information items. As access control can be part of a security system, here, both approaches can be seen as having a similar purpose. In general, several factors has been introduced in networked systems that include the following.

- User — The user is the key component in a system that needs to get access to any shared data. As pointed by Chen *et al.* [62], it is important to separate the users and their data into different disjoint sets of security classes, so that both sets use different secret keys. The users can be arranged hierarchically in order for proper management of access control.

  In grid computing, role-based access is widely used to enable dynamic access rules for a user during a session that allows a user to change role or have multiple roles. This model is called *Role-based Access Control* (RBAC). It starts by having a centralised

administrator [115], and then is enhanced to a distributed hierarchy that broadens the role of the server as well as client with the support of locality [192, 59, 31]. Further enhancements have been made to RBAC. For example, combining RBAC with XML [59] for multi-domain environments [171], adding the concept of team for *Team-based Access Control* (TMAC) in collaborative environments [315], putting other desirable behaviours (such as time and location) for *Context-based Team Access Control* (C-TMAC) [127] and *Temporal Role-Based Access Control* (TRBAC) that allows security to be applicable during runtime [36] and to fulfil complex security needs in *Dynamically Administered Role-based Access Control* (DARBAC) [219].

In Xie *et al.* [331], a framework for security integrates users' access rights, users' history behaviours and resources protection. Therefore, users' behaviours are tracked and then according to these behaviours, secure levels are calculated and users' access rights are assigned. This work illustrates dynamic access control mechanisms.

- Event — It is important to have an access based on events as proposed by Bhide *et al.* [38] in e-commerce environments that can reduce customer response time. The access control rights are based on policies that are executed every time an event occurs. Such mechanism is supported by a database system and the validation rules stored in a policy database.
- Knowledge hierarchy — In most systems that are supported by databases, users are interested in access to the knowledge or information in such systems. The access is usually performed by *SQL queries* and there are rules and regulations set using a *Knowledge Description Language* [32]. The concept of hierarchy in knowledge discovery allows the knowledge to be expressed in higher level abstractions. However, it requires complexities in the implementation that should consider the SQL primitives, indexing and encoding of rules and much more [32].
- Location — Location is an access factor especially important for mobile computing. In such environments, information is location-dependent while the location service is general purpose [197]. The implementation of such systems also require supports from a database system and the location of involved parties can be partitioned using a hierarchy where access control is determined according to the user location domain [198].

As the above factors apply to general access control methods, mainly individuals in particular, collaboration of groups has special requirements for security policies as pointed out by Ellison and Dohrmann [108]. They address the need of each group member to have the same access rights and before the group is formed, it is important to be aware of each user's authorisation rights. Therefore, each member of the group is identified by a public key.

According to Tseng [319], in group collaboration that requires one sender to multicast data to a large number of authorised receivers, it is necessary to have a scalable key-management scheme. The group members usually share an encryption key that needs to be renewed if changes in the group members occur. It is also common to have a hierarchical tree structure of users for easy management and cost reduction in this scheme.

2.4.2.2 Management Strategies

There are a few strategies proposed by researchers in different environments to manage data access in accordance to the need of applications or system requirements.

Using the proper design of a system architecture, the system can be managed systematically with clearly defined functions for its components. For example, *Closed Collaboration Teams* (CCT) [92] for business transactions supports the use of *public key certificate.* The environment has several entities that consist of *local security administrators*, *team managers* and *team members.* By the breaking entities, the system supports dynamic formation and self-management of virtual collaborative networks.

The management of data access can be based on the basic access control model, a matrix model, first formulated by Harrison *et al.* [146]. It is the relation between subject and object where an authorisation is expressed according to the access rights and access modes [274]. The matrix model has strong features that have been adopted by many researchers in determining authorisation decisions [340]. As pointed by Zhang *et al.* [340], in this model, there are three elements,

- A set of objects — All the entities to be protected in the system (passive entities).
- A set of subjects — Set of active entities that can request access rights or execute some permissions on an object.
- A fixed set of commands — A condition or sequence of primitive operations that can change the state of a system.

Example access models based on this matrix model include *schematic protection model* (SPM) [275], *type access matrix* (TAM) [276], *usage control* (UCON) [274] and *attribute-based matrix model* (ABAM) [340]. Most of these support modern access control that has dynamic system states.

In MASSIVE-3 [135], there are several consistency mechanisms for data items within its database. The consistency model features the following characteristics

- Single owner of single data items at one time for ensuring data consistency.
- Two part sequencer or counter for each data item for defining an unambiguous order of all updates.
- A list of item sequencer values in every message for indicating event enactment.
- Allowable ownership request for a data item to ensure unambiguous order of ownership transfer.

With the proposed model, the concept of mutual exclusion, centralised control of the data owner and distributed management with the ownership transfer has been proven to support interaction and interest management in networked collaborative systems. Further description on the state update is shown in Figure 2.4. Figure 2.4 (d) is similar to the JACIE interest management design in terms of the ownership transfer concept and centralised control (de-

scribed in Chapter 6), but there are many differences in the concurrency control, shared resource types and application implementation. In ownership transfer, the master, who is the owner of the shared data receives the ownership request from a client. When the ownership is granted to the client, the client can perform as many updates as required by the application provided that only the current owner can do the data updating. Figure 2.4 (a) shows this concept on the first time data update, while Figure 2.4 (b) shows the subsequent updates performed by the client after getting the ownership. With the centralised update in (c), every updating event must be acknowledged and sent to the master that can cause delay to the observer, therefore a *Collaborative Immersive Architectural layOut* (CIAO)-style [302] update (Figure 2.4 (d)), which combines the ownership request and data update in one event, reduces the delay. The numbering at all the arrow lines in Figure 2.4 represents the sequence of events during the update process.

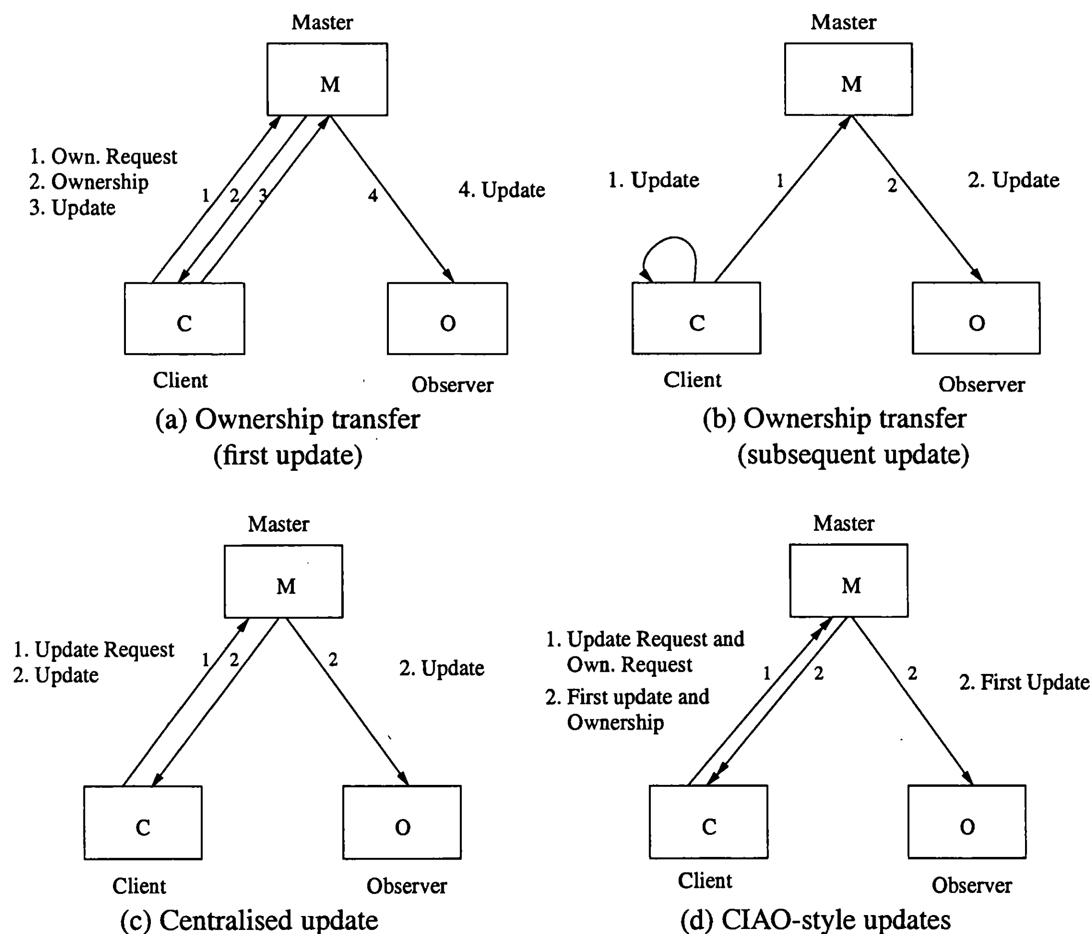


Figure 2.4: Consistency Mechanisms in MASSIVE-3 [135].

2.4.2.3 Sharing Factors

As data can be built from many types, the term data sharing may cover a lot of topics in collaborative systems such as sharing of variables, objects, components or knowledge. The

variable sharing may be similar to memory sharing and sharing of information or knowledge is usually applicable in database and agent systems.

There are many kinds of resources that people share in the networked systems. When several users engage in an interactive collaboration, there are a few types of resources that they are typically interested in, including the following.

- Memory — Sharing of memory is similar to sharing of variables as a variable is always represented by a high level symbolic name. They refer to addresses in computer memory [258] that data exchange is performed by operating system services [223]. Therefore, throughout this discussion, both terms are interchangeable. The shared memory is widely discussed in concurrent and parallel systems as memory is one of its important factors in providing faster computation [180].

  Parallel systems are usually composed of homogeneous processors that either share a global memory, accessible by all the processors in the distributed systems, each processor has its own memory. Although the concept of shared memory can be centralised or distributed, the way communication is performed mainly depends on the system hardware and programming. This gives rise to issues of network connection (or topology), distributed operating systems [288], virtual memory management, multilevel caches [60] and parallel programming languages [258].

- Object — *Object* is 'a thing, an entity or a being' or in a 3D model, it is 'a representation of physical object' [223]. In interactive collaborative virtual systems, sharing a 3D graphics object, has given rise to several issues that include transmission rate [220, 202] and data management [128, 213].

  A prototype system called TeCo3D (TeleCooperation 3D) [220], a virtual reality model, allows distributed users to view and share a single-user application. Users can share interactive and dynamic 3D models over a distributed architecture while leaving the sharing mechanisms performed by the application rather than the 3D-model itself. The application is equipped with customised collaborative sensors for user interaction and some programs as interfaces to support its main tool, a VRML. Its 3D-model application dependent uses generic approach so that it is possible to reuse this system for various applications.

  Another example is the Shared Simple Virtual Environment (SSVE) [202], an object-oriented framework for highly interactive group collaboration. It has rich features for interaction and information sharing that can support only a small number of users. Data consistency is achieved by combining atomic method exclusion (operating system support) with transaction locking using object properties implemented at a high level. This framework is implemented using simulation that runs a single user mode and has proven the efficency of group collaboration through subgrouping and regrouping mechanisms.

  Other work by Geyer *et al.* [128] use object-centric sharing to support collaboration activities. The main target is for lightweight activities that allows users to aggregate and organise shared objects into activity threads. It also allows users to move seam-

lessly back and forth between different modes, interactive and asynchronise systems. Using peer-to-peer shared objects, this research is aiming to fill the gap between ad hoc communication and formal collaboration.

- Database — Sharing information in a database system is very common in distributed systems nowadays. It benefits a lot of people in accessing the latest information available. It is common to have a distributed database system with distributed data management. A database system can be used to support applications with massive data. For example, in multi agent systems, a database is used for knowledge sharing and in concurrent systems, especially parallel systems, a database is used for data and resource sharing. The sharing of information in e-commerce relies on a sophisticated database system for everyday transactions. It is also possible to achieve the integration of several database sources. On top of that, as pointed out by Agrawal *et al.* [8], it is possible to have information integration from autonomous entities with minimal sharing based on more than one DBMSs. Their work examines sharing the integration of only a required part of the systems and hiding the rest of the information. Another system, HERMES (HEterogenous Reasoning and MEdiator System) [52] also provides a platform for global security policy in mediated systems that allows the integration of database systems with different local security policies.

  The main issues in sharing database are security and privacy. The locking protocol [264, 150] is one of the popular ways to ensure data consistency in updating information. In the distributed database, replication is required for consistent data maintenance. For example, Navas and Wynblatt [235] describe and implement the data management by borrowing the internet concept since the network is the database.

  There is much research on handling the database management and access methods as the database becomes more complex. For example, Cavazos *et al.* [56] propose a 3-tiered client-server component-based architecture that provides a systematic and secure execution sequence for data operations and schema operations. Another example by Niemi *et al.* [237], proposed a powerful and advanced query language for manipulating complex entities.

- Device - It is also possible to share devices, peripherals or instruments remotely and collaboratively. Although it may not be easy to share one or more instruments between one or more users, several investigations have been undertaken by either limiting the number of resources, such as using only a single robot through multiple sensors [132] in controlling robot motion, or limiting the number of users, such as in controlling one or multiple cameras [78], or applying simple protocols such as locking mechanisms in video control [260] and agent support for robot vision systems [85].

Although sharing resources in networked systems may include other entities such as files and documents, these resources can be represented as objects or variables when they are implemented in networked applications.

Chapter 3

Programming Languages and Tools for Developing Networked Applications

Contents

3.1	**Introduction**	40
3.2	**Network Programming Languages and Tools**	41
3.3	**Interaction Management**	54
3.4	**Interest Management**	56

3.1 Introduction

Software supports communication and interaction among users in networked systems. It is defined as a term for various kinds of programs used to operate computers and related devices, and also considered as an intermediary between electronic hardware and data [329, 223]. Software that contains system software, programming software and application software, supports programmers and designers in several different ways to achieve their objectives. System software often requires low level programming in which only a programmer with specialised skills and experience is able to write a networked application. Many programming languages have emerged nowadays to assist in a much easier approach for the development of different types of applications. Furthermore, clients can now communicate through *web pages* that can be considered as documents written in *Hypertext Markup Language* (HTML), which provide great flexibility for display and interaction. Such documents can be supported by other back end programs such as *Common Gateway Interface* (CGI) programs for further development features.

There are also several tools specifically designed for networked applications that help users

to reach others, share information or work collaboratively. Software products such as *Lotus Notes and Domino* [158], *Basic Support for Collaborative Work* (BSCW) [118], *Microsoft Netmeeting* [73], and *Novell GroupWise* [239] are also available usually to assist users in general collaborative activities. This type of software system is also named as *groupware*. Since most of these products serve users in general purpose collaborative work, programming tools may give better opportunities to assist users in having the specific required applications. Such tools usually provide software developers with convenient programming environments. Example of such tools include text editors, scripting languages [269, 325, 161, 101] and toolkits [281, 193, 257, 88]. However, in using such tools, some basic knowledge of network programming and web technology [322] are still required.

Interaction and interest management are important factors in implementing interactive networked collaborative applications. As pointed out by Shirmohammadi and Georganas [284] that at the application level of sharing multimedia applications, there are two main factors in multi-user environments: consistency of data and application access control. The former factor is indirectly related to interaction management while the latter factor is directly related to interest management. Since data is the most important element that people share in collaborative applications, interaction control protocols guarantee data can be kept consistent among users by allowing at most one user to change the shared data at a time while interest management helps in controlling particular users to access particular parts of an application.

This chapter discusses network programming languages and tools for developing networked applications by looking into the developmental features and implementation issues of the existing systems. Interaction and interest management issues are also reviewed.

3.2 Network Programming Languages and Tools

Since software can cover many topics in networked systems, from high level languages in implementing web pages using web technologies to low level languages in network operating systems, only an overview of such technologies, system design and implementation is discussed. Regarding the issues of programming languages used in the implementation, Java [105] is one of the commonly used high level languages. It has been used in many existing collaborative systems for its capability of running on any platform that has a *Java Virtual Machine* (JVM) [162]. It is also the base language used in this research work where the JACIE scripting language is translated. Java has the technology to insert its applets into HTML documents, mostly used as client programs and allows server programs to be executed as application programs. Therefore, this section also gives an overview of some of the existing Java-based collaborative systems.

There is no progamming language that provides all types of applications demanded by networked users, therefore, several types of languages and tools have been developed. Although Java and .NET framework have provided popular platforms for general purpose programming, there is also the need to handle special purpose applications that require such languages or tools to provide powerful mechanisms and simpler approaches. For example,

scripting languages have became popular for their 'shift in application mix toward gluing applications' [246] that would mostly be found in web pages CGI scripting. There are also *component* [223] based tools for software interoperability.

3.2.1 Web Technology

Web technology is about the *World-Wide Web* (WWW) that was 'developed to be a pool of human knowledge, which would allow collaborators in remote sites to share their ideas' with the original target application to support collaborative work [35]. It can also be considered as the implementation platform and activity space. Therefore, it became the most convenient way to access information on the Internet with WWW browsers to integrate different network services into a common, easily accessible and platform independent user interface [123].

In WWW technology, the *HyperText Transfer Protocol* (HTTP) is a protocol used to transfer information from a specified address in *Universal Resource Identifiers* (URIs) (or also referred to as *Uniform Resource Allocator* (URL)) [35]. Although HTTP is not intended for real-time data feeds, CGI programs allow users to interact with the web server as well as its active contents such as Java applets, to make interactive group collaboration possible [123].

Most application developer utilises WWW as the main vehicle to deliver a collaborative application using several programming languages such as C++ and Javascript. To design a group collaboration in the WWW, Gall and Hauck [123] highlight the importance of network topology, which can clearly distinguish client and server subsystems in designing a system architecture such as a *client/server model* [234, 329, 289], and the need of programming models for communication, concurrency, floor control and session controls. JACIE also uses a client/server model in managing group collaboration by supporting any user activities through a server where the server initiates a session and allows a finite number of users to join a session. However, a programmer can choose to program either centralised or distributed applications since both server and client components can support communication and session control.

A client/server model is one of the significant concepts in networked systems. It is a general description where a client program initiates contact with a server program, which is usually located on a different machine, for a specific function or purpose. The client exists in the position of the requester for the service provided by the server [234]. Although the client/server model can be used by programs within a single computer, it is significant to use this model in a network environment. This model provides a convenient way to interconnect programs that are distributed across different locations [329]. In this model, one or more clients can interact through a server, along with the underlying operating system and interprocess communication systems, form a composite system allowing distributed computation, analysis and presentation [289].

According to Chun and McLane [65], programming in a client/server model uses the basic concept of multi-tier architecture where each tier has its own unique web-related technolo-

gies, languages and software. To them, the architecture consists of

- Presentation — Concerned with the way information is presented at both client and server.
- Application — It focuses on developing 'business' applications which are associated with web servers and application servers.
- Data access — Concerned with the database handled by database server.

The work in this thesis has the most focus on the application tier that plays the most important role in controlling the underlying process of applications, and some concern with the presentation tier that inserts Java classes into HTML documents for displaying applications at clients' web pages and supporting clients activities. JACIE has no database support, so it is not concerned with the data access tier.

In the application tier, server-side programming can be performed on HTML embedded language as well as high level programming language using Java classes, JavaBean or *Component Object Model* (COM) [223], and JACIE chooses to use Java classes in all its applications.

3.2.2 Networked Applications Development

Nowadays, there are various networked applications available to fulfil user demands in many different types of systems such as parallel and concurrent systems [218, 180], virtual systems [45, 202, 124, 243, 212], agent systems [256, 166, 37] and database systems [264, 8, 282]. Research on improving software technologies is still an on going process so that the following goals can be achieved.

- Ease of use — A software developer looks for an easy to use tool or simple language to design and implement applications.
- Flexibility — It is important to have a programming language or tool that can provide a platform for various types of applications.

Issues on developing networked applications especially in *distributed systems* [69] have been discussed for some time that involved operating system support where, initially, only operating system provides interprocess communication [293]. Network operating system programming is mostly concerned with the work of processors that were connected by a communication network. Several high-level distributed programming languages had been developed to support program execution on these distributed processors [22].

When looking into the history of high level distributed programming languages, most languages during that era were sequential languages and some were concurrent languages based on shared variables such as Concurrent Pascal [144] and Modula-2 [278]. In executing these languages, each processor had its own concurrent program that 'many operating systems for uni-processors are structured as collections of processes, executing in a quasi-parallel mode, and communicating through shared variables' [22]. Other concurrent languages like SR [12] and Ada [249] used shared variables as well as message passing to increase concurrency.

Message passing had became widely accepted although in this approach the name of the message receiver's exact location was required. Various schemes to name the receiver's location have been proposed that enable such destination points to use either 'direct naming' (exact address), *ports* [223] or *global names (mailboxes)* [70]. Later, a *remote procedure call* (RPC) [293, 223] was introduced to enable a client to make a 'call' to a remote machine to support at least two messages exchanged in a client/server interaction [11].

As hardware and software technologies advanced, *distributed operating system* provided data access regardless of the receivers' locations [133]. Therefore, these primitive approaches towards shared variables, message passing and remote procedure calls had gone through several phases of developments. Such developments provide a programmer with techniques for writing networked applications by ignoring the details of communication processes. For examples, higher level abstractions such as *Application Program Interface* (API) [162, 223] packages and abstract memory representations [258, 179] were proposed to support application development from high level programming languages [293].

Nowadays, there are several general purpose network programming languages such as Java [162] and C# [84] to provide a software developer with such design and implemention. Although such languages still require some programming skills and network knowledge, their general features make them able to provide platforms for different fields of applications. There are also scripting languages such as Python [325], Perl [269], Tcl [160] and JavaScript [71] that are widely accepted for their simple and easy programming features using *scripts*. These languages are mostly used for CGI scripts in HTML documents to enhance user interaction [145] and several other specialised languages or libraries such as Mawl [16], DiCons [18] and Curry [145] have emerged for greater flexibility for the CGI applications.

3.2.2.1 Method of Communication

Clients are in distributed places in a network and communication between them must be made through servers. According to Bal and Tanenbaum [22], two way interaction between two processes can be performed either using *Remote Procedure Call* (RPC) or *Rendezvous* [252], while interaction between one sender to many receivers can be done through broadcasting and multicasting. Broadcasting enables information from one source to be transmitted and received by all others in the network while multicasting allows information output by one source to be received by a specific subset of others [142]. The sender can use communication ports or mailboxes for the receiver's location to avoid explicit addressing of processes [22]. Therefore, to enable users' collaboration, data at one location must be passed to others by one of the following methods.

- Message Passing — Communication by message passing requires message queues for both sender and receiver. It is usually in the form of a loosely coupled system [47]. In some languages, such as Hermes [17], message passing is performed by language constructs. The general form of message sending is as follows.

```
send <expression_list> to <destination_point>
```

In this language construct, expression_list indicates the values to be transmitted while the destination_point can have one or more points or *nodes*, that denote the destination of the message to be sent.

A process receiving a message from a source point is usually represented by the following statement.

```
receive <expression_list> from <source_point>
```

Similar to sending a message, expression_list has the same function as above while the source_point usually represents only one source per message [11].

- Remote Procedure Call — It uses the concept of procedure calls in a structured language such as Pascal [183]. This concept provides the flexibilities of other programs or routines from one section in a program (or other user in the network environment) to use other routines with or without parameters. Therefore, one or more values in the original routine can be copied or modified.

  The primitive RPC approach was represented by a 'call statement'.

  ```
  call <service> (<value_argument>; <result_argument>)
  ```

  The service indicates the source node's name if it is referred to the place of service, or otherwise, it can represent the name of a service if the 'abstract' address of source node is used. Both call 'parameters' denote the value to be sent to the requesting node and the value to get in return [11]. By far, the RPC mechanism has changed since the *object model* [22] is introduced. The object model gives greater flexibilities in communication methods where a parameter value can be referred to as an object. Now, it has a mechanism of transferring control between the original to the calling routine that facilitates an easy communication between virtual nodes.

The issue of message passing is often found in parallel systems where *Message Passing Interface* (MPI) [119], a computer communication protocol, is a de facto standard for communication among the nodes running a parallel program on a distributed memory system (i.e. clusters of *Symmetric Multiprocessors* (SMP) [223]). MPI implementations consist of a library of routines that can be called from a high level general purpose programming languages such as Fortran [112] or C [194, 223] that usually involve buffers, byte counts and data types. Another example is 'MPI Ruby' [245], a scripting language that integrates MPI with the object-oriented language Ruby [321], to achieve a simpler programming approach.

The RPC mechanism can be configured in several different ways that allows calls to be made within the same machine or different machines, as shown in Figure 3.1. The figure illustrates two machines, M and M', where machine M has two processes P1 and P2 while machine M' has only one process P3. Each process can contain one or more threads, therefore, RPC can be performed between threads in one process (denoted by RPC1) or between processes within one machine (RPC2) or remote machine (RPC3) [293].

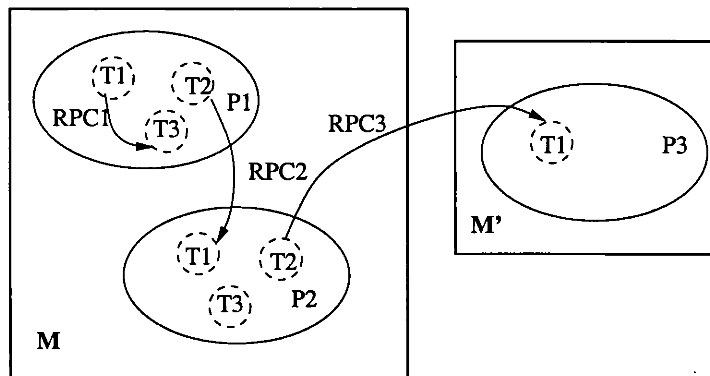


Figure 3.1: Possible Configuration of RPCs [293].

3.2.2.2 The Challenges

Since a network can consists of many heterogeneous computer systems, constructing a programming language for such environments is challenging. The emergence of generic platforms such as CORBA [286], DCOM [72] and JVM [162], together with the use of general purpose network programming languages can support higher level software design such as scripting languages and toolkits to help designers in building interactive networked collaborative applications. Scripting languages and toolkits have provided a simpler way for application design compared to the use of general purpose network languages. Languages are often based on Java and C, such as Python is on C language and Yoix is on Java. Examples of group collaboration toolkits include DistView [257], Suite [88] and *Collaboratory Builder's Environments* (CBE) [193]. As these toolkits are useful, the *language approach* design may give greater flexibility for developing various types of application.

For example, Figure 3.2 shows a session manager in DistView [257], a toolkit for building collaborative applications [241]. In this session manager, users are grouped by a high level grouping mechanism using a *room model* where users in the same virtual room can collaborate with each other. Users can perform their work on both private and shared workspaces and they are able to move tools and data between these workspaces. This toolkit also supports mechanisms for access control with password authorisation on users who can choose to be registered users or 'guest' users.

There are several other challenges to provide a software tool to a system designer in implementing networked collaborative systems. These include multiple media channels for various forms of communication, the collaborative environments over the Web from multiple platforms (e.g., Java-enable Web browser) and the mechanisms for access control to ensure consistent shared data [193]. These factors enable collaboration among users from many different places to be performed in many different ways while maintaining the consistency of shared data among them.

Another challenge, apart from the application development issues, is probably the network transmission delay that can affect the user response time. A brief discussion on this issue is

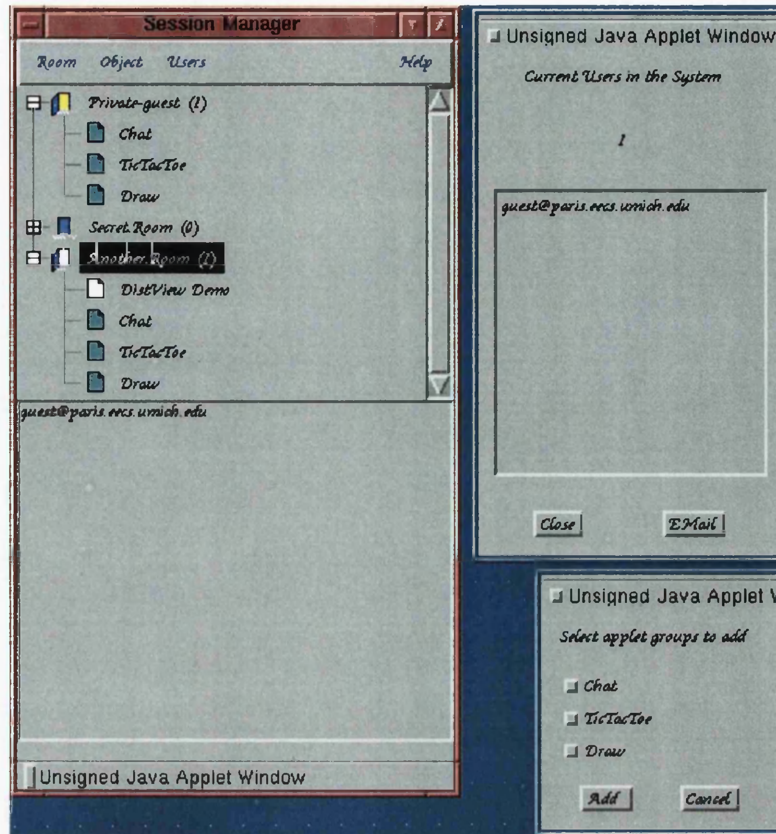


Figure 3.2: Session Manager in DistView [241].

provided in Section 7.5.3. However, further details will not be provided since the focus of this research work is more on the design of language features rather than the examination into overall system performance.

3.2.2.3 The Techniques

Network protocol and communication are usually performed by operating systems since all machines connected in a network are controlled by this system software. However, nowadays, these jobs can be managed at a high level through several methods. Application Program Interface (API) [162, 223] is the most common approach in almost all programming languages that support programming for networked applications.

Application Program Interface (API) is a set of routines or tools for building software applications. It usually takes the form of a library of routines that enable the higher level programming languages to make use of the lower level features. In the traditional operating systems, only assembly language can use the APIs to instruct the operating system kernel. However, when the software technology improved, a programmer or even a computer user can use API from a high-level language to activate some features in the modern operating system. For example, the C programming language is used in writing the operating systems

such as UNIX, was designed with the API facilities that makes programming system programs much easier and convenient. Other examples include the Orca programming language [20] that works with the Amoeba operating system [240], and Ada for low level message passing communications on BIGSAM Distributed Operating System [225]. In this way, network protocols can be performed at the user level using application program interface (API) or user libraries [21, 225, 314].

With the API approach, there are several object-based component system [223] platforms provided to integrate heterogeneous systems. The most common platforms are as follows. [265]

- *Common Object Request Broker Architecture*(CORBA) — This represents a standard architecture to support object interoperability on networks, produced and maintained by the *Object Management Group* (OMG) [286]. 'It defines APIs, communication protocols, and object/service information models to enable heterogeneous applications written in various languages running on various platforms to interoperate' [223].

  CORBA applications are composed of objects that are invoked by CORBA clients using *interface description language* (IDL). IDL interface definition is independent of programming language, but 'maps' to all of the popular programming languages via OMG standards. Such programming languages include C++, Java, Cobol, Smalltalk and Ada [286]. IDL uses simple syntax that enables a call to a software component while hiding the detailed implementation such as a component running code and data. It not only allows object interoperability between different programming languages, but also inter-machine architectures. The operation is achieved through the concept of remote procedure call [223]. The CORBA approach has become the most popular in many networked applications because of the varieties of languages that it can support.

- *Distributed Component Object Model* (DCOM) — This is a Microsoft technology for software components and extends the Component Object Model (COM) [72] allowing COM components to communicate across network boundaries. Traditional COM components can only perform interprocess communication across process boundaries on the same machine. DCOM uses the RPC mechanism to transparently send and receive information between COM components in the Microsoft Windows-family of Operating Systems. For example, ActiveX Controls [73] which have the API concept are widely used for software development tools and end-user productivity tools [70, 223, 72].

Both CORBA and DCOM have different architectures and use their own functionalities that enable these platforms to work together. For example, in CORBA, communication between clients and servers is through *Internet Inter-Orb Protocol* (IIOP) [286], while in DCOM, this communication is performed via RPC [73]. Since CORBA can work on any system but DCOM is only for Windows environments, several attempts have been made to enable DCOM to have cross-platform interoperability such as Microsoft's *Simple Object Access Protocol* (SOAP) [73, 80], and the proposal for an inter-working framework for CORBA and DCOM namely Active COM [79].

In addition to CORBA and DCOM, *Remote Method Invocation* (RMI), developed by Sun

Microsystems Inc., is another object-based component platform. Like DCOM, it is also specialised for one environment that is Java. It 'enables a programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines' [162]. With the concept of procedure call, it is in the form of an application program interface that is compatible with CORBA.

3.2.2.4 The Characteristics

Since a networked system can be viewed as several connected remote computers to form one 'large' system, therefore, in providing users with shared application, software tools must consist of the following characteristics.

- Concurrency control — This refers to simultaneous execution of multiple interacting computational tasks that occur either in separate programs or as a set of processes or *threads* in a single program [223]. This required feature is necessary to accommodate with the computing environment of networked systems. Some languages such as Java, C and Concurrent Pascal [144] support such feature in the form of language constructs that usually can also be referred to as *concurrent programming languages*. These constructs may involve multi-threading that are implemented in several ways including message passing and shared memory. Other languages such as Ada [249], Erlang [111] and Oz [272] use application program interfaces that have interprocess communication support for network operating systems.

- Communication Support — Programming languages and tools can provide 'comfortable' environments for users at high level where usually such tasks and activities are handled by the operating systems for actual inter-user communications. A *microkernel* [223] often provides communication primitives in some systems while other systems rely on high abstraction level such as reliable message passing and RPC. There are also systems that use a microkernel such as FLIP [176] in the Amoeba distributed operating system, to build higher level communication protocols in the form of libraries in user space [240].

  In user space, several protocols have been implemented such as *Transmission Control Protocol/ Internet Protocol* (TCP/IP) [131] and *User Datagram Protocol* (UDP) [270]. While TCP/IP uses socket point-to-point connection for establishing user communication that guaranteed data to reach its destination, UDP is reliable for broadcasting and multicasting data [132].

In particular, most programming languages that support distributed systems have the above characteristics. In addition, languages such as Ada and SR, also support failure detection that is a significant reliability factor in distributed systems where such systems can continue functioning properly even though failures have occurred at some parts of the systems [21].

3.2.3 Java-based Collaborative Framework

Java [105] is a technology developed by Sun Microsystems [162] for machine-independent software purposely for Internet computing. It encompasses of the following items.

- Java programming language — An object-oriented high-level programming language that allows a programmer to write a powerful, 'enterprise-worthy' programs that can run on any machine that installs JVM.
- Java Virtual Machine — A virtual machine that runs Java byte code. It is a program that interpretes the Java programming language.
- Java Platform — A 'software-only' platform that runs on top of other 'hardware-based' platforms. Since hardware platforms can vary in many aspects such as storage and network connection, there are several specialised platforms to accommodate these differences. These platforms include the followings.
  - Standard Edition: This consists of *Core Java* and Desktop Java application environments as well as Java Web Services. The main target is for desktop environments.
  - Enterprise Edition: The standard for developing component-based multi-tier enterprise applications with the target being server environments.
  - Micro Edition: A set of technologies and specifications for consumer and embedded devices.
  - Java Card Technology: Java platform for smart cards and other intelligent devices that have limited memory and processing capabilities.

  Hence, many types of applications can be developed for networked systems since the specialised platforms are able to provide such programming environments. In addition, each platform is based on Java Virtual Machine [223, 162].

Although Java is often compared to .NET [313], both have their own specialities for their target applications and have provided software developers 'convenient' environments for programming networked applications compared to using low level programming with the support of network operating systems. While Java uses varieties of platforms to cater for different hardware bases, .NET allows the integration of several types of programming languages for a standard runtime environment called *Common Language Runtime* (CLR) [15].

There are many Java-based frameworks for interactive collaborative systems, which use capabilities provided by this language. For example, *Java Collaborative Environment* (JCE) [2] allows a single user Java application to be shared through data packet exchange using socket connections. Its system architecture is divided into several components that handles events, session and user interaction activities for providing several users to collaborate. JCE develops these collaboration mechanisms by extending Java components with a new package called 'collawt'.

Another Java framework called MultiTel [121] also hides the details of the underlying tech-

nologies and the complexities of collaboration patterns from its designer by offering reusable software components. It supports a distributed, compositional platform that manages multimedia and networking resources in the design of complete multimedia collaborative services. It allows collaboration between components and runtime composition of different multimedia products, chats or GUIs [121].

JCell [267] uses the same concept of component and further integrates it with the concept of module, to provide a software designer an internet language that is built on top of Java. With this concept called 'cell', cells expose typed linking interfaces that may allow import (*plug-in*) and export (*plug-out*) classes and operations. It uses extensions of the Java syntax for the language and allows integration of objects to support local and distributed service innovations. Although this language has provided flexible system integration, a programmer still needs the specialised skill in network programming.

There are several other Java-based framework for specific systems such as agent systems [190], mobile systems [232], collaborative virtual systems [324] and WWW [107, 123, 284]. For example, in the *collaborative Web computing environment* (CWCE) system for agent-base applications, Java-based distributed computing is used to guarantee fair resource utilisation with global computing performance, using an object allocation mechanism [190]. Similar to mobile computing such as Sync [232], it uses the provided Java classes to develop applications based on object-oriented replication. Java 3D also facilitate designers to implement virtual systems with static and dynamical virtual environments [324]. Some systems that use VRML utilise ordinary Java object classes to integrate VRML with database systems in building virtual systems.

For the WWW, user collaboration can usually be achieved using Java applets that are embedded into HTML documents and there are also scripting languages built on top of Java in order to facilitate simpler programming environments that are used for CGI. Promondia [123], is an example framework for interactive group communication over the WWW. It consists of a server program and session starter implemented as Java applets called session-management applets that are embedded into HTML documents to offer text-based chatting, shared whiteboard, voting and surveys, and games for a small number of users.

Java has rich features on network capabilities, supports independent platforms and has provided network programmers varieties of application program interfaces (API) in designing and implementing networked applications. Therefore, on top of Java, there are several other language-based or component-based approaches. JACIE [139] is the example of language-based approach while Jcell uses the component-based approach in application design. The objectives of such 'new' languages are to provide application developers with simpler or flexible development processes. Example of other language based approach in the form of scripting languages include Yoix [102], Iava [266] and DiCons [18].

Yoix is a general purpose scripting language that uses syntax similar to the C programming language. Although this language is interpreted into Java code, the language itself is not object oriented and is able to provide some access to most a standard Java classes [101].

Iava is an embeddable interpreter for scripting within a Java platform that provides full access to Java classes and APIs. It accepts a subset of the Java language itself, Iava and Java source code can 'migrate' to each other without rewriting [266].

DiCons (Distributed Consensus) Language [18] uses Java servlets in implementing a package of classes to specify different parts of the language. Its basic constructs consist of *users and roles*, *interactions*, *behaviour*, *presentations* and *data*, which are capable of supporting asynchronous small group collaboration, with the main purpose to provide easy tools in developing networked applications.

3.2.4 Scripting Languages

The idea of using a script in the computer field comes from the UNIX world with the term 'shell script' introduced in the 1970s [24]. The popularity of scripting language has arisen as they are easy to use programming languages that allow programmers to develop applications much faster compared to the traditional methods. This type of language can be embedded within HTML to add more functionality to a web page. Therefore, the web page can become more dynamic. On the client-side, the language affects the data in the user's window browser, whereas, on the server side, concern is mostly with manipulating a database. Perl [269], Python [325], Tcl [160], VBScript [161] and JavaScript [71] are examples of such languages. [24].

Scripting languages have been used in many different ways due to their rich functionality and ease of use. They are often used in system programming such as UNIX system administration [116], CGI scripting for web pages as well as to interconnect diverse pre-existing components to accomplish a new related task. In fact, they can be found at almost every level of a computer system and cater from simple computer tasks to complex computer applications [223]. For example, the *REstructured eXtended eXecutor* (REXX) language [157] is normally used for job control as shell scripts, STEP [155] provides a platform for virtual environments based on agent technology using VRML/X3D, *Hypertext Preprocessor* (PHP) [68] supports HTML server side technology that is mostly applicable to database-driven application [333] and SLIS [1], an Ada-based script language is used to implement simulation applications.

According to Barron [24], these languages have some common features that include the followings.

- Compile and run integration;
- Easy to program;
- Strong functionality; and
- Not much focus on the language efficiency.

Many existing scripting languages are interpreted. Some of them operate on an immediate execution basis, while others are implemented as 'strict interpreters' where operations are performed when a valid keyword or construct is recognised [24]. There are also compiled

languages available. Most of these languages use simple statements that allow a program to be written much simpler compared to usual programming in conventional languages. Some scripting languages are typeless with no data types to be declared [262]. For functionality, some languages can enhanced functionality in some area such as allowing easy access to low level operating system facilities. They can also be programmed without much focus on the language efficiency, but the developers' needs are often met [24]. Therefore, there is a huge selection of scripting languages available that serve either general or specific purpose application development.

Table 3.1 lists some of the scripting languages used for developing networked applications. In the table, all languages are classified according to some of their features. JACIE and Yoix [102] are languages built on top of the Java programming language to help a programmer to develop applications without being concerned with the technical details of network programming and object manipulation in Java. Although both languages utilise Java capabilities, both differ in several perspectives.

- JACIE is a special purpose language for developing interactive collaborative applications, Yoix serves as a general purpose language.
- JACIE uses Java classes for application-based design in web technology while Yoix is meant for client based CGI scripting that supports HTML embedded documents.
- JACIE uses a compiler compared to Yoix that uses an interpreter.

Language	Purpose	Compiled	HTML embedded	Other features
JACIE	Special	Yes	No	Built on top of Java
VBScript	Special	Yes	No	ActiveX scripting host
Perl	General	No	Yes	Interpreter written in C
Python	General	No	Yes	Support C/C++ extension
Tcl	General	No	Yes	Every statement is a command
JScript	General	No	Yes	Object without class
Yoix	General	No	Yes	Built on top of Java
DiCon	Special	Yes	Yes	CGI extensible
Php	Special	Yes	Yes	CGI extensible

Table 3.1: Comparison on Scripting Languages.

Comparing JACIE and VBScript, which both feature special purpose languages, the difference is in their language platforms where VBScript is for MicroSoft using activeX technology. The rest of the scripting languages in the table are for HTML embedded documents either general or special purpose languages. In classification of these CGI extensible languages, some of them can be programmed for both server and client programs, while language such as (PHP) [68] only support server-side technology.

In general, specialities of the language features in using scripts have attracted many software designers to propose numerous of scripting languages that can be used in many different computing fields. As one language is not necessarily better than the other, each language has its own strength to achieve the needs of an application. These specialities also encourage

this research work to have a scripting language as the project background.

3.3 Interaction Management

Designing floor control policies have raised several issues such as determination of control sequence, fairness to users, the length of time for a control, mutual exclusive permission and many more. As pointed out by Boyd [41], floor control policies in multi-user applications can be classified into several dimensions that include the following.

- The degree of interaction — Can be either automatic with entirely 'application dependent' without any user input or 'interactive' that allows users to react to the application for turn request and release.
- The extent of user characteristics influences a policy — A policy may be uniform with providing 'fair control' to each individual or allowed to have a 'master' that represents a single dominant role to determine the policy.
- The granularity of control — The policy implementation may cover the whole application or some parts of an application.
- The duration of control — A policy may be for a long term or short term basis.

Considering some of these mentioned dimensions, such issues are related to how collaborative applications are implemented. Some toolkits such as CBE [193] and JASMINE [107], allow interactive user interaction through applets and utilise Java for their framework base. With the similar concept to protocol reservation in JACIE where users can have interactive floor request, these toolkits usually can provide limited protocol choices.

Most existing collaborative frameworks provide a dominant or master role especially in developing e-learning application for teacher/student environments [124] and implement protocol contention, which they call 'free access' [138, 211, 25, 135]. Since both floor policies, master and contention, may not be fair to users, protocol round robin is usually implemented in some systems [177, 78], which they call 'token based', that allows every user to have equal opportunity of holding a turn.

The last two items mentioned above are concerned with the flexibility of setting the floor control and the inclusion of time factors in the policy design that influences user waiting time. These factors can be manipulated in some of the toolkits mentioned above through selecting a specific 'button' on the user interface. In comparison to JACIE, the approach of offering such protocol customisations is still quite limited and with little flexibility in implementing applications.

3.3.1 Implementation Tools

Although interaction between users can be viewed as having the support of some existing systems such as database systems [173] and agent systems [238, 85], it can also be im-

plemented using software tools such as commercial products [158, 118, 239] and toolkits [281, 193, 257, 88], as well as programming on language-based [102, 139] or component-based [267] facilities. While software tools and toolkits may provide software developers with a simple approach but limited protocol selections, programming can allow them to develop complex applications with more flexibility in the implementation techniques. Therefore, application development can be performed using software tools and programming either by a language approach or component approach.

3.3.1.1 Using Software Toolkits

There are several techniques implemented by toolkits to provide user interaction in collaborative systems. The implementations are based on several approaches that include API-based framework applets such as in Jasmine [107] or Jets [284] and JavaSoft's Java shared toolkit (JSDT) [162], additional abstract window toolkits such as in JCE [2], new created applet such as in Habanero [320], Windows system protocol such as in Netmeeting [74] and additional software libraries in the MVL Toolkit [242]. In most of these toolkits, floor control policy is performed using contention where users can have access, but only one user at a time can hold the floor. This is implemented using a locking mechanism either with pessimistic or optimistic concurrency [332]. In general, these toolkits provide the software developers to build new applications.

3.3.1.2 Using Programming Languages

Most programming languages that support the development of networked applications can be used to handle user interaction. Programmers can design and implement some algorithms using these languages. Language types such as object-oriented, concurrent, scripting and functional are the common languages found in building networked systems.

As an example, in a multi-user multi-camera environment [78] that uses *simple contention*, *equal round robin* and *weighted round robin with timeout*, Java is used in the floor control mechanisms that is based on a server for the technique and implementation. Network programming skill is required to ensure proper connection between devices can be established and maintained. Similar requirements apply to control interactive video that implement a *first in first out* policy [260], and distributed multimedia in distance learning uses contention and round robin [283]. In all these examples, floor policies are determined and built in the server program with some choices being left to users to select the floor policy with which to work. JACIE goes beyond this approach as the provided language constructs enable additional policies to be added to cater for the application's need.

In research on agent systems by Demazeau *at al.* [85], agents could use an artificial language, which they call 'Interaction Language', to interact between agents through messages. The example of its general syntax is as follows.

```
<interaction> := <communication> <multiagent>
                 <application>
```

Here, `communication` is the main concern in the interaction among agents where messages must be sent efficiently. Upon receiving such messages, they should be interpreted correctly. Floor control policies can be determined in two ways, either using a *control model* or *social control model.* Control model uses an agent's knowledge to determine the floor according to the current states of the system, while the social control model is based on a controller who is the master to assign the floor according to some rules. In controlling robot vision using this artificial language, the social control model has proved to be more preferable since proper turn control can be determined.

3.3.1.3 Using Component-Based Approach

The component-based approach introduces another programming style where many components are provided for software developers to use. With this approach, it supports the developers for not only implementing new applications, but also allows them to extend the existing systems and applications.

For example, in the MOVE system [124] that uses a framework called ANTS, such framework has *Computer Supported Cooperative Work* (CSCW) [223] components and is able to provide synchronous group collaboration. Its architecture has several layers that enable the development of new coordination mechanisms to have interface with different middleware services. Floor control policies are determined by a master with the concept of a teacher and students environment.

Another example is a MultiTel framework [121], which is developed using Java, is composed of components built upon Java objects that run on distributed platforms. It separates the coordination and data processing into different components. The aim of all components is to allow software reusability by hiding its complex underlying technologies. Interaction between components can be performed using Java/RMI and the floor policy is based on contention.

3.4 Interest Management

From the software point of view, interest management mechanisms arise mostly in distributed simulations [187], distributed operating systems [185] and virtual collaborative systems. In distributed simulation research, it is often evaluated jointly with the factors of *load balancing* and *synchronisation* to evaluate system performances [187]. The main filtering factor is messages that are passed between users for a large scale networked systems. Other factors can include objects and variables which are performed through remote procedure call activities.

As interest management includes both filtering and access control in the context of this research, filtering issues and data sharing in programming languages and tools are reviewed in this section to investigate the characteristics and implementation factors. Some security and privacy issues are also included.

3.4.1 Filtering Issues

In systems that use message passing for user collaboration, it is crucial to reduce these activities, especially when they involve interactive collaboration. For example, Morgan *et al.* [228] have proposed interest management using standard *message-oriented middleware* (MOM) technologies to provide scalable message dissemination for networked games. Such games are implemented using *CORBA Notification Service* (CORBA NS) [136], one of popular MOM standards, with predictive aura-based user interest determination. Unlike JACIE, this approach has separated message filtering with user access and security in its development and only focuses on the filtering of messages to improve system performances.

Another example is a toolkit called CBE [193], that uses the concept of rooms to filter users through applet support. As this software supports interactive user commands at runtime, the user room selections determine user interest. This high level interest factor is later translated into an object-based mechanism through the application program interface. The proposed concept of room also supports user access by controlling users to have access to particular rooms. While this work has a similar approach to provide a similar concept of interest management in JACIE, its implementation uses applet-based applications rather than language constructs.

According to Dewan [90] who uses the term 'interaction model' for user interest from a high level point of view, filtering can be performed either by a 'room model' (like CBE above) or an 'aura model'. Upon presenting room selections as 'virtual rooms' in an application, users who enter a specific room can only 'see', 'hear' or interact 'closely' with others in the room, and know nothing about users in other rooms. With the 'aura model', objects that present users in 3D space can navigate and interact with other objects when they get close enough to be 'aware' of each other [90].

3.4.2 Data Sharing

Data sharing is a very common issue in programming languages or application software. In programming languages, data can contain several values such as numbers, characters and images, and can be represented by various forms such as single variable, a list of variables or objects. The term 'global' refers to data that does not belong to any routine or class and can be accessed from anywhere in a program [223]. In contrast, 'local' data can be accessed by only a part of a program [175]. Almost all programming languages used in developing networked applications do not support 'remote global' variables in either client or server program. They are often executed differently due to the fact that they often reside in different computers and different locations. JACIE had made the first attempt to allow the declaration

of 'global' variable in its client's program since JACIE provides both client and server program in one program, which leaves the separation of these programs to the compiler. Other languages allow 'global' sharing of variables or objects through some mechanisms such as message passing or RPC.

Normally, in computing, data sharing can be found in two situations as listed below.

- Sharing Within the Same Computer — In this situation, a computer can either have single processor or multiple processors. For a single processor system, data is shared among two or more program components. The term 'public' as in Java and C, is the keyword used to refer to the declaration of global variable type, or this global variable can be declared at the top of a program block as in Pascal, so that all the program routines can access it.

  For multiple processors that support parallel processing, a programming language can either use the similar concept of single processor with additional support of concurrent language features or facilitate inter-process communication through mechanisms such as message passing and RPC. Concurrency can be found in languages such as Modula-2 [278], concurrent C and concurrent Smalltalk.

- Sharing Among Different Computers — In this situation, all the computers are linked by a network connection where every computer usually has its own processor. Sharing of data is usually similar to the above mentioned 'global' variable, however, in sharing memory for distributed systems, processors can share one global memory or distributed memory that comes with each processor. For example, Linda introduces 'tuple space' for the global memory sharing and Agora [39] uses 'maps' for distributed sharing [22].

Nowadays, many programming languages have emerged to support not only sequential programming, but also concurrent programming. There are computers such as workstations that can have more than one processor to support large computational data. Network programming languages usually support threads of execution, where a program is able to split itself into two or more simultaneously running tasks [223]. Several attempts have been made to provide language constructs for high level language to support programming networked applications especially distributed applications. Such languages include Hermes [17], COOL [57], Eiffel [54] and many more. Most of these languages are built on top of the existing programming languages. Furthermore, in sharing global data, message passing is usually used besides RPC.

3.4.2.1 Sharing Factors

In networked systems, sharing of resources between computers at remote sites, can be in several forms that include the following.

- Memory/Variable — Much research has been undertaken on shared memory as it is a powerful abstraction for interprocess communication [300], and largely being discussed in networked environments such as parallel systems [180], concurrent systems

and *distributed shared memory* (DSM) [258].

In parallel programming, shared variables is probably the oldest paradigm that mainly relies on the operating system support for interprocess communication and they are used for tightly coupled systems while message passing is for loosely coupled systems [22].

According to Steinke and Nutt [300], the concept of shared memory originated from multiprogramming on uniprocessors and bus-based multiprocesses where a *simple model* of the memory system is enforced in hardware. The model consists of the following statements.

- Each variable is represented by a physical memory cell that both the memory and the processor have the same states.
- The memory operations are in sequentially forms where *read operation* returns the value of its current state and *write operation* changes the current state of the physical memory.
- The operations of each process take place in the order specified by its program.

They further pointed out that their concept of *consistency model*, which has a function that maps each input to a set of allowable outputs, allows the shared memory not to be tied to the physical implementation of memory cells. This leads to the idea of its representation as an API as shown in Figure 3.3. It illustrates how an application program can process shared data without the knowledge of the real hardware implementation by both program and shared memory agreeing on this model [300].

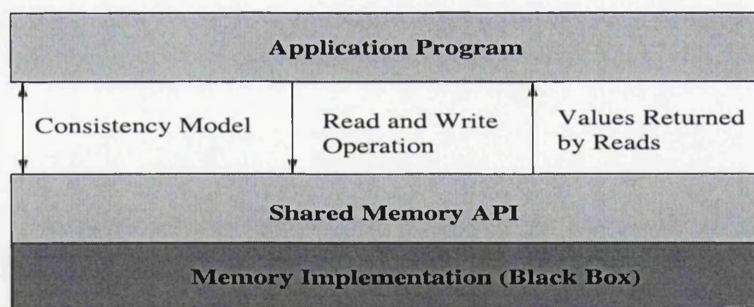


Figure 3.3: Shared Memory as an API [300].

- Object — In programming, an object is an individual unit, which is used as the basic building block of programs, for binding data with methods that operate on that data [223]. Object-oriented programming has attracted programmers not only for writing sequential programs, but also concurrent programs that allow networked applications to be implemented [22]. Object-based sharing is often found in various types of applications such as virtual systems [304], parallel systems [57], agent systems [166] or also in some database systems that use object-oriented databases [167]. To enable the sharing facilities, various platforms have been introduced with generic features for cross-platforms abilities. In particular, CORBA from OMG [136], DCOM from

Microsoft [72] and *Remote Method Invocation* (RMI) from Sun MicroSystem have made successful attempts [221].

The term *component* is usually referred to part of the software that is built from objects. It is usually defined as a small part of a large program that is independent and reusable. It may also be defined as a black box. In [209], the detailed descriptions of the terms are illustrated that may vary according to different perspectives. The notion, software component, interface, service, encapsulation, reuse and plug-n-play can be referred to it. Software components and executable software components very much the same characteristics as in an object module, however, object reusable uses the class libraries while the component reusable can be platform independent [209]. In most network systems, components and objects may be used interchangeably. From the design and implementation perspectives, the adaptability of components to suit the needs of the designer is the main objective.

In building a software system or tool for a distributed system, software engineers use distributed component technologies and interfaces [110]. Therefore, a large number of component technologies exists such as Enterprise Java Beans(EJB), COM, CORBA [286], RMI, and CORBA Component Model(CCM) [136] to allow distributed execution on various platforms. The technologies, based on object-oriented programming give economical and cost effective ways of software development and usability. The component models provide them with some mechanisms to compose components through well defined interfaces rather than developing new or changing the existing components. The use of APIs and RPC in the system design and implementations are also becoming very common [110].

There are many object-based component systems that have CORBA or Microsoft ActiveX approach with separate interface definition language to map the passing and object mapping for the remote objects [265]. The Jinni project [311] is an example of gluing together components and objects in networked client/server applications. In this way, it can support platform independence between Prolog [44] and Java for effective integration of inference technologies.

Object and component sharing are supported by programming languages such as Smalltalk [290], Eiffel [294], Ruby [321], Java, C# and many more. Toolkits such as COAST [281] also support object sharing through the use of application interfaces. As toolkits are implemented using an existing programming language, the underlying approach and mechanism are similar to that programming language, even though they provide application designers with different application development environments.

- Database — In database systems, the contents can be shared among users by either allowing access to one's database or it is also possible to have integration between databases. The former approach is simpler and only requires the inclusion of user access rules, compared to the latter approach that may require *ontologies* to help designers to understand the semantics of database objects [207]. 'Software database drivers are available for most database platforms so that application software can use a common application programming interface (API) to retrieve the information stored

in a database. Two commonly used database APIs are JDBC and ODBC' [223].

There are several other factors such as files and system clock ('external system resources'), called 'externalities' by Begole *et al.* [28] to describe resources that need state representation which is external to the application. In this example of resources sharing, a replicated application-sharing system is implemented, namely *Flexible JAMM (Java Applets Made Multiuser)* by allowing transparent, dynamic replacement of externality with a switching from direct local access to 'proxied' remote mode when the application is switched from single to multi-user mode. Such implementations are performed by modifying the Java core library and native platform classes on the Java platform [28].

3.4.2.2 Security and Privacy

Security and privacy are important factors in almost all types of network applications and in fact, it must be included in the design and implementation of networked collaborative applications. Even though there are many collaborative frameworks available, the security and restrictions on user access are usually either omitted (as these frameworks only concerned with providing rather than controlling collaborative environments), or they rely on the 'back-end' support such as the operating system or database system [89].

In database and operating systems, this issue is common where databases usually deal with information in terms of records, and operating systems handle files. Access to a particular record or file is managed in such a way that access rules are set at the beginning of a collaborative session and they remain unchanged throughout the application. As pointed out by Dewan and Shen [89] user interface control at the 'front end' of an interactive program needs to have more flexible approach to user access. They presented this approach through a toolkit called Suite [88] where the design is based on an *access matrix* model.

Access matrix or access control matrix is 'an abstract, formal security model used in computer systems, that characterises the rights of each subject with respect to every object in the system. It was first introduced by Lampson 1971' [223]. Figure 3.2 shows the general form of the model that has a list of subjects against a list of available objects in the system with their corresponding access rules. The access rules usually contain access to several operations such as read and write.

	object 1	**object 2**	**...**	**object$_{(n-1)}$**	**object$_n$**
subject 1	access rule	access rule	access rule	access rule	access rule
subject 2	access rule	access rule	access rule	access rule	access rule
...	access rule	access rule	access rule	access rule	access rule
subject$_{(n-1)}$	access rule	access rule	access rule	access rule	access rule
subject$_n$	access rule	access rule	access rule	access rule	access rule

Table 3.2: General Form of Access Control Matrix Model.

There are many collaborative systems adopting this matrix model in their design of access control and user rights besides the Suite toolkit, JACIE also uses the same model in the design process and this model is also largely used in grid computing [280].

In implementing the access rule and security, languages are often used to do the control and checking. In database systems, query languages such as SQL [137] and XQuery [322], are the common approach and in addition, middleware such as *Semantic Access Control Enabler* (SACE) [247] can also be used for flexible control of information access. With this flexibility, sharing databases is not only limited to a single database, but also covers several heterogeneous information systems. So far, there is no attempt to provide user access control and security protection integrated into one control in the form of high-level language.

Chapter 4

JACIE Overview and Enhancements

Contents

4.1 Introduction 63
4.2 Overview of JACIE 64
4.3 JACIE Compiler 65
4.4 Main Features of JACIE 67
4.5 Managing Collaboration 69
4.6 Race Condition 73
4.7 Improvements to the JACIE Language 79
4.8 Summary 88

4.1 Introduction

This chapter covers the fundamentals of JACIE. This work builds on the previous research work, the original language is referred to as JACIE I, and the extended version JACIE II. An overview of JACIE I is first presented, followed by a description of its main features. The enhancements that have been made to the language are then described by looking into several issues such as race condition, data type and compilation errors while major extensions on the interaction and interest management are to be discussed in Chapter 5 and 6.

The JACIE compiler was built using a multi-layer software architecture as shown in Figure 4.1. At the top level of the architecture are the JACIE language scripts. These scripts are kept in a file with an extension `.jacie`. At the next layer, the JACIE compiler takes the file `x.jacie` and translates the script into the equivalent Java files that reside in two different directories. One directory contains all the client programs, while the other directory has all the files for the server programs. All these client and server Java programs must be compiled to produce Java runnable classes. Some client programs are not only in the form of Java files, but they can also be html files. The lowest layer of the system architecture contains the Java Virtual Machine to process the Java classes produced by the JACIE program.

It is also possible to have a WWW browser and the Netmeeting software to display the html file and process the Netmeeting applications, respectively.

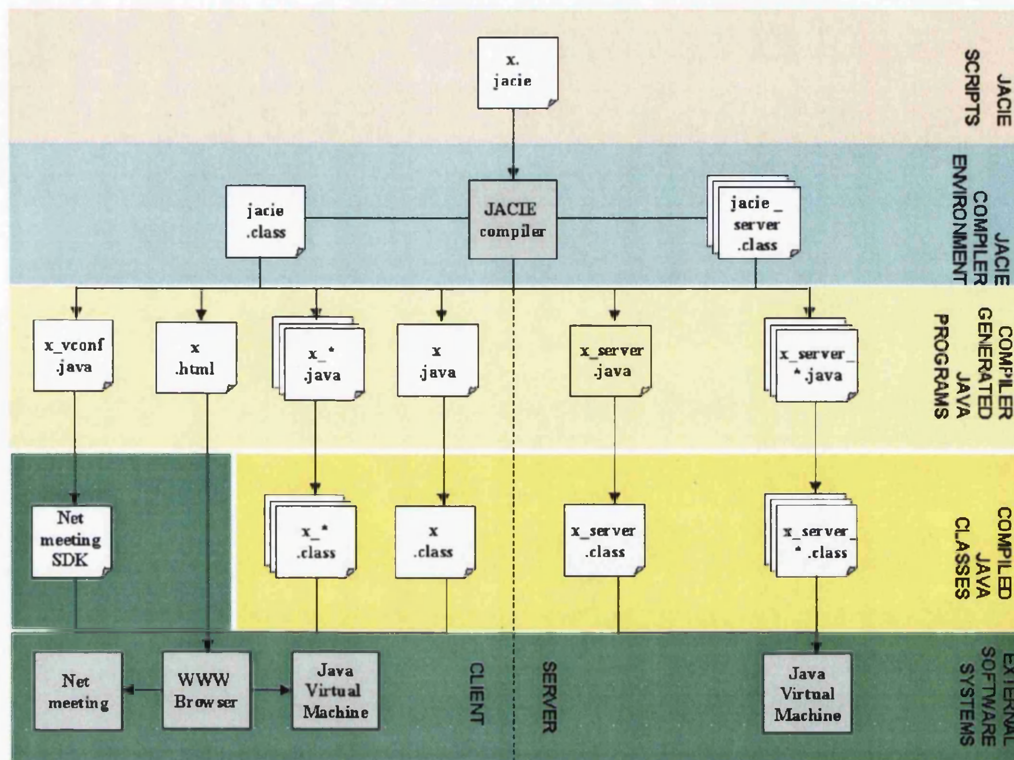


Figure 4.1: JACIE Software Architecture.

Although new language constructs have been introduced and JACIE I has been extended, the original software architecture remains the same. The extensions merely involved modifying and adding more language statements instead of changing its overall structure. Therefore, such amendments and additional language constructs for JACIE II are performed in the JACIE compiler (in the JACIE compiler environment layer) and its next layer, 'compiler generated Java programs'. The JACIE II design and changes are also related to several existing Java Classes and methods that are scattered over 150 different files. Any new statements introduced must be compiled by the JACIE compiler and details on this process is illustrated in Figure 4.2 and described in Section 4.3.

4.2 Overview of JACIE

JACIE (*Java-based Authoring language for Collaborative Interactive Environments*) is a scripting language designed for rapid prototyping of distributed collaborative applications. In particular, it targets a collection of applications for which the existing programming tools would incur expensive development costs. From a software engineering perspective, the key design principles of JACIE are therefore *special purpose* and *programming efficiency.* A detailed description of JACIE can be found in [140]. This language was first designed and

built over the period 1997-2001.

JACIE provides a software development tool that enables distributed collaborative applications to be developed at a very low development cost, within a short development period, and by possibly inexperienced programmers. Such applications may include groupware (e.g., a collaborative design environment), e-learning courseware (e.g., a teamwork exercise), and web-based games (e.g., board games and card games). These applications commonly feature interactive collaborative activities, shared working canvases, controlled access domains and structured communication. It is often very difficult for applications in these areas to generate much commercial profit, and thereby difficult to attract resources to develop them in the first place. For example, implementing an online bridge game in JACIE (illustrated in Chapter 7), requires much less development skills and effort than would be needed with Java, assuming the programmer has no previous experience in either language.

JACIE contains a collection of built-in language constructs for supporting structured and unstructured communications through various communication channels such as canvas, message, chat, voice, video and whiteboard. JACIE first introduced a small set of interaction protocols as built-in language constructs, which define the rules that govern the means of interaction between users in a collaborative environment, and are used to coordinate the input from users and the display on a channel.

Learning the popularity of scripting languages in Internet technologies, JACIE was designed as a scripting language, and its compiler generates target programs in Java. JACIE also allows the inclusion of Java code as part of a JACIE program, enabling experienced programmers to utilise Java for the implementation of complex code segments. To alleviate the difficulties in network programming, JACIE also employs a template-based approach and uses a single program to specify both server and client.

In the original design of JACIE, the communication between clients had to be implemented using the message channel. As the message channel is routed through a server, relevance filtering and message security can be implemented at the server but relies largely on user-defined code segments. It is hence desirable to introduce more sophisticated language constructs for interest management, which can improve programming efficiency in developing applications that feature shared data and states.

4.3 JACIE Compiler

The standard compiler construction process takes a program written in one language as the input, and then produces an output in the form of a target language. Two compiler tools, JFlex and JCup are used at the 'lower level' of the compilation. JFlex undertakes the lexical analysis of the JACIE program and JCup then performs the parsing. Both tools produce the output in the form of Java files. Figure 4.2 shows the main flow of translating a JACIE program into a corresponding Java program.

In the diagram, the green rectangular boxes represent written files that must be used in the

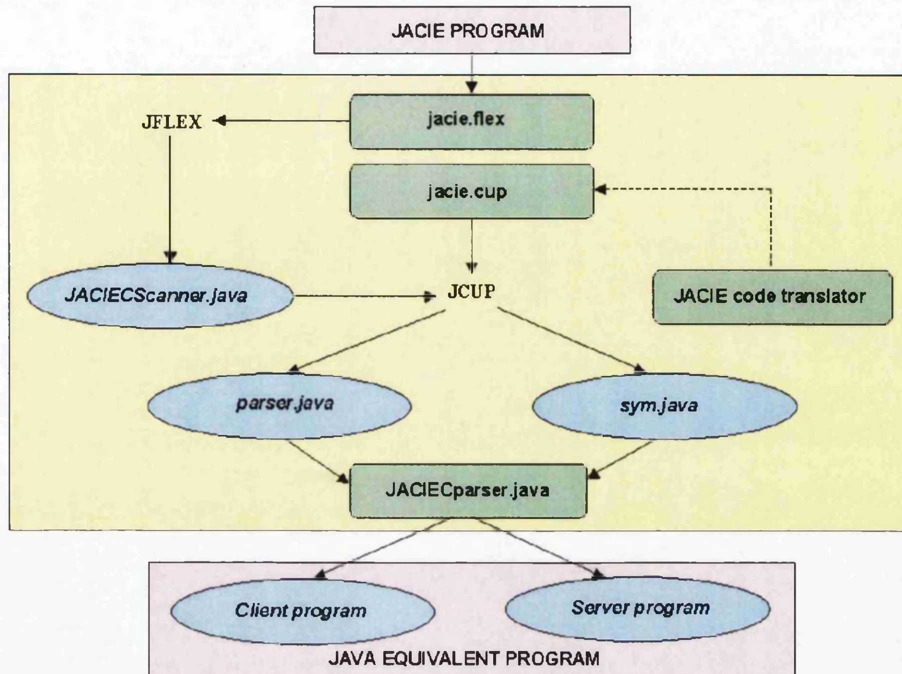


Figure 4.2: JACIE Compiler.

compiling process. The file, `jacie.flex`, contains the JFlex specifications of tokens. JFlex produces the identified tokens in a file called `JACIEScanner.java`. This file together with `jacie.cup`, which contains the JACIE grammar specification, are taken by JCup to produce intermediate codes and compiling states (`parser.java` and `sym.java`). The file `jacie.cup` also contains several objects which are used by the `JACIE code translator`. The `JACIE code translator` is written in Java that must be compiled prior to executing JCup. It is the main part of the JACIE compiler and consists of over a hundred Java classes. It is where most of the work for the extensions and enhancements to the language reside.

The final stage of compiling is in the file `JACIECParser.java` that links the intermediate codes and all Java classes produced by the code translator to get the equivalent Java program. All the output produced in different stages of the compiling process are presented in ellipses.

For JACIE II, only the `JACIECParser.java` has not been changed while the rest of the files, such as `jacie.flex`, `jacie.cup` and most of the `JACIE code translator` files, have to be modified. As many of the old files have been modified and new files with Java classes introduced, a complete understanding of JACIE I for all layers of its architecture is needed.

4.4 Main Features of JACIE

In this section we highlight the essential features of JACIE, which are as follows:

- Template based programming style — A JACIE program fits into a prescribed template that eases the programming task. It consists of three main components, *System Configuration*, *Client Body* and *Server Body*. Furthermore, in the *Client Body* and *Server Body*, a template based style is also adopted, for example, the segments of *Session Start*, *On Session* and *Session End*. The JACIE programmer only needs to fill in the template. The *System Configuration* is the component where the initial values for communication are set, such as port number, applet name, initial protocol, number of users and message identifiers that are shared by the server and all clients. In Figure 4.3, the JACIE standard template layout is displayed.

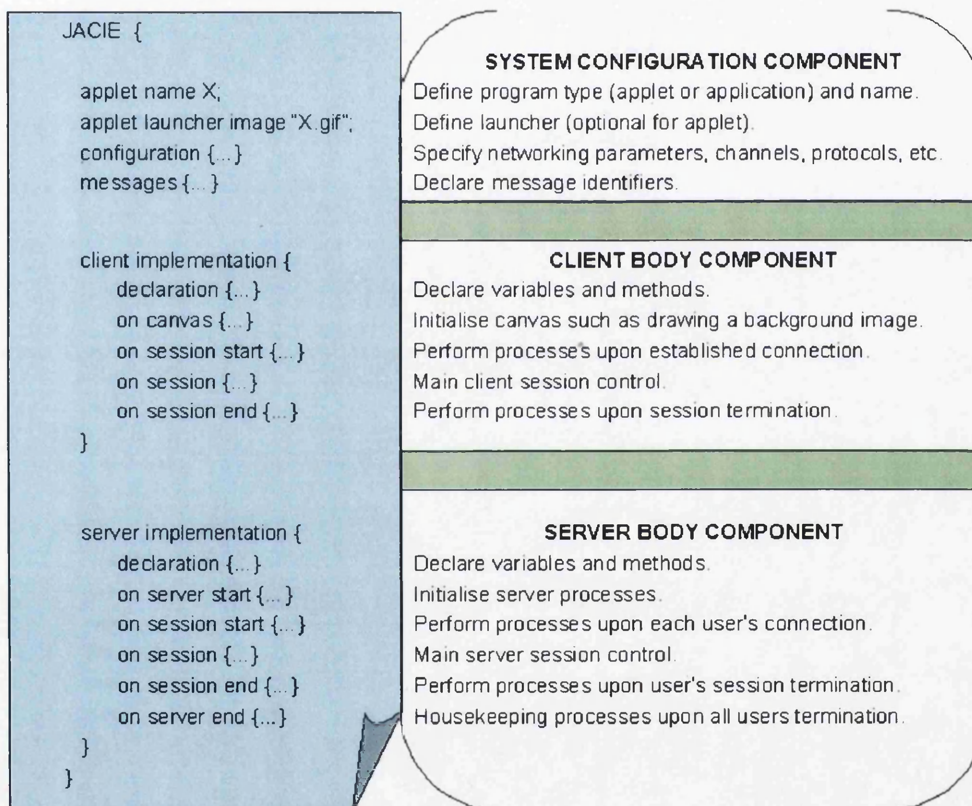


Figure 4.3: Standard JACIE Component.

JACIE is an event based programming language, examples of such events are mouseclick triggers, messages received from the server, turn control and many more. In the server program, more segments are included that consist of *On Server Start* and *On Server End*. The template based style influences the programming of the session control. *On Session* always contains the main events for the collaborative processes. Therefore, a programmer can really focus on this segment to design effective applications.

- Single program for server and client — JACIE allows the integration of the code segments for both server and client in one program. Most other network programming language separate these into two programs as both may play different roles and feature different environments. JACIE allows this combination in order to ease the data manipulation between server and clients. It is the compiler's job to generate the separate programs that run on separate computers.
- Communication channels — There are several built-in communication channels that facilitate different forms of media and communication methods. The channels are *canvas*, *message*, *chat*, *voice*, *video* and *whiteboard.* The last three channels are supported by the Microsoft Netmeeting. At present, JACIE supports two dimensional graphics on a canvas of default resolution of 632 by 360 where each point on the canvas is represented by a *grid point.* Several *special variables* are provided for grid manipulation such as 'getting grid point' and 'checking on a specific area on the grid'. The most frequently used communication method is the message channel between client and server as JACIE can support server mediated communication. Even though the communication is server mediated, the management of any application can be fully distributed, centralised or a combination of both.
- Interaction protocols — In JACIE I, all the interaction protocols that were introduced were built-in and managed by the server, with only the round robin and contention protocols implemented and tested. Other built-in protocols were described but not implemented. JACIE II introduced additional interaction protocols enabling almost all common collaborative applications to be programmed.
- Multithreading — A language for building network applications must provide multithreading in order to accommodate the computing environment of a networked system. Since JACIE is built using Java which has this feature, it was not difficult to include this facility. For example, in the *On Session* section where the main event (user session) occurs, several sub-events such as message received, timer options and waiting events can occur any time within the event.
- Interfacing with Java — As the JACIE compiler was written in Java, it is desirable for JACIE to have a programming interface with the Java language. The inclusion of Java code as part of the program enables experienced programmers to utilise Java for meeting additional requirements, for example, in a more complicated graphics application.

The language can produce three different types of client programs, namely an applet, a separate window invoked by an image launcher within an applet, and a separate window invoked by a text button within an applet. Figure 4.4 shows the layout of the basic user interface that demonstrates the nested implemented environment. The labelled layer starting with the letter x, are the objects implemented by built in Java Classes which include the Java Abstract Window Toolkit (AWT), the runnable standard interface and the Frame. The main panel, which is seen by users, basically has three components, menu bar, canvas and message bar. The menu bar contains all the communication commands such as image buttons for connecting and disconnecting to the server, channel selection button, as well as text input for host name and username. The message bar has the server generated messages called *Local Message* and *Server Message.* This message bar can also display a text input option

if required by the application.

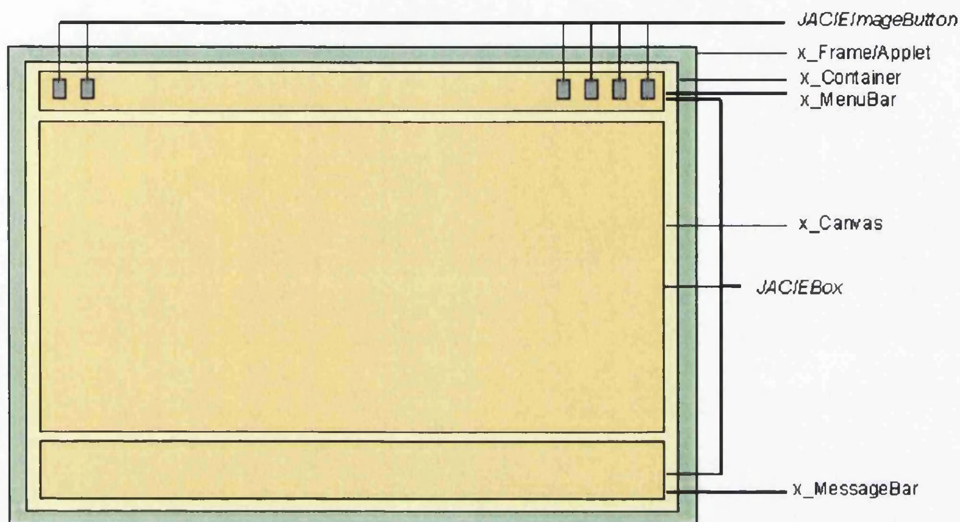


Figure 4.4: Layout Diagram of JACIE User Interface.

In addition to all these features, some statements in JACIE II consist of several option tags for various control and flexibility in programming. The lower level and hard coded programs are always hidden. This allows the programmer to concentrate on the higher level program and leave all the details concerning network connections and sockets to the compiler.

4.5 Managing Collaboration

In managing collaboration among users, all users must establish their connections to the server. Due to the structure of JACIE which is designed in the form of transition states through a typical client/server interaction, interaction activities for a client or server can be specifically focused. Figure 4.5 illustrates the state transition diagram of the involving states. A server starts its execution and waits for any user to establish a connection until the appropriate number of users are connected to it. At a client, once it starts opening an application by referencing to the address of the server, a user is in a `starting state`. When a connection to the server is established, a user is put into a `waiting state` if more users need to join a session, otherwise, the user goes to the `interacting state`. While in the `interacting state`, both client and server may continue in this state while performing activities in accordance with the instruction specified by the application. This is the state where collaboration among users take place. The network connection remains active and messages are exchanged until either the client or the server terminates the session. At the `terminating state`, some 'house keeping' for the specific client may be performed before the link to such client is closed. At the server, it is in `reinitialising state` or `ending state` when all client connections are gracefully terminated where some 'house keeping' may be performed. Then, either the server goes back to the `starting state` which enables it to stay in the `waiting state` and ready for the next action, or it is also

possible to terminate the server by stopping its execution.

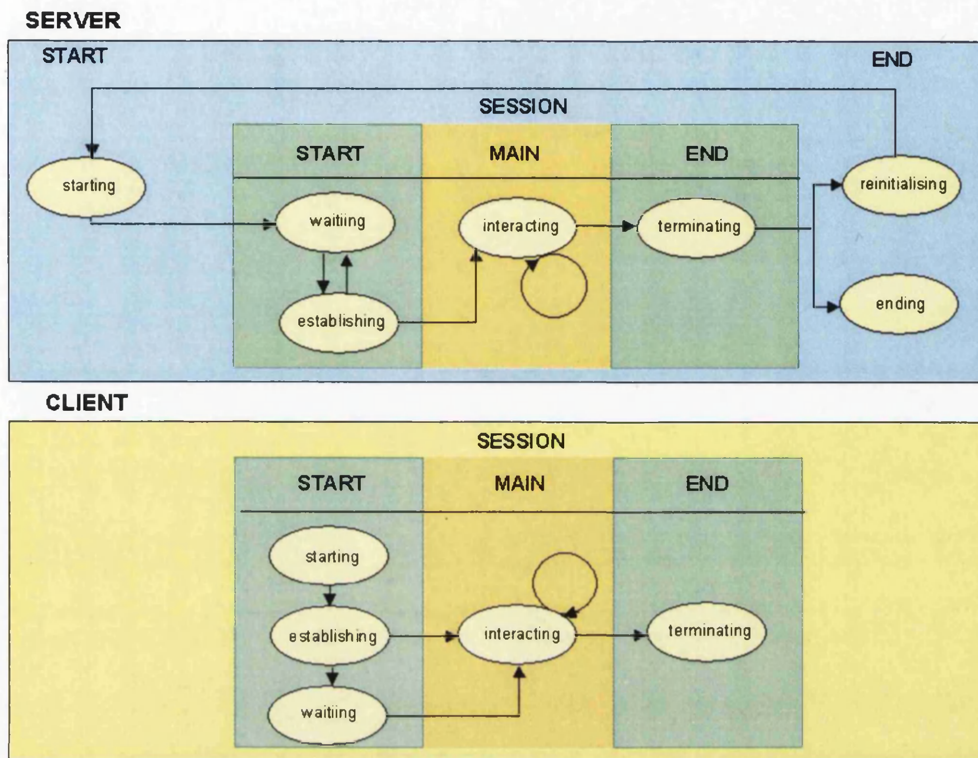


Figure 4.5: State Diagram for Server and Client.

JACIE's main communication method is message passing. Each message can be of various types (i.e. integer, string, *etc.*) but only in one format. Each JACIE message is appended by a header of type integer. The header is called the *message identifier* and determines its content. Figure 4.6 shows a JACIE message and its representation in relation to other network layers. The message is in a high level form that allows itself to be part of the TCP packet, IP datagram and message frame that is transmitted through the actual network link. In this way, it provides flexible message specification and location transfer.

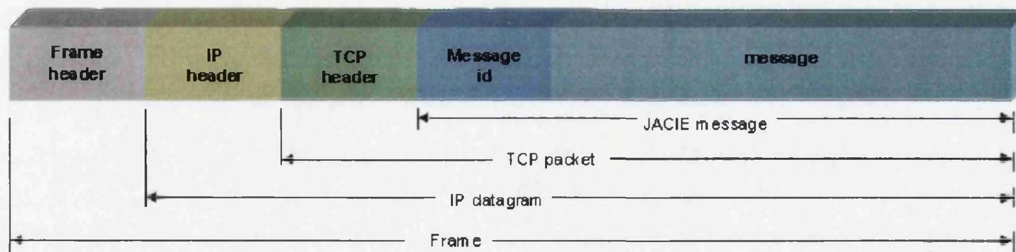


Figure 4.6: A JACIE Message and Its Representation in Relation to Other Network Layers.

In general, a JACIE message identifier can be of two different kinds, either *system defined*

or *user-defined.* The *system defined* message identifier is a preset value in the JACIE compiler and hidden from a JACIE programmer. In JACIE I, there are 20 message identifiers that deal with the establishment and verification of users, notification of states, and some information on the active session such as user number, group number, turn number (for current user in control) and group turn number. JACIE II extends the value range of these message identifiers up to 39 by introducing more message specifications for managing floor request and timers in interaction management and handling the access specification and verification of variables in interest management. Below is the example of the *system defined* message header for JACIE I for session handling and user turn control. These values are numbered from 10 to 20 and represented by variable names written in capital letters of type Java 'constant values'. These values are set in both client and server modules which are called *ClientSession* and *ServerSession* components, respectively.

```
public static final int WAITOVER                  = 10;
public static final int STARTSESSION              = 11;
public static final int USERGROUPNUMBER           = 12;
public static final int USERGROUPINFORMATION      = 13;

public static final int TURNNUMBER                = 14;
public static final int GROUPTURNNUMBER           = 15;
public static final int PASSTURN                  = 16;
public static final int ONLINEUSERTERMINATES      = 17;

public static final int TERMINATECONNECTION       = 18;
public static final int TURNEXPIRED               = 19;
public static final int NEWTURN                   = 20;
```

Figure 4.7 illustrates an example of JACIE message that represents the message identifier name, 14, which is TURNNUMBER and its value is 2. This means that the user turn is currently set to a user with the *user number* 2.

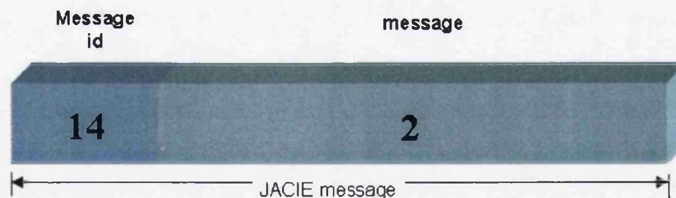


Figure 4.7: An Example of a JACIE Message with Values.

The *user defined* messages are associated with JACIE standard messages and declared in the JACIE program. They are usually listed in the JACIE *system configuration component* (Figure 4.3) as identifiers. During compilation, the JACIE compiler sets these identifiers to start with the value 1000. These messages are used in JACIE application programs that are associated with *send* and *receive* statements. The language constructs for these statements are as the following.

```
send <identifier> [ <expression list> [ to <permission list> ] ]
```

```
receive <identifier> [expression list]
```

For every *send* and *receive* statement, the `<identifier>` that represents a message header, is included to acknowledge the type of message to be exchanged. It is usually followed by `<expression list>` for indicating the actual value. In sending a message from the server, it is optional to have a `<permission list>` specifically to address the receiver that is represented by an identifier. Below is an example code segment that shows the declaration of the message identifiers namely `gridX`, `gridY`, `posX` and `posY` with their corresponding *send* and *receive* statements.

```
JACIE { ...                                   // start of JACIE program
  configuration { ...                         // Configuration section
    host prompt;
    port 2000;
    username prompt;
    ...
  }
  messages {                                  // ◄ user-defined message identifiers
     gridX, gridY, posX, posY, ...
  }
  client implementation { ...                 // Start of Client Body section
      gX = GETGRIDX;
      gY = GETGRIDY;                          // gX and gY are local variables
      ...
      send gridX gX;                          // ◄ Send Statement with the message identifier gridX
      send gridY gY;
      ...
      receive posX ptX;                       // ◄ Receive Statement with the message identifier posX
      receive posY ptY;
      ...
  }
  server implementation { ...                 // Start of Server Body section
      receive gridX pointX;                   // ◄ Receive Statement with the message identifier gridX
      receive gridY pointY;
      ...
      send posX pointX to all;                // ◄ Send Statement with the message identifier posX
      send posY pointY to all;
  }
```

All the user-defined message identifiers are translated and coded into an equivalent Java program in the server component called *Global Data Manager*, which will be discussed in detail in Section 4.6. The following code segment shows the translated codes for the declaration of these message identifiers.

```
...
public static final int GRIDX = 1000;   // ◄ The value of message header starts with 1000
public static final int GRIDY = 1001;
public static final int POSX = 1002;
public static final int POSY = 1003;
...
```

In managing these identifiers, JACIE separates the location of the two different types of message identifier declaration, *system defined* and *user-defined* into two separate components of the server program. A similar action is taken in the JACIE client program.

In managing a session, JACIE adopts the *Transmission Control Protocol* (TCP) with point to point connection using sockets. TCP connections simultaneously transmit and receive

data that support full-duplex transmission. It is able to broadcast standard messages to all the clients and JACIE also facilitates multicast message sending. It has several program components that work closely and support one another in manipulating all these messages.

4.6 Race Condition

'A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly' [329]. Race condition detection is a common issue in an operating system or concurrent programming where there is no specific algorithm to detect its condition in a program [55]. It is possible to have race conditions in JACIE as it allows concurrency with shared resources.

4.6.1 The Causes

In JACIE, a race condition may occur in the queues that handle messages at server and client. Figure 4.8 shows the diagram of the processes occurring at both queues. Message exchange between client and server are handled by *ClientSession* and *ServerSession* components, respectively. *ClientSession* component gets the receiving messages and puts them at the back of the client message queue. Like *ClientSession*, *ServerSession* component also does the same for the server message queue.

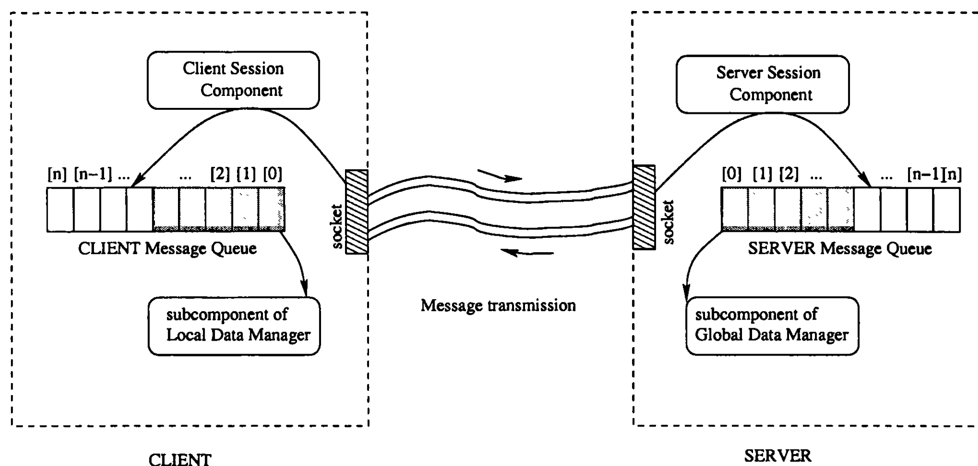


Figure 4.8: Message Exchange.

JACIE supports concurrency in both client and server sessions. It monitors this message queue using a *semaphore*. The queue itself is represented by a Java *Vector* class that allows its index counter to be increased or decreased corresponding to the 'add' or 'remove' operations. Below is a code segment of the Java translated program that handles the 'remove' operation. This operation is synchronised with the remove action is taken place when the appropriate semaphore counter (determined in `isEmpty()`) is achieved.

```
public synchronized void remove() {
   while (isEmpty()) {
      try {
         // wait until available
         wait();
      } catch (InterruptedException e) { }
   }
   queue.removeElementAt(0);
   ...
}
```

At client, data in *CLIENT Message Queue* is retrieved by a subcomponent of *Local Data Manager.* At this queue, the race condition can happen when the *Client Session Component* adds a new message into the queue and at the same time, the *Local Data Manager* is retrieving a message from the queue. Similar to the server, *Server Session Component* may adds a new message into the *SERVER Message Queue* at the same time when the subcomponent of *Global Data Manager* retrieves a message from the queue. This allows a race condition to happen when the queue's index counter can be incorrect. As pointed out by Christopher and Thiruvathukal [64], 'threads can try to update the same data structure at the same time. The result can be partly what one thread wrote and partly what the other thread wrote. This garbles the data structure, typically causing the next thread that tries to use it to crash'.

In addition to having two actions ('add' and 'remove') on the message queue from two different components at the server, JACIE I also allows the *Global Data Manager* to add data into the queue, which can happen in a separate thread. Hence, the total of three types of operations are possible in updating the queue. This leads to even higher possibility of having race conditions.

4.6.2 The Detection

In using JACIE I to execute an application such as the bridge game, there is no specific algorithm to detect a race condition. From the users' point of view, they may potentially experience the undefined states that all users cannot proceed with the collaboration. For example, in executing the bridge game, all the connected users for the JACIE server would have the same server message that shows 'It is your opponent turn..'. Therefore, no user is in control of the turn and the game cannot continue.

From a JACIE compiler point of view, it is possible to debug the server program by printing some messages on any action regarding the message queue. In this way, the sequence of data to be inserted and removed can be traced to see where and when the race condition may happen. Below is some of the debugging messages printed by several components of the JACIE server. In the example, there are four users who exchange messages with the server. The codes written at the left of all the statements and in between symbol '[' and ']' represent symbolic names of the server components. The code which is represented by '*?*' indicates the action of adding more data into the message queue by one of the server components. This data addition may potentially clash with the messages received from clients. All the numbers listed below either represent the actual data passed as messages, *system defined*

or *user defined* message identifiers.

```
[SA1] sending to Siti : 14:3     // (first) broadcasting messages for 'user turn' (id value 14) with data 3
[SA2] sending to Rudy : 14:3
[SA3] sending to Ann : 14:3
[SA4] sending to Susan : 14:3
[*?*] queue add - 20:            // Sever Data Manager inserts message id represented by value 20
[*?*] queue add - 20:
[DA1] queue remove - 20:
[*?*] queue add - 20:
[*?*] queue add - 20:
[DA2] queue remove - 20:
[DA3] queue remove - 20:
[DA4] queue remove - 20:         // balance number of queue 'add' and 'remove' on dealing with id number 20
[SA3] received from Ann : 1018:0
[SA3] queue add - 1018:0
[*?*] queue remove - 1018:0
[SA3] received from Ann : 16:
[SA3] queue add - 16:
[DA3] queue remove - 16:
[SA1] sending to Siti : 14:4     // (second) broadcasting messages for 'user turn' (id value 14) with data 4
[SA2] sending to Rudy : 14:4
[SA3] sending to Ann : 14:4
[SA4] sending to Susan : 14:4
[*?*] queue add - 20:
[SA4] received from Susan : 1018:0
[SA4] queue add - 1018:0
[SA4] received from Susan : 16:
[SA4] queue add - 16:
[*?*] queue add - 20:            // ◄ the race condition problem is suspected to occur
[*?*] queue add - 20:
[*?*] queue add - 20:
[DA3] queue remove - 20:
[DA2] queue remove - 20:         // the number of queue 'add' and 'remove' for id number 20
[DA1] queue remove - 20:         // must be a total of 4 each
```

The example concludes that the 'unbalance' operations (adding and removing) items on the queue after the second broadcasting of messages, result in the race condition where the thread has crashed before another add item for the value 20 and the removal of the identifier for the value 16 can be performed.

4.6.3 The Solution

The common approach in handling race conditions is by a locking mechanism to ensure mutual exclusion on shared resources. On top of using the semaphore, JACIE II also chooses to avoid any 'unsafe' condition. The 'unsafe' condition is when there is a high chance of having a race condition.

The following code segment is part of the JACIE I compiler that can lead to the race condition. The code segment shows how messages at the server queue are retrieved and how subsequent activities are handled. In the `TURNPASS` option, more messages are added to the queue by the server in order to activate the `TURNEXPIRED` or the `NEWTURN` option selected after the `TURNPASS` action. This situation may cause an error as adding the new messages into the queue may violate the mutual exclusion because at the same time, the server may receive new messages from clients that have to be added to the same queue. Therefore, this

condition is not safe. The code segment is part of the *Global Data Manager* that handles the multithreading at the server. This unsafe condition arises when a call to the *Server Session* component is made.

```
...
private int currentTurn;                              // 'currentTurn' determines user turn
...
while (!Thread.interrupted()) {                       // Listen to any message
 currentMessageId = inputStreamQueue.checkMessageId();
 // retrieve message from the Server Message Queue
 if (currentMessageId != 0)                           // A new message detected
  switch (currentMessageId) {
   ...
   case BMServerSessionManager.TURNPASS :         // ◄ retrieve message header 'TURNPASS'
    ...
    myTurn = false;                                   // set off any user turn
    onSession();                                      // process current user messages
    currentTurn = BMServerFloorManager.nextTurn();    // get next user turn
    BMServerSessionManager.broadcastTurn(sessionAssistant);
    ... // ▲ make a call to session manager to broadcast new turn number
    myTurn=BMServerFloorManager.currentTurn==sessionAssistant.usernumber;
    ... // initialise turn
   break;
   case BMServerSessionManager.TURNEXPIRED :      // ◄ retrieve message header 'TURNEXPIRED'
    ...
    myTurn = false;
    onSession();
   break;
   case BMServerSessionManager.NEWTURN :          // ◄ retrieve message header 'NEWTURN'
    ...
    myTurn=BMServerFloorManager.currentTurn==sessionAssistant.usernumber;
    ... // ▲ actual turn setting
   break;
```

When the *Server Session Manager* broadcasts the new turn number to all the clients, the new message NEWTURN is put into the server message queue to enable the proper assignment of the turn number. This step cannot only cause serious problems, but may also delay the processing time in assigning the real turn to all the clients. Below is the code segment that shows how the method, `broadcastTurn`, handles the actual broadcasting of messages to inform all the clients of the new turn number and does the updating to the queue.

```
public static void broadcastTurn ("+name+"ServerSessionAssistant sender) {
... // broadcast new turn number and then make the actual turn setting
 broadcast(TURNNUMBER+":BMServerFloorManager.currentTurn, sender, true);
 synchronized (userList) {
  Enumeration enum = userList.elements();
  while (enum.hasMoreElements()) {
   BMServerSessionAssistant clientHandler=(BMServerSessionAssistant)enum.nextElement();
   clientHandler.inputStreamQueue.put(new JACIEMessage(NEWTURN+":"))
   ... // ▲ put message into the server message queue
  }
 }
}
```

To solve the above problem and increase the program efficiency, both message header, `TURNEXPIRED` and `NEWTURN` are deleted from the option message header list. At the `TURNPASS` option, the user's turn assignment is updated and broadcast. The control vari-

able `currentTurn`, which is initially set as private, is changed to static, making its value to be globally available to all the server program components. Therefore, once its value is set, all the connected users have its value immediately.

The new improved safer condition is illustrated in the following code segment. The call to the *Server Session* component is still made, however, instead of calling the method `broadcastTurn` that sends the new turn value to all clients before adding a new message to achieve the turn setting for every client, the call is made to the `broadcast` method to set the user turn directly.

```
...
static int currentTurn;                        // ◄ change variable type to static
...
while (!Thread.interrupted()) {                // ◄ Listen to any message
 currentMessageId = inputStreamQueue.checkMessageId();
 myTurn = BGServerFloorManager.currentTurn ==
                         sessionAssistant.groupNumber;
 ... // ▲ Set the actual turn
 if (currentMessageId != 0)
  switch (currentMessageId) {
   case BGServerSessionManager.TURNPASS :
    currentMessage = inputStreamQueue.get
                    ("[DA"+sessionAssistant.usernumber+"]");
    ... // ▲ Retrieve current message
    myTurn = false;
    onSession();                               // ◄ process message from the current client in control
    currentTurn = BGServerFloorManager.nextTurn();
    BGServerSessionManager.broadcast
                    (BGServerSessionManager.TURNNUMBER+":"+
                     BGServerFloorManager.currentTurn,
                     sessionAssistant, true);
    ... // ▲ broadcast the new turn setting immediately
   break;
  ...
```

The example given above is for the server side and the following code segment deals with the race condition at the client side. At the client, the possibility of having 'unsafe' state is less compared to the server. While the server may have three different attempts of updating the queue at one time, the client can have only two attempts that are made by the *Client Session* component and *Local Data Manager.*

The example code shows how the client can face a problem when a message is retrieved during the `NEWMESSAGE` event in the JACIE program. Like the server that retrieves all the *system defined* messages, this program instruction allows a user to retrieve *user-defined* messages at the client. Retrieving a *user-defined* message is a higher level approach provided to a JACIE programmer which actually instructs the *Local Data Manager* to retrieve such messages from the client message queue.

In avoiding the 'unsafe' condition, a JACIE program must try to retrieve the message as quickly as possible to avoid any delay because the server may continue to send more messages to the client that requires the *Client Session* component to add those messages into the same queue. The race condition occurs when there are many JACIE assignment statements or control statements while retrieving the message and additional `receive statements`

(message retrieval) are added within the event of retrieving some messages.

In the example, after receiving two messages, `playNum` and `opBid`, other statements for checking certain conditions are written before two more server messages are accepted. While the computation and control statements are executed, the states of two other messages, `bChoice1` and `bChoice2` are undetermined. Most of the time executing the JACIE program that has these codes, a user would experience the indefinite waiting due to system crash.

```
JACIE {
...
on NEWMESSAGE {                                    // listen to any message from the server
  if (MESSAGEID == playNum) {
      receive playNum nPlayer;                     // ◄ actual messages received
      receive opBid oppoBid;
      refresh;
      if (bidHistory[nPlayer][rNumber] == -1) {         // do some checking
         bidHistory[nPlayer][rNumber] = oppoBid;
         if (oppoBid == 1) {
            receive bChoice1 bOne;                 // ◄ receive more actual messages
            receive bChoice2 bTwo;
            refresh;
            bidHS1[nPlayer][rNumber] = bOne;            // do some setting
            bidHS2[nPlayer][rNumber] = bTwo;
         }
      }
  } ...
```

In order for the condition to be safer, the client `NEWMESSAGE` event section cannot have many statements. Avoiding other actions except receiving message is very important. Therefore, the use of a boolean variable might be necessary to set a certain flag for doing other actions. Here is the improved condition for manipulating messages at the client side.

```
JACIE {
   ...
   boolean getdatabid = false;                    // ◄ initialise boolean flags
   boolean getbid = false;
   ...
   on NEWMESSAGE {                                // message received event
      if (MESSAGEID == playNum) {                 // ◄ receive first set of messages
         receive playNum nPlayer;
         receive opBid oppoBid;
         refresh;
         getdatabid = true;                       // ◄ set boolean flag
      }

      if (MESSAGEID == bChoice1) {                // ◄ receive second set of messages,
         receive bChoice1 bOne;
         receive bChoice2 bTwo;
         refresh;
         getbid = true;                           // ◄ set another boolean flag
      } ...
  } ...
  ... // Assignments and control checking are performed 'outside' the 'NEWMESSAGE' event
  if (getdatabid) {                               // ◄ do checking on the first sets
    getdatabid = false;
    if (bidHistory[nPlayer][rNumber] == -1)
      bidHistory[nPlayer][rNumber] = oppoBid;
  }
```

```
if (getbid) {                              // ◀ process second set of messages
  getbid = false;
  bidHS1[nPlayer][rNumber] = bOne;
  bidHS2[nPlayer][rNumber] = bTwo;
} ...
```

Any message received by the client is automatically accepted without any delay. Moreover, no validation or checking is performed during the `NEWMESSAGE` event.

From the examples presented, JACIE II has made an attempt to identify 'unsafe' situations that could lead to race conditions for both server and client. With the presented improved conditions, JACIE example programs and applications (presented in Chapter 5, 6 and 7) are executed correctly and are 'safe'. However, in the future, it is desirable to include locking mechanisms for further improvements.

4.7 Improvements to the JACIE Language

This section covers a few enhancements that have been made to improve the language. The improvements facilitate JACIE programmers in producing shorter programs and better programming environments. The major enhancements to the extended features for interaction and interest management are covered and illustrated in Chapter 5 and 6, respectively.

The process of adding new and modifying the existing language constructs always start with the `jacie.flex` (Figure 4.2) file for declaring new tokens, followed by the new productions added to JACIE grammar in the `jacie.cup` file. The rest of the additions are in the `JACIE code translator` in terms of new Java Classes or modifications to the existing Java compiler.

The modifications and improvements include introducing a new data type that leads to solve the problem of code repetition and producing some informative error messages for the compiler. A mouseclick event is also improved by enabling it automatically for the users who are having the turn control or otherwise, it is always disable.

The discussions in the following subsections are focused on the new data type, code optimisation and the compiler issues. They provide significant information for further enhancements of this language in the future.

4.7.1 Enhancement of Data Types

JACIE supports the primitive data types of Java as well as arrays that include any data of typed *integer*, *double*, *boolean*, *string* and *image*. In JACIE I, the canvas manipulation that deals with the *grid*, which is represented by a Java object, allows a JACIE programmer to have several instructions such as locating a point, defining an object and posting text, lines and images on the canvas. JACIE I does not include a *grid* as a variable type, but allows a

statement called `draw statement` to initialise and draw such objects. Therefore, when compiling this statement, the JACIE compiler must perform three actions, declaring an object, initialising all of its corresponding values and then drawing the object on the JACIE canvas. Below is an example of JACIE I grid objects that are included in the `on canvas` section.

```
client implementation {
 declaration {
   ...
   int[13] currentCard = -1;
   int[13] dummyCard;
   ...
 }
  on canvas{
   //  draw screen window and declare four new grid set
   ...
   draw grid nCard at 70,45 step 13,1 size 20,47 colour black width 1;
   draw grid eCard at 355,53 step 1,13 size 34,17 colour black width 1;
   draw grid sCard at 70,270 step 13,1 size 20,47 colour black width 1;
   draw grid wCard at 20,53 step 1,13 size 34,17 colour black width 1;
   ...
 } ...
}
```

In order to optimise a program that has many *grid* objects, JACIE II has added a new data type called *grid2D*. By making a *grid* a type, a programmer needs to declare any variable of type grid in the declaration section before any instruction concerning such grid is performed. The following code segment illustrates several *grid* objects' declarations in the JACIE `declaration` section before any action on such objects are performed in the `on canvas` section.

```
client implementation {
 declaration {
  shared int[13] currentCard = -1;
  int[13] dummyCard = -1;
  ...
  grid2D nCard;        // ◄ nCard is typed grid2D
  grid2D eCard;
  grid2D sCard;
  grid2D wCard;
  ...
  on canvas {
   ...
   draw grid nCard at 70,45 step 13,1 size 20,47 colour black width 1;
   draw grid eCard at 355,53 step 1,13 size 34,17 colour black width 1;
   draw grid sCard at 70,270 step 13,1 size 20,47 colour black width 1;
   draw grid wCard at 20,53 step 1,13 size 34,17 colour black width 1;
   ...
  } ...
```

Although the code segment for the `on canvas` section above are the same for both JACIE I and II, the compiler code for JACIE II differentiates between creating a new class `grid` from its constructor call `draw grid`. Thus, the actual process and compiler action is different. During `declaration`, the grid object is created and its appropriate values are initialised. When the `draw grid` statement is executed, only one action is taken by the JACIE II compiler, instead of three actions that were performed by the JACIE I compiler. In

this way, any *grid2D* variables can be treated similar to any other types of variables.

4.7.2 Supporting Code Optimisation

The introduction of the type *grid2D* into JACIE II using type grid in the parameter list of a method can lead to the avoidance of repeated code. For example, the following is part of the JACIE I program implementing a bridge game. Here, statements are repeated in determining which player selects a card from the individual's hand.

```
client implementation {
 declaration {...}
 on MOUSECLICK {
   gridX = GETGRIDX;                        // select a grid point x and y
   gridY = GETGRIDY;
   if (!playGame) {...}                     // process bidding action
   else
     if (myTurn && !chooseCard) {           // no card is selected yet
       if (userNumber == 1) {
         if (userNumber == decPlay && playP) {
           if (GETGRID == eCard) {          // check on the canvas grid
             chooseCard = true;
             tricks[turnTrick] = dummyCard[gridY];
             dummyCard[gridY] = -1;
             posCard = gridY;
             gX[turnTrick] = 2;             // choose a card
             gY[turnTrick] = 1;
           ]
         }
         else {
           if (GETGRID == wCard) {          // check on the canvas grid
             chooseCard = true;
             tricks[turnTrick] = currentCard[gridY];
             currentCard[gridY] = -1;
             gX[turnTrick] = 0;             // choose a card
             gY[turnTrick] = 1;
           }
         }
      }
      else if (userNumber == 2) {
        if (userNumber == decPlay && playP) {
          if (GETGRID == sCard) {           // check on the canvas grid
             chooseCard = true;
             tricks[turnTrick] = dummyCard[gridX];
             dummyCard[gridX] = -1;
             posCard = gridX;
             gX[turnTrick] = 1;             // choose a card
             gY[turnTrick] = 2;
          }
        }
        else
          if (GETGRID == nCard) {           // check on the canvas grid
             chooseCard = true;
             tricks[turnTrick] = currentCard[gridX];
             currentCard[gridX] = -1;
             gX[turnTrick] = 1;             // choose a card
             gY[turnTrick] = 0;
          }
        }
      }
      else if (userNumber == 3) {...
      ... //  the same program codes are repeated with different values
```

Clearly there is much repetition of code in the above, which could be avoided using a *method.* With the introduction of the type *grid2D,* it is now possible to have a method with *grid2D* parameters to remove this duplication. Below we see the improved code segment rewritten using a defined method `detgrid` with its appropriate parameters.

```
client implementation {
 declaration {...
  void detgrid(grid2D card1,grid2D card2, int z,int a,int b,int c,int d) {
  ... //  method in client program with grid type as the parameter lists
   if (userNumber == decPlay && playP) {
     if (GETGRID == card1) {              // check on the canvas grid
       chooseCard = true;
       tricks[turnTrick] = dummyCard[z];
       dummyCard[z] = -1;
       posCard = z;
       gX[turnTrick] = a;                 // choose a card
       gY[turnTrick] = b;
     }
   }
   else {
     if (GETGRID == card2) {              // check on the canvas grid
       chooseCard = true;
       tricks[turnTrick] = currentCard[z];
       currentCard[z] = -1;
       gX[turnTrick] = c;                 // choose a card
       gY[turnTrick] = d;
     }
   }
}...
 on MOUSECLICK {
   gridX = GETGRIDX;
   gridY = GETGRIDY;
   if (!playGame) {...}                  // process bidding action
   else
     if (myTurn && !chooseCard) {        // no card is selected yet
       if (userNumber == 1)
         detgrid(eCard,wCard,gridY,2,1,0,1);  // call to the method for grid selection
       else if (userNumber == 2)
         detgrid(sCard,nCard,gridX,1,2,1,0);  // call to the method for grid selection
       else if (userNumber == 3)
         detgrid(wCard,eCard,gridY,0,1,2,1);  // call to the method for grid selection
       else if (userNumber == 4)
         detgrid(nCard,sCard,gridX,1,0,1,2);  // call to the method for grid selection
     }
   }
 }
```

This approach of code optimisation can improve program readability and help in structuring the program. However, the total time for executing such programs may not necessarily be shorter.

4.7.3 Compilation Errors

In compiling JACIE programs, the JACIE compiler has to go through two phases, lexical phase and translation phase, which is similar to other *automated compiler construction*

[109]. Figure 4.9 shows the outline structure of the compiler that is supported by compiler tools for a Java environment, JFlex and JCup in both phases respectively.

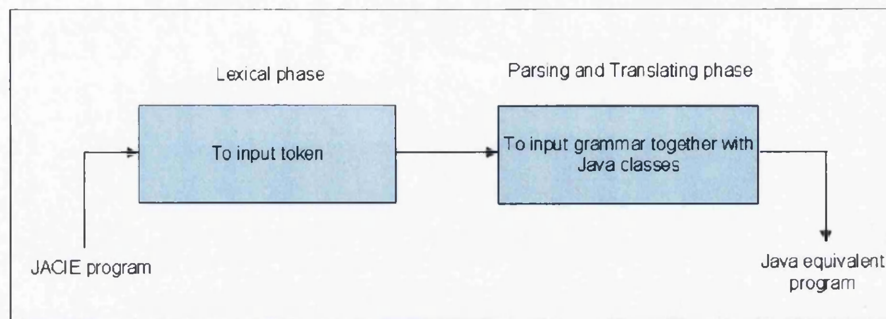


Figure 4.9: Outline Structure of JACIE Compiler.

With both tools, at least three different files should be written before compiling JACIE program to produce the output in the form of Java programs. The files are

- JFlex specification — It contains a list of all *tokens* or *terminal symbols* that can be recognised by the compiler.
- JCup specification — It consists of JACIE *LR(1) grammar* [9] with a list of *productions*.
- JACIE code translator — It contains several Java objects that can be distributed in several files to perform the translation of some specific *non-terminals* defined in the productions provided for JCup. All these Java objects must be in the form of Java classes during the *parsing* and *translating* processes.

When a JACIE programmer writes a JACIE program, the compiler needs to provide some mechanisms for checking any errors that may possibly be made by the programmer. During its first phase (lexical), the compiler checks all the tokens for the language. It is possible for a programmer to make mistakes in writing tokens. For example, in writing a *string* value which is represented by double quotation at the beginning and end of a string, a programmer may miss the close quotation that results in an error detected by the lexical analyser and forces JACIE compiler to stop the execution. Once the JACIE progam is starting to be compiled, the token specifications that has been processed by JFlex is invoked so that the compiler can check on all the tokens found in the program. Figure 4.10 is an example screen shot in compiling a program that has the error of missing close quotation on a *string* value. It shows the command 'jacie', followed by the file name, `ncGeneralise.jacie` is used to start the compilation which is then stopped with an error message, 'Error: Unterminated string at the end of line' that is detected at the end of the program.

For its second phase (parsing and translating), the compiler starts its parse from the beginning of the program by checking the syntax of all statements while at the same time, the translated codes are produced and kept in some files until appropriate instructions in the `JACIE code translator` are executed. In preparing for this phase, JCup must be used to execute and process JACIE grammar specifications. It has a built-in error detection mech-

```
cssiti on cspcgl03: /compsci/partition3/cssiti/jacienc - Shell - Konsole
Session Edit View Settings Help
partition3/cssiti>
partition3/cssiti>
partition3/cssiti> cd jacienc
cssiti/jacienc> jacie ncGeneralise.jacie
setting jflex
setting clpath
Compiling JACIE Compiler
Compiling JACIE Script...
Start compiling JACIE Programme...
Exception in thread "main" java.lang.Error: Unterminated string at end of line
        at JACIECScanner.yylex(JACIECScanner.java:1090)
        at parser.scan(parser.java:2285)
        at java_cup.runtime.lr_parser.parse(lr_parser.java:527)
        at JACIECParser.main(JACIECParser.java:19)
End successfully
cssiti/jacienc> []
New Shell
```

Figure 4.10: Error on the Lexical Phase.

anism for ambiguous and wrong language that prevents the compiler from doing program compilation. Therefore, before a JACIE program is compiled, JFlex and Jcup specification files need to be executed for these tools to recognise JACIE tokens, grammar, and all necessary Java classes used in the language translation processes.

Figure 4.11 shows the process of specifying grammar to JCup by a command 'jcup' to run `jacie.cup` file that contains the specifications. The tool detects ambiguous situations that leads to some conflicts in the parsing process. Therefore, several error messages are printed with the summary of the execution. In this example, 1 error and 4 warnings are found that result in no output being produced. Thus, a JACIE program cannot be compiled since the whole language is not recognised.

Both of the compiler tools provide simple and convenient environments to include error messages. In addition, it is also possible to include some Java codes into the `jacie.cup` file for printing messages on syntax errors found in JACIE programs by specifically stating the location of the error as shown in Figure 4.12. In this example, the JACIE code segment shows that an error occurs on its last line of the statement, 'protocol roundrobin' with a missing semicolon (';') at the end of the line. Therefore, this error stops the compilation process.

With regards to the wrong syntax in Figure 4.12, an appropriate error message should be printed to indicate the error. Figure 4.13 illustrates the action taken by the compiler with error messages to inform the location of the error, 'in line 19, column 4'. However, no details on the correct syntax or information on the error that causes such a problem were given.

JACIE I provided some error messages including all the examples shown above. In general, it can handle almost all syntax errors even though the messages may not provide detailed descriptions. It is significant to have informative error messages that can help programmers to make corrections.

During parsing and translating phase, *semantic* errors may also occur. JACIE II introduces

```
cssiti on cspcgl03: /compsci/partition3/cssiti/jaciec - Shell - Konsole
Session Edit View Settings Help

cssiti/jaciec> jcup
setting jflex
setting clpath
setting java Main
Opening files...
Parsing specification from standard input...
Checking specification...
Building parse tables...
  Computing non-terminal nullability...
  Computing first sets...
  Building state machine...
  Filling in tables...
*** Reduce/Reduce conflict found in state #340
  between send_choice ::= send_choice (*)
  and     send_option ::= TO send_choice (*)
  under symbols: {SEMICOLON}
  Resolved in favor of the second production.

*** Shift/Reduce conflict found in state #340
  between send_choice ::= send_choice (*)
  under symbol SEMICOLON
  Resolved in favor of shifting.

*** Shift/Reduce conflict found in state #340
  between send_option ::= TO send_choice (*)
  under symbol SEMICOLON
  Resolved in favor of shifting.

  Checking for non-reduced productions...
*** Production "send_choice ::= send_choice " never reduced
*** More conflicts encountered than expected -- parser generation aborted
Closing files...
------- CUP v0.10i Parser Generation Summary -------
  1 error and 4 warnings
  171 terminals, 169 non-terminals, and 357 productions declared,
  producing 671 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  3 conflicts detected (0 expected).
  No code produced.
---------------------------------------------------- (v0.10i)
Compiling JACIE Compiler
End successfully
cssiti/jaciec>
New  Shell
```

Figure 4.11: Error on Determining Grammar.

some facilities to detect these errors that are included in the `JACIE code translator`. The error messages are produced on any attempt to do illegal actions. Some illegal actions may stop the program execution while some others can cause the compiler to skip some programming codes without processing the current statement. For cases when the compilation process continues, the appropriate *warning* message is printed. Below are the list of cases that are recognised to produce the *warning* messages. Upon printing these messages, the JACIE compiler will skip the current executable statement and continue with the compilation process.

- An assignment is made to a correct declaration of 'global' variable but no permission is given to use or refer to that particular variable — When a 'global' variable is declared in a client program, the JACIE compiler keeps the list of all these variables in a table. Then, it creates another table to filter these 'global' variables that can be actually referred to and used throughout the translation process. Therefore, any attempt to include the 'global' variables that are not in the former table, into the latter table, causes a semantic error.

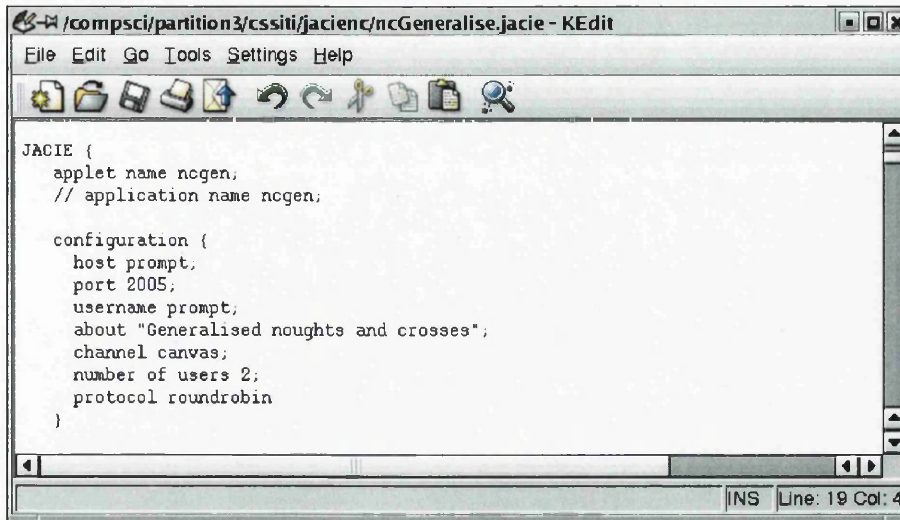


Figure 4.12: Syntax Error in a Program.

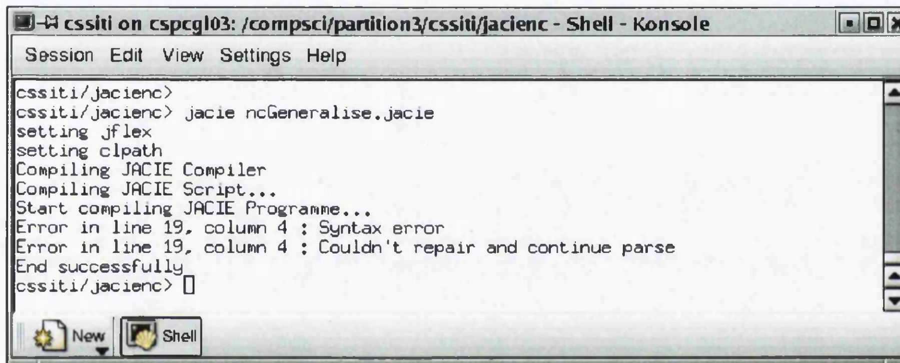


Figure 4.13: Error Message on Syntax.

- Password condition is not stated when accessing a shared variable that requires password — In JACIE II, a statement called *access statement* can include password verifications. The compiler keeps the information on all the variables to be shared based on password. Therefore, any attempt to use such variables without any password 'option' included in the statement would force the compiler to ignore this statement in the code translation process.

The warning message is also printed in the case where there is no 'global' variable used for statements that are supposed to have them. However, in some of these statements, ordinary variables are permitted to be included in the statements. This action is to be processed accordingly without skipping any codes. The *warning* message is just there to inform the possibility of inefficient executing time.

The following list of situations result in the program compilation terminates. This condition requires the programmer to make the correction and rerun the program. Upon dealing with this situation, the compiler would print *error* messages.

- A variable of type *grid2D* or *image* is only allowed to be used in the client program since these types are only significant for canvas manipulation. In fact, the server will not share or use the canvas. Therefore, while doing the code translation on the declaration section of a server program, `JACIE code translator` will detect any attempt to do any illegal action on this matter.

- Any changes of interaction control protocol must be done only in the server program for flexibility in changing user floor during program execution. Therefore, the code translator checks any attempt to include such statement in client program to prevent such action.

- A 'global' variable in JACIE II is declared in a special way so that the compiler can distinguish between local and 'global' variables. These 'global' variables have some special statements for performing value assignment or anything to do with 'global' sharing, therefore, it is illegal to have an ordinary variable exists in a statement called *permission statement.* Figure 4.14 illustrates the attempt to include an ordinary variable `roomNumber` to be used in one of 'global' variable's special statements (the statement is marked by a blue line).

Figure 4.14: A Local Variable is Treated as Global.

Figure 4.15 shows a screen shot of the messages printed by the code translator in dealing with the translation process and the semantic error mentioned above. Since this attempt is treated as an error, the compilation stops and no Java equivalent program is produced.

- In handling 'global' variables, it is also significant to check on any *write access* that is permissible on the variable that has no *read access.* The compiler can stop the execution because without the `read access`, it is impossible to change the unknown value.

```
cssiti on cspcgl03: /compsci/partition3/cssiti/jc2im - Shell - Konsole
Session Edit View Settings Help
Copying nochOff.gif ...
Copying statusOff.gif ...
Copying ipAddOff.gif ...
Copying netMOff.gif ...
Copying gateWOff.gif ...
Copying rson.gif ...
Copying rsoff.gif ...
Copying choose.gif ...
Creating client applet NT1.java ...
Creating NT1Container.java ...
Copying JACIECMethodTable.java ...
Copying JACIEGrid.java ...
ERROR: variable is not declared as shared! :permission statement
End successfully
cssiti/jc2im>
```

Figure 4.15: Error Message on Semantic Checking.

Furthermore, it is also possible to have *syntax* and *semantic* errors after a JACIE program is successfully translated into Java. In this case, all the errors are detected by the Java compiler with appropriate error messages.

4.8 Summary

The JACIE compiler has verified that JACIE scripting language can be a tool for implementing interactive and collaborative applications. Its special features such as simple template style and event-based programming help a Java programmer to write a structured and organised program which also allow them to concentrate on the application development especially the manipulation of ongoing session between user and server.

Major enhancements have been made to JACIE I in extending its features on interaction and interest management besides improving its compiler in reducing the chances of having a race condition at both client and server. A new type of variable called *grid2D* is introduced to ease the programmer in writing codes on the manipulation of *canvas* channel. The introduction of *grid2D* can lead to code optimisation which allows a programmer to include *grid* in a parameter list of a JACIE method call.

Some improvements are also made to `JACIE code translator` that enables the detection of some *semantic errors* while doing program translation. In detecting these errors, it is possible to skip the current statement or stop the compilation with the appropriate *warning* or *error* messages to be printed out immediately.

In general, JACIE high level language features allow flexible program coding and improve readability.

Chapter 5

Interaction Management

Contents

5.1 Introduction 89
5.2 Related Work 90
5.3 The Noughts and Crosses Game and Its Variations 92
5.4 Interaction Management in JACIE 100
5.5 Language Enhancements 111
5.6 Other Protocol Design Issues 116
5.7 Summary 121

5.1 Introduction

Almost all collaborative applications involve the management of interactions among remote users. While the implementation of such software usually involves system level programming interfaces for network communications, it also requires high-level features such as carefully formulated interaction management policies, correctly designed interaction protocols, and sometimes, a consistent and secure means for managing shared data. While many software solutions have been proposed over the years in the context of various applications, these high-level features are rarely supported by software development tools in a coherent manner. Especially, hardly any programming language has language constructs for providing direct support for interaction management.

Interaction management is concerned with the protocols that govern structured interactive activities among multiple users or agents in networked and collaborative environments. The implementation of the protocols is often not a trivial task in the development of an application involving structured communication. The provision of protocol control mechanisms is the weakness of most existing programming languages and development tools. For example, software engineers desire developer support, which can alleviate the burden in implementing correct and reliable codes for managing an online meeting, a teamwork exercise or a

multi-user web camera in a structured manner.

Research on interaction management has been conducted largely in the context of specific management for various applications, including video conferencing [260], group coordination [95], web-based camera control [78], 3D collaborative virtual environments [78] and agent-based collaboration [254, 85]. However, our work provides a general management using a high-level approach for easy programming and managing the interaction.

In this chapter, we attempt to identify a collection of useful interaction protocols that are common in many collaborative applications. We consider an abstraction of various collaborative applications in the form of variations of the noughts and crosses game. We examine the needs of these games for programming interaction protocols, and propose a comprehensive collection of program constructs for supporting interaction management.

5.2 Related Work

Human-human interaction in a distributed collaborative environment often requires coordination in a structured manner where the term *floor control* is often used. In [94, 260, 283], floor control is defined in a more formal agenda and usually must guarantee a mutual exclusion condition among users. Since the environment is often tightly coupled, the floor control mechanism is usually applied to a small number of users. In a small scale system, it usually limits the number of users to collaborate in order to manage the activities properly [202], and for a large scale system, the users are usually formed into groups with similar interests or objectives [97]. When interaction among users exists, the collaboration often requires scheduling to achieve structured interactive activities.

5.2.1 Interaction Protocols

A protocol is a set of rules that governs the order of communication among users that are distributed in a networked system. Interaction in collaborative environments require users not only to communicate, but also to work together and share some resources. The sharing of resources such as data, have to be managed in such a way that in a given state, all users must hold the same shared values. Therefore, when a data update is performed, this event must be in a mutually exclusive manner. It is common in distributed operating system that mutual exclusion issues are discussed [133], and nowadays, these issues are also discussed in the systems that support data sharing and allow this data to be processed concurrently [335]. Concurrent processing is often implemented in several network programming languages. For example, Java has method calls such as lock() and up() to handle mutual exclusion. Hence, to ensure that mutual exclusion always holds in dealing with shared data, users' turns can be scheduled in such a way that they would follow a proper sequence.

The types of floor control commonly implemented in networked collaborative systems include protocol *contention* [191, 135, 138, 25], *master* [124, 94], and *round robin* [78, 94]. *Contention* gives all users the chance to fight for turn control by allowing only one user to

have control over the shared data at any one time. When the user has a turn, this particular user has the right to manipulate the shared data while other users either continue their tasks without any intervention to the shared data or they may be blocked. In situations where other users' tasks are blocked, the protocol is called *first-in first-out* [78]. In operating system concepts, *first-in first-out* scheduling often requires queue or buffer manipulation to keep a list of processes that wait for their turn, which is determined by their arrivals [287, 310]. However, in implementing collaborative applications, most systems that use a 'blocked' mechanism choose a user who gives the fastest response and simply block others who are later have to fight again for another attempt. The *master* protocol has a single dominant controller to determine the floor. In this protocol, the master can freely determine the user turn or it is also possible for the controller to make a selection based on users' requests. The *round robin* protocol can be considered a 'fair' scheme that lets all users have the turn in a sequential order. In implementing this scheme, information on all the users must be kept to ensure every user gets the turn. When a user holds a turn, other users are usually blocked from doing any task [78].

The implementation of floor control is usually performed using software packages [257, 193, 88], programming languages [102, 139] or component-based programming [267]. While the software packages may allow a software developer to program networked collaborative applications based on the provided built in interaction protocols, in particular, programming requires a programmer to write specialised algorithms. Whatever the case, handling interaction protocols requires some basic elements such as flow control algorithms, communication media (e.g. voice, messages, images, *etc.*), and sometimes, buffer management schemes. JACIE I represented one of the first attempts to provide interaction management through high-level language constructs [140, 139]. However, the original set of interaction protocols in JACIE I is quite limited, and more than half of the games discussed as case studies in Section 5.3.4 cannot be directly managed by the original protocols without a noticeable amount of programming effort. We will address this issue in Section 5.4.

5.2.2 Temporal Coordination

Temporal coordination is the management of processes in cooperative work that specifically depends on time factors, viewed as the *state* of a process at a particular time. As time plays a very important role in determining data consistency, time cannot be ignored in designing interaction protocols for collaborative environments. It can also influence the outcomes of the system performance [23]. Bardram [23] defines temporal coordination as an activity that integrates actions in all aspects of distributed collaborations and is influenced by temporal conditions and its surrounding socio-cultural context. He analyses the three levels of temporal coordination namely synchronisation, scheduling and time allocation at a surgical department. Synchronisation refers to the continuous 'rhythm' in the work flow for the dynamic teamwork. Scheduling is viewed as the work plan and goal, while time allocation refers to the need to ensure an adequate match between work demand and resource capability.

In distributed multimedia systems, temporal synchronisation, either intramedia or interme-

dia, that incorporates user interaction into actions, needs to guarantee the quality of the multimedia presentation. Liao and Li [200] focus on single user intramedia synchronisation especially on the rendering of the playback and retrieval schedules. They use the request-on-demand protocol in their design. For the intermedia synchronisation, Mirbel *et al.* [226] propose a method for checking the temporal integrity of interactive multimedia documents that may consist of a variety of multimedia objects, and includes event modelling and composition. In distributed interactive systems, Zhang *et al.* [339] propose a 'timed token' protocol in handling real-time traffic for network communications. For multiagent systems, the agent collaboration tasks which are subject to temporal constraints must be handled accordingly. Hunsberger [156] describes a distributed control of a temporal network among a group of agents.

Recently, more formal approaches have been applied to interaction management, and these include GTRBAC [170], which introduces the notion of *role enabling* and *role activation*, and TILCO[30], which separates the *external* view of the process of interaction from its *internal* view. In determining the state of data consistency and integrity, in Section 5.3.2 we propose in our design a formal notation for the interaction protocol using a discrete temporal function. Such notation is defined by considering the formulation of generalised noughts and crosses games that are used for modelling interaction management in JACIE II.

5.3 The Noughts and Crosses Game and Its Variations

In this section, we first define a set of abstract notations for modelling the noughts and crosses game, its variations and the corresponding interaction protocols in later sections. We then describe the traditional game in 5.3.3. A summary of the various games is given in section 5.3.4 where we highlight their main protocol features in turn control and domain control, linking them with real life collaborative applications. We relegate the details of the variations of the noughts and crosses games to Appendix A.

5.3.1 History of the Noughts and Crosses

According to [7], the history of this game was probably started during the Roman empire, but there was no strong evidence to prove this. While in the United Kingdom, this game has been played for several centuries. It appears that the first software program to play this game was designed by A.S. Douglas at Cambridge University for the EDSAC computer in 1949, as part of his PhD thesis. This game is also called Tic Tac Toe in some other countries. Although the basic game seems quite simple, there are many variations of the game with different levels of complexity. Some of the varieties of the traditional game are Tic Tac Tower, 3D tic-tac-toe, 4D tic-tac-toe and 2D game variations (with different materials, layout, or applications, *etc.*), Figure 5.1 shows some examples of the noughts and crosses type games. These are two players games with various turn controls and domain controls.

Some researchers use this game as a case study or implement this game for entertainment or as a teaching aid. Addison and Thimbleby [6] claims that this game was one of the first

computer programs. Gibson [130] used this game written as a Java applet to teach programming to primary school students. This game has data that must be known (or *shared*) by both players and they need to follow some rules to place their symbols on the game board. Therefore, it is useful to look into the interaction and the resource sharing when this game is played as an interactive collaborative system.

5.3.2 Definitions

The standard noughts and crosses game can be generalised in many different ways, including increasing the size of the game board, altering the rules governing the game, changing the definition of winning state or the allowed states and valid symbols of each cell, and so on. In our generalisations, we choose to give a high degree of freedom to the specification of game rules in order to explore a variety of interaction protocols and cover a broad range of applications. We restrict ourselves to using only two basic symbols, namely nought and cross in our discussions, which enables us to maintain a reasonable level of abstraction in order to focus on the interaction management in the games rather than on the games themselves. The extension to an arbitrary set of symbols is relatively trivial in terms of both specification and implementation, but it generally does not bring much benefit to the discussion of interaction management.

Once we move away from the traditional game, it is inevitable that having a 3×3 game board is too restrictive. Therefore, a ***generalised game board*** for noughts and crosses is defined over a grid of $N_x \times N_y$ cells. Each cell $< i, j >$ may exhibit one of the three basic visual states, namely *empty*, with a *nought*, $\circ$, or with a *cross*, $\times$. During a game, relative to a player, each cell may also be in two different modes indicating whether or not the visual state of the cell is modifiable by the player concerned. We therefore consider six valid states for cells, whose visual representation are □, ■, ⊡, ◙, ⊠, ▣, where formally, the set of symbols are elements of the set $OX \times Mod$ where $OX = \{empty, \circ, \times\}$ and $Mod = \{mod, unmod\}$. So, for example, ⊡ and ◙ are the visual representations of $< \circ, mod >$ and $< \circ, unmod >$ respectively. A white background indicates that the contents of the cell is *mod*ifiable and a black background that it is *unmod*ifiable.

A single game board G is then a function indicating the state of each cell and so is of type

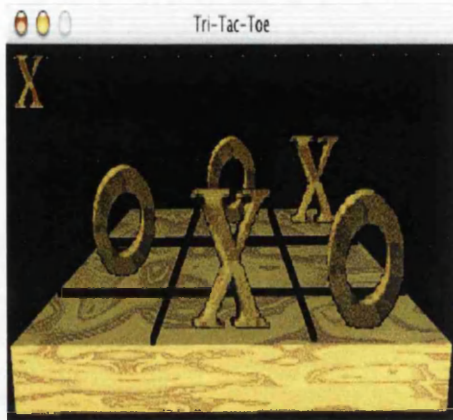
$$G : \mathbb{N}_x \times \mathbb{N}_y \rightarrow OX \times Mod.$$

where we let $\mathbb{N}_x = \{1, \ldots, N_x\}$ and $\mathbb{N}_y = \{1, \ldots, N_y\}$. We use '*.ox*' and '*.mod*' to select components of the range. G is said to be *empty*, if the state of every cell is empty (either modifiable or unmodifiable), i.e.,

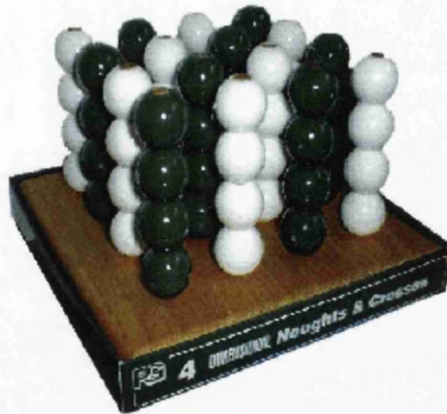
$$\forall < i, j > \in \mathbb{N}_x \times \mathbb{N}_y (G(i, j).ox = empty).$$

G is said to be *active*, if there is at least one modifiable cell in G, i.e., $\exists < i, j > \in \mathbb{N}_x \times \mathbb{N}_y (G(i, j).mod = mod)$. Otherwise, G is said to be *inactive*.

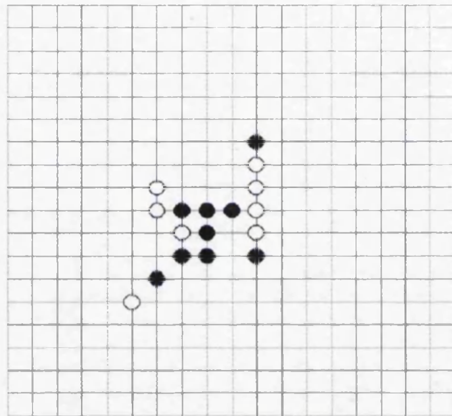
As we are interested in the change in the board over time, we extend the notation and introduce a time parameter t which ranges over a semi-bounded domain $T = [t_{start}, \infty)$, where



3D Noughts and Crosses [295]

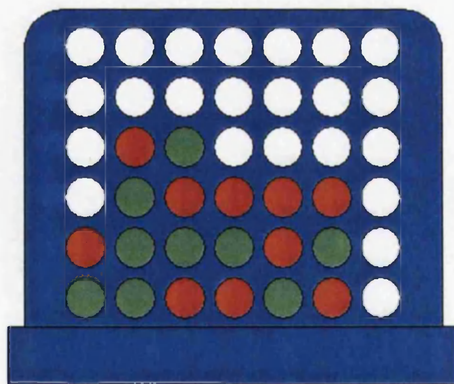


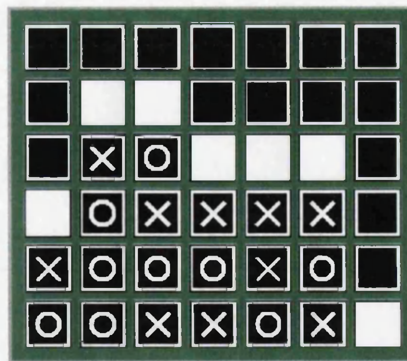
4D Tic-Tac Toe [165]



Five-in-a-line

Three Stones [208]





Connect 4 Board and Its Representation

Figure 5.1: Various Versions of the Noughts and Crosses Type Games.

t_{start} is the time when a game commences. Hence we use $G(t)$ to denote the state of the game board at time t with G having domain $T \times \mathbb{N}_x \times \mathbb{N}_y$. In a paper-based noughts and crosses game, or its implementation on a centralised single processor environment, there should be a single board at any specific time t. We call this the *primary* game board and denote it by $G_{prim}(t)$. Nevertheless, in a distributed, collaborative environment involving a collection of processors, identical boards cannot easily be guaranteed across the processors. Let $p_0, p_1, p_2, \ldots, p_K$ be $K + 1$ processors in such an environment. At a particular time $t \in (t_{start}, \infty)$, a user associated with a particular processor p_k has access to a particular version of $G_{prim}(t)$, which is denoted as $G_k(t)$. All $G_k(t)$, with the same t, together with the corresponding primary state $G_{prim}(t)$, are called *homothetic states* of the game board. Formally G is now a function of type

$$G : T \times \mathbb{K}_0 \times \mathbb{N}_x \times \mathbb{N}_y \to OX \times Mod$$

$\mathbb{K}_0 = \{prim, 0, 1, \ldots K\}$ where $G_k(t)(i, j)$ is shorthand for $G(t, k, i, j)$ and we can consider each $G_k(t) : \mathbb{N}_x \times \mathbb{N}_y \to OX \times Mod$. In any distributed collaborative environment, it is not necessary, and often impossible, to ensure identical homothetic states within $G_k(t)$ for all k. However, it is necessary to have correctly designed and implemented interaction protocols to ensure a consistent state transition of $G_{prim}(t)$ and $G_k(t)$ for each individual p_k.

At any particular time, p_k can attempt to perform an action by trying to assign a new state in cell $< i, j >$. In this chapter, we consider only two types of actions, attempting to assign an $\circ$ or $\times$ in cell $< i, j >$. The actions are described by the function *act* of type

$$act : T \times \mathbb{K}_0 \times \mathbb{N}_x \times \mathbb{N}_y \to OX \cup \{no_op\}.$$

no_op is used to indicate that no real action is being performed, that is, no attempt is is being made to change the state of a cell.

We can take a curried form of G to encapsulate all the boards at time t, that is we take

$$\mathcal{G} : T \to \mathbb{K}_0 \times \mathbb{N}_x \times \mathbb{N}_y \to OX \times Mod$$

where $\mathcal{G}(t)(k, i, j) = G(t, k, i, j)$. Let $A(t_a, t_b)$ be the graph of *act* where time t is restricted to $[t_a, t_b)$, that is,

$$A(t_a, t_b) = \{< t, k, i, j, act(t, k, i, j) > | t_a \le t < t_b, k \in \mathbb{K}_0, i \in \mathbb{N}_x, j \in \mathbb{N}_y\}.$$

This records all the actions during the time interval $[t_a, t_b)$. An *interaction protocol*, therefore, is essentially a temporal function, ψ, that computes all homothetic states $\mathcal{G}(t)$ as:

$$\mathcal{G}(t) = \psi(t, A(t_{start}, t), \mathcal{G}(t_{start}))$$

where $\mathcal{G}(t_{start})$ represents the initial homothetic states of the game board on different processors. As it is not feasible to maintain a continuous change of $\mathcal{G}(t)$, on a specific processor p_k, $G_k(t)$ is updated in a discrete manner as

$$G_k(t_1), G_k(t_2), \ldots, G_k(t_s), G_k(t_{s+1}), \ldots,$$

where $t_{start} < t_1 < t_2 < \ldots < t_s < t_{s+1} < \ldots$, which do not necessarily have a regular time interval. The state represented by $G_k(t_s)$ defines a *temporal action domain* for p_k, during the period $[t_s, t_{s+1})$.

Ideally, one would like to simplify ψ to a function that operates on a discrete time series, $\ldots < t_s < t_{s+1} < \ldots$, as $\mathcal{G}(t_{s+1}) = \psi(t_{s+1}, A(t_s, t_{s+1}), \mathcal{G}(t_s))$. However, the time series operating at each processor is based on its local events (including clock, interaction and communication events), hence is not synchronised with that of other processors. This poses the major challenge to the design and implementation of any interaction protocols for net-centric collaborations.

In a distributed collaborative environment, the implementation of ψ has to be realised using a set of concurrent sub-functions, $\psi_a, \psi_b, \ldots$, which operate in different processors in a distributed and co-ordinated manner. To facilitate more intuitive and coherent discussions in the following sections, we assume a client-server model, with one server p_0, and K clients $p_1, p_2, \ldots, p_K$. We also assume there is only one user (player) at each client, and only one sub-function ψ_k at each processor $p_k, 0 \le k \le K$. In practice, for instance in JACIE, a ψ_k can be realised by multiple threads or processes. We also take a 'coarse' view of the time series in our presentation of algorithmic operations for managing protocols, rather than a 'fine' view which would separate any algorithmic step into many tiny small discrete events at the micro-instruction level.

5.3.3 Traditional Noughts and Crosses

In a traditional noughts and crosses game, two players are assigned to nought and cross respectively. Each player takes it in turn to place his/her symbol, either a nought or a cross, in one of the empty cells. The player, who first completes a 3-cell line horizontally, vertically or diagonally, wins the game. It is possible for a game not to have a winner.

We can generalise the game to a $N_x \times N_y$ game board. Each player may have $\delta_{turn} \in (0, \infty)$ seconds to complete a move. A player who first completes a w-cell winning line wins the game. Referring to the primary board, a winning line is a set of contiguous cells, that satisfy one of the following conditions, (a) for a horizontal line, (b) for a vertical line, and (c) or (d) for a diagonal line:

$$\exists i_0 \in \mathbb{N}_x, j_0 \in \mathbb{N}_y$$
$$\text{(a)}\ \forall l(0 < l < w \rightarrow G_{prim}(t)(i_0 + l, j_0).ox = G_{prim}(t)(i_0, j_0).ox)$$
$$\text{(b)}\ \forall l(0 < l < w \rightarrow G_{prim}(t)(i_0, j_0 + l).ox = G_{prim}(t)(i_0, j_0).ox)$$
$$\text{(c)}\ \forall l(0 < l < w \rightarrow G_{prim}(t)(i_0 + l, j_0 + l).ox = G_{prim}(t)(i_0, j_0).ox)$$
$$\text{(d)}\ \forall l(0 < l < w \rightarrow G_{prim}(t)(i_0 + l, j_0 - l).ox = G_{prim}(t)(i_0, j_0).ox)$$

For cells on the boards $G_k(t)$, the game requires the use of four states, namely □, ■, ◙ and ⊠ as defined in Section 5.3.2. The interaction protocol is essentially a round robin mechanism (see also Section 5.4.1), which can be found in many practical applications, including a variety of board games, question-answer based user interfaces, option menu interactions,

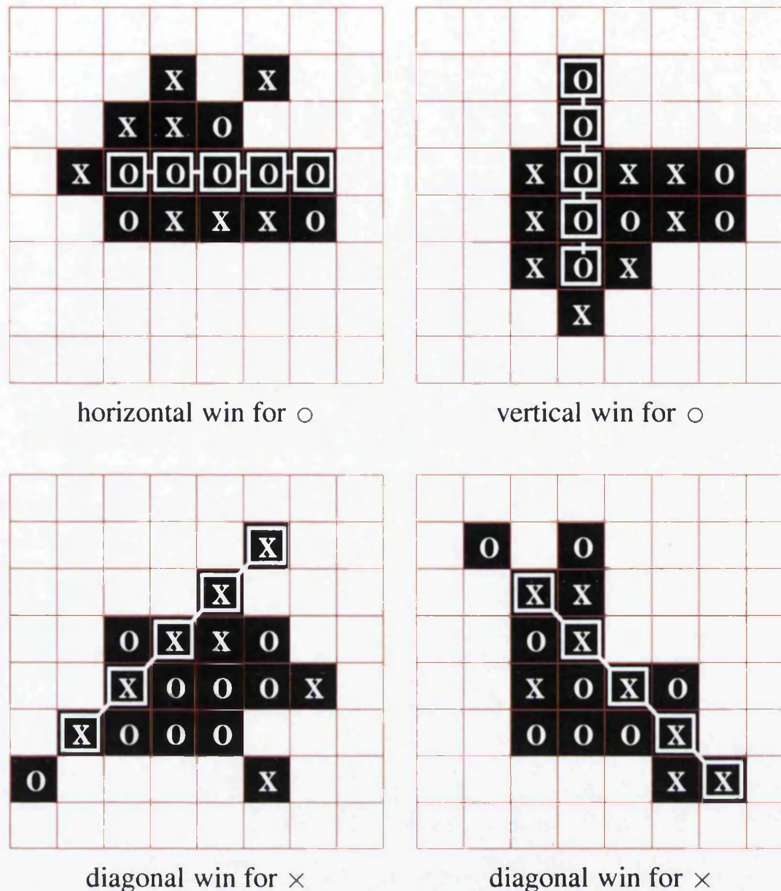


Figure 5.2: Win Conditions.

and automatic tele-information service. The turn time (or timeout) management can also be found in applications such as fault-tolerance automatic tele-information service (i.e., handling a user's hesitation) and menu facilities in many television sets.

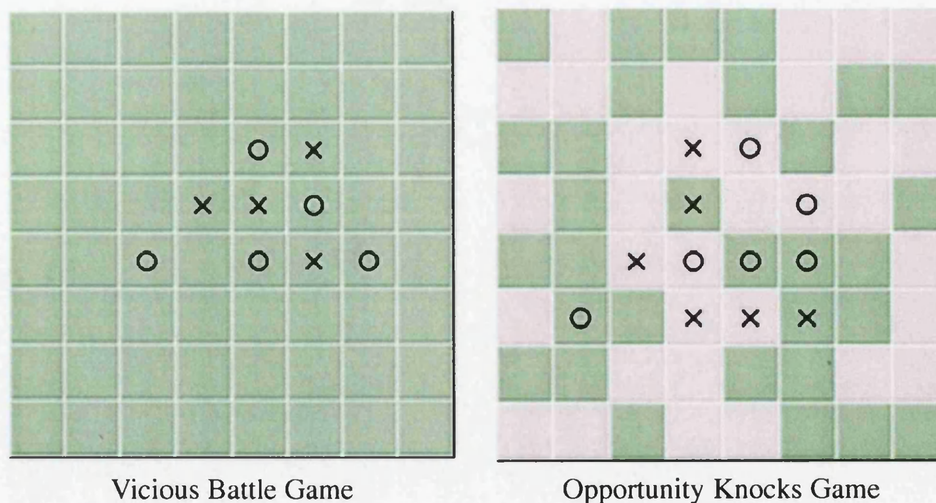
For example, in a question-answer based user interfaces application, a user is given a set of questions that need to be answered. The *round robin* protocol is applied to the user and the system where a user needs to produce the answer and the system produces the question. During this process, such question and answer are unique without any repetitions or changes in the answer given previously. The concepts are similar to what has been represented by the four cell states, □, ■, ◙ and ⊠.

5.3.4 Summary of Variations

Traditional noughts and crosses captures a simple interaction protocol commonly used in a variety of collaborative applications, such as question-answer dialogues, many different types of board games, and online information services. However, it is not adequate enough to represent other complex collaborative activities. For instance, when two or more people

are working on a document collaboratively, it is not necessary, often not desirable, for them to follow strictly a round robin order. Many collaborative activities involve a 'master' who controls the turn of other participants. In order to accommodate such collaborative applications, we have studied many other forms of board games, such as five-in-a-line and three stones, which exhibit the main features of the noughts and crosses game. In addition, we have invented several new variations, including hasty battle, dictator's entertainment, *etc.* The detail description of these games is provided in Appendix A. Though some of the invented variations may not be suitable for a real competitive contest, they effectively capture some common interaction protocols used in our everyday life. For example, gentlemen's battle captures a friendly interaction protocol used in many non-competitive collaborations, while vicious battle exhibits a selfish interaction protocol adopted by many in accessing shared networked resources.

Here, we choose a more abstract approach based on these variations, instead of application-based design. These games enable us to focus on having a comprehensive collection of turn control protocols. They are simple enough for us to concentrate on the interaction requirement of the language and applications. At the same time, the real applications that are associated with these variations are identified. Therefore, these games and JACIE language act as the platform for the design and experimentation. Table 5.1 summarises and compares these variations of the noughts and crosses game. In the table, we highlight, for each game, the relevant interaction protocol to be discussed in Section 5.4, the main cell states to be considered in the management of temporal action domains, and applications typified by and reflected in the game. The cell states for protocol contention in vicious battle (Appendix A.5) is the only protocol that does not have unmodifiable cell, ■. This represents the continuous turn control for all the users, while the rest of the protocols have shown the need for a proper scheduling in determining users' turns. A 'blocked' on other users' tasks is one of the implementation techniques.



Vicious Battle Game | Opportunity Knocks Game

Figure 5.3: Screenshots of Two Noughts and Crosses Games.

Figure 5.3 shows the screenshot of vicious battle (Appendix A.5) and opportunity knocks (Appendix A.9) games. In the vicious battle, all the cells are green to indicate that users

always have the turn to play. For the cells with ⧇ and ☒, green also indicates that these symbols can be overwritten at any time. In the opportunity knocks, there are green and red cells that can contain one of both symbols, ○ and ×. Since green represents the modifiable cells, pink is the unmodifiable cells. All the full cell states are shown here where in any green cell, users can put their symbols at any empty cell or overwrite any symbol. Any attempt to do the same action to any pink cell is not permissible.

Name (Section)	Turn Control	Cell States Used
Generalised game (5.3.3)	round robin	□, ■, ◙, ⊠
Applications : question-answer user-interface, menu, board games, tele-info. service.		
Five-in-a-line (A.1)	round robin	□, ■, ◙, ⊠
Applications : question-answer user-interface, menu, board games, tele-info. service.		
Connect-4 (A.2)	round robin	□, ■, ◙, ⊠
Application : online form filling.		
Three stones (A.3)	round robin	□, ■, ◙, ⊠
Application : online form filling.		
Hasty Battle (A.4)	contention	□, ◙, ⊠
Applications : shared database access, printing queueing.		
Vicious Battle (A.5)	contention	□, ⧇, ☒
Applications : shared whiteboard, co-authoring.		
Gentlemen's Battle (A.6)	tapping	□, ■, ◙, ⊠
Applications : discussion forum, tele- and video conferencing		
Dictator's Entertainment (A.7)	master	□, ■, ◙, ⊠
Applications : e-learning class and testing, structured online meeting.		
First-come, first served (A.8)	reservation	□, ■, ◙, ⊠
Applications : document downloading, web camera control.		
Opportunity Knocks (A.9)	round robin	□, ■, ⧇, ◙, ☒, ⊠
Application : emotional IQ test.		
Secret Switch (A.10)	round robin	□, ■, ⧇, ☒
Application : distributed database access.		
Group games (A.11)	round robin & group protocol	□, ■, ◙, ⊠
Applications : group work and team games.		

Table 5.1: Variations of the Noughts and Crosses Game, and Their Main Features.

This list represents only a small proportion of possible variations. Many other games, such as Go, Othello (or Reversi) and Droughts, can also be thought of as some kind of complex variation of the noughts and crosses game. As the complexity concerns largely the rules, movement and strategies, rather than interactions, it is not essential to include them in the discussions. Most of the variations can be moderated using various timers. We have only selectively discussed these timers in a few variations to avoid an unnecessary coverage of all possible combinations.

5.4 Interaction Management in JACIE

In this section, we give the functional specification of each of the six protocols, and describe its use and implementation. We give the syntactic specification of the language construct for each interaction protocol. As JACIE is a scripting language, most arguments (or extensions) of a protocol are optional, which facilitate 'fast scripting' for simple and commonly-used protocols, and the extensibility when introducing new variations and extensions.

5.4.1 Round Robin

protocol roundrobin [turn [timer] δ_{turn}]
[overall [timer] $\delta_{overall}$] [silence [timer] $\delta_{silence}$]
[[max] action σ_{action}] [rest [timer] δ_{rest}]

turn pass

action start ... action end

Round robin is one of the most commonly used protocols for managing interactions. In a basic round robin protocol, only one user, at most, is authorised to alter the states of the game board at any time during a game. In other words, at any specific time, t, we have either:

$$\exists k, 0 < k \leq K \text{ such that a game board at client k, } G_k(t) \text{, is active, and } \forall l \neq k, \text{ we have } G_l(t) \text{ is inactive}$$

or

$$\forall k (0 < k \leq K \rightarrow G_k(t) \text{ is inactive}).$$

The protocol management function ψ resides mainly at the server p_0 as ψ_0. This will initialise $\mathcal{G}(t_1)$ as follows:

$$(i)\ \forall i \in \mathbb{N}_x, j \in \mathbb{N}_y (G_0(t_1)(i,j) \leftarrow < empty, mod >),$$
$$(ii)\ \forall i \in \mathbb{N}_x, j \in \mathbb{N}_y (G_k(t_1)(i,j) \leftarrow < empty, unmod >), i = 2, \ldots K, \text{and}$$
$$(iii)\ G_1(t_1) = G_0(t_1).$$

Upon a valid action from the authorised client p_1 at t_2, ψ_0 computes a new active $G_2(t_3)$ for client p_2, and a new inactive $G_k(t_3)$ for each client $2 < k \leq K$. The order of clients being activated is organised according to the order of their registration with the server, and is on the first-come, first served basis in a circular manner. For efficiency reasons, JACIE facilitates an additional sub-function ψ_k at each client p_k, which validates each action $act(t,k,i,j)$ against the current $G_k(t_c)$, where $t \geq t_c$. Upon a valid action, ψ_k automatically replaces $G_k(t_c)$ with an inactive $G_k(t_{c+1})$, by making $G_k(t_{c+1})(i,j).mod \leftarrow unmod, \forall i,j$ rendering all subsequent actions at p_k invalid until an active $G_k(t_f)$ is received from the server at time $t_f > t_{c+1}$.

In JACIE, the behaviour of the basic round robin protocol can be modified with three

main timeout timers, namely `turn timer`, `overall timer` and `silence timer`, and another timer called `rest timer` that must be declared together with the multiple `actions` selection. Such timers are particularly useful in applications where more precise control of time, or more efficient use, of shared resources is necessary, such as structured meeting, various board games, and online teamwork exercises.

The `turn timer` is governed by a variable δ_{turn} (default ∞). When it is switched on, with a finite value of δ_{turn}, it can be used to restrict each client to perform valid actions within δ_{turn} seconds after gaining the turn control. The turn timer is managed by each client sub-function ψ_k, and is activated upon receiving an active $G_k(t_c)$. It generates a time-out event at $t_{c+1} = t_c + \delta_{turn}$, which leads to the replacement of $G_k(t_c)$ with an inactive $G_k(t_{c+1})$.

The `overall timer` is governed by a variable $\delta_{overall}$ (default ∞) and it can be used to restrict each client to have a fixed amount of total time for holding a turn. The overall timer is managed by each client sub-function ψ_k, and is activated upon the session start. If the user's turn time is accumulated to a total equal to $\delta_{overall}$, the user's session will be ended.

The `silence timer`, $\delta_{silence}$ (default ∞) represents the maximum amount of continuous time that the client in control is allowed to remain inactive. The timer starts upon receiving the turn, and is reset upon every interaction initiated by the client. Once the silence timer has expired, the turn is automatically passed back to the next client via the server.

The maximum number of actions in each turn, which is set with `[max] action` (default 1), represents the number of actions that a client can make during the given turn. This option is introduced to accommodate circumstances, such as in an application, when it is sometimes more efficient and cost effective for a party to perform multiple actions before passing the turn to another party. Upon choosing the multiple actions in a turn, the `rest timer`, δ_{rest} (default 0), can be set to any value by the JACIE programmer. The rest timer, which defines the minimal timing gap between two consecutive actions, can be used by JACIE programmers to moderate between the fast speed of user interaction (e.g., through the keyboard and the mouse) and the relatively slower speed of network communication. If there is no specification of `[max] action`, the parameter of `rest timer` has no effect as there is only one action per turn.

The following JACIE code shows how the protocol is implemented with the `turn timer` is set to six time units and `silence timer` set to three units in the generalised noughts and crosses game 5.3.3.

```
JACIE {
   applet name ncGen;
   configuration {
     ... // other declarations
     number of users 2;
     protocol roundrobin turn 6 silence 3;   // ◄ interaction protocol
   }   ...
   client implementation {
      declaration { ... }
```

UNIVERSITY OF WALES SWANSEA LIBRARY

```
      ...
      on session { ...
        on MOUSECLICK { ...
           action start;                        // ◄ signifies the mouseclick action
           ... // get x and y point on canvas if click on the right cell
           criticalsection start;               // ◄ continuously send messages to server
           ... // send some messages to the server and update board
           action end;                  // ◄ similar to pass the turn control if number of action = 1
        } ...
      } ...
   }
   server implementation {
      declaration {  ... }
      ...
      on session {
         on TURN {                              // ◄ process current in control user
            ...  // process an action
         }  ...
      }  ...
   }
}
```

The statement `start action` is usually included in the program if the `silence timer` is in the protocol option. In this example, the `start action` is activated by a mouse-click. Therefore, any mouse-click within 3 units of time allows the user to continue with the turn until the `turn timer` expires. Otherwise, the user turn is terminated and the turn is passed to the next user. Upon successfully making an action within the time limit, the `end action` triggers the end of the user turn. Since the number of actions is not specified in this example, and by default, only one action per turn, the `end action` is equivalent to `turn pass`.

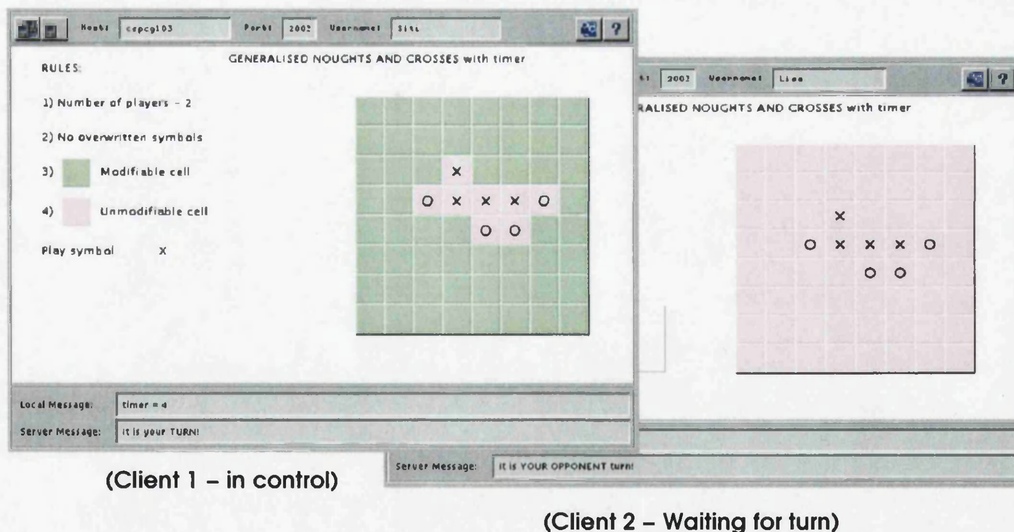


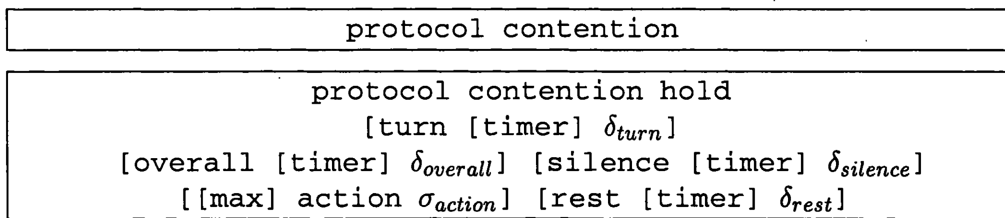
Figure 5.4: Screenshots of Generalised Games.

Figure 5.4 shows the output of this program for both players at time t. The user in control has a message in the JACIE *Local Message* to show the timer counter for easily acknowledging the time limit. A text message is also displayed in JACIE *Server Message*, 'It is your

TURN' to represent the user's current turn. The game board state with □ shows that there are some modifiable cells on the board. While at the opponent, the state on the board is ■ to avoid the player making any grid selection. The text message on the user interface, 'It is YOUR OPPONENT turn' indicates that the opponent has the current turn control.

The statements `action start` and `action end` must be used in conjunction with `[max] action` where $\sigma_{action} > 1$ or a finite value for the `silence timer`. The `action start` resets the `silence timer` automatically, while `action end` indicates the completion of one action cycle and increments the action count. The first action cycle is started by an activity defined in an application, such as keyboard or mouse events, text input or opening of a communication channel. In the case where the `silence timer` expires before the `turn timer` ended, the user's turn is 'forcefully' terminated. By default, only one action is permitted in each turn, therefore, the `action end` has the equivalent effect as a `turn pass` statement if the value of `action end` is not set.

5.4.2 Contention



`protocol contention`

`protocol contention hold`
`[turn [timer]` δ_{turn}`]`
`[overall [timer]` $\delta_{overall}$`] [silence [timer]` $\delta_{silence}$`]`
`[[max] action` σ_{action}`] [rest [timer]` δ_{rest}`]`

Interactions under a *contention* protocol is perhaps considered as almost unmanaged since all clients are authorised, at almost any time during a game, to alter the game board, though it does not guarantee the success of every action.

The protocol management function ψ almost totally resides at the server, which initialises the client action domains as:

$$(i)\ \forall i \in \mathbb{N}_x, j \in \mathbb{N}_y (G_0(t_1)(i,j) = \square),$$
$$(ii)\ G_k(t_1) = G_0(t_1), k = 1, \ldots, K.$$

With a contention protocol, it is particularly difficult to maintain a consistent set of homothetic states at any time t during a game, due to the non-deterministic behaviours of users, client computers and the network. It is possible, $G_k(t)$ may vary substantially from client to client.

Each user action $act(t,k,i,j)$ is first validated by the client sub-function ψ_k against the current $G_k(t_c)$. It is then forwarded to the server, entering at the end of an action queue. As it is possible that actions from different clients may not compatible with each other because of concurrent activities, all actions need to be examined again at the server. As long as the queue is not empty, the server sub-function ψ_0 continues to fetch actions one by one from the head of the queue. Each action is revalidated against the current game board state, $G_0(t_s)$, stored at the server end. If the action is invalid, it is simply discarded. Otherwise, ψ_0 computes a new $G_0(t_{s+1})$ and propagates it to all clients.

As all $G_k(t)$ are active most of the time, it is possible for the protocol management to become highly ineffective because of a combination of continual user actions and delays in network communication. In such a case. the `rest timer` can be used to ease the pressure on protocol management by restricting each client to 'rest' for a small period between two consecutive actions. It is governed by a variable δ_{rest} and is managed by each client sub-function ψ_k. Upon the receiving of a user k action at t_c, $G_k(t)$ becomes inactive immediately and ψ_k activates the rest timer while processing the action. All actions issued by the user during $(t_c, t_c+\delta_{rest})$ are discarded by ψ_k. $G_k(t)$ becomes active again at $t = t_c+\delta_{rest}$.

Besides the rest timer, the contention protocol can also be influenced by the `overall timer`. For the `overall timer`, the function ψ_k will activate the timer upon the session start. When the overall time expires, ψ_k will stop all the clients' activities. With this timer, clients are forced to act even faster, not only to get their data to reach the server quickly, but also to fight against the time given to finish their activities in the specified time.

One variation of the protocol is `contention hold`, which allows a client to hold onto a turn once the client's action is accepted. All actions from clients will be placed in a *turn-waiting queue*, and an automatic filtering mechanism ensures that only one action per client is allowed. Hence, the contention takes place in the turn-waiting queue, rather than the action queue. When a client is holding a turn, all the subsequent actions of the client will be placed in the *action queue*, and be processed ahead of other actions in the turn-waiting queue.

The following code shows the hasty battle noughts and crosses game. It presents the contention protocol declaration with the `overall timer` option, where the total time of ten units is allocated to a user to play the whole game.

```
JACIE {
   applet name ncHB;
   configuration {
     ... // other declarations
     number of users 2;
     protocol contention overall 10;   // ◄ interaction protocol
   }  ...
   client implementation {
      declaration { ... }
      ...
      on session { ...
        on MOUSECLICK { ...
            ... // get x and y point on canvas
            ... // send some messages to the server
        } ...
      } ...
   }
   server implementation {
      declaration {  ... }
      ...
      on session {
           ... // validate board and process an action
           ... // broadcast updated board information
      }  ...
   }
```

}

It has a slight difference from the *generalised noughts and crosses* in validating the game board. Since the turn control is always given to all the players, the client board game is updated after receiving the validation from the server. For changing the turn protocol in the `Configuration Section`, it is only a matter of changing one line of code.

Contention protocol has another choice available, called the contention hold, also known as `first come first served` protocol. With this protocol, `start action` and `end action` are also included as in the *round robin*. At the start session, all the players have the turn control. At the `start action`, a message is sent to the server to place the player in the server *turn-waiting queue*. The `start action` is activated by a mouse-click event. Once the server receives this message, the user turn is set and at that time, only one player gets the turn, and the other player's turn is blocked. When the current player in control finishes the turn, which is in this example six units of time for a turn or one action is completed (which ever comes first) another message is sent to server to acknowledge the completion.

```
JACIE {
   applet name  ncVB;
   configuration {
     ... // other declarations
     number of users 2;
     protocol contention hold turn 6;   // ◀ interaction protocol
   }  ...
   client implementation {
      declaration { ... }
      ...
      on session { ...
        on MOUSECLICK { ...
           action start;
           ... // process an action
           action end;
        }  ...
      }  ...
   }
   ...  // the remaining code is similar to generalised game, Section 5.4.1
}
```

As the `turn timer` usually has to be set in this protocol, the program coding is very similar to the *round robin* protocol. However, the slight differences are that the order of the turn exactly follows the order of the processes in the *turn-waiting queue* and all the players are always have the turn control if there is no player in action.

With the *contention hold*, the `turn timer` and the `silence timer` influence the client turn. Similar to the roundrobin protocol, the `turn timer` is used to indicate the length of time for a turn and the `silence timer` is to stop the client turn if no action is initiated. The `overall timer` can be used for the total length of time for holding turn control. The `rest timer` is only applicable for the declaration of multiple actions in a turn.

5.4.3 Reservation

```
protocol reservation
[turn [timer] δturn] [overall [timer] δoverall]
[silence [timer] δsilence] [[max] action σaction]
[rest [timer] δrest]
```

```
turn request Boolean
```

This type of protocol allows clients to make their requests to the server for their turns. The requests are made, in a round robin fashion, by all the clients at the start of the session. Any request is put into a queue. After all the clients have the chance to make their requests, the server checks the content of the request queue. If the request queue is not empty, the protocol *reservation* is set and the turn control will follow the requests in the queue. After all the requests are fulfilled, the protocol is set to *round robin* again to look for another set of requests. This protocol can also be modified by the timer choices given before. The statement `turn request` is used by the client in the round robin request mode with the Boolean set to TRUE to ask for a turn. Otherwise, without any request from a client, by default, the Boolean is FALSE.

For the reservation protocol, the JACIE statement `turn request` indicates whether the current user wants to request a turn (with value TRUE) or not (value FALSE). During the turn, there exists two different modes, either the `request turn` mode or the actual `action` mode.

```
JACIE {
   applet name  ncRev;
   configuration {
     ... // other declarations
     number of users 2;
     protocol reservation;   // ◀ interaction protocol
   }  ...
   client implementation {
      declaration { ... }
      ...
      on session { ...
        on TURN {
           ...          // get the current state set by the server
        }
        on MOUSECLICK { ...
           if (state == TURN_REQUEST) {
              ... / select button to make option
              if (grid == REQUEST)
                 turn request TRUE;      // client requests to play
              else
                 turn request FALSE;
           } else   // set state to PLAY
           ... // update the board
           turn pass;
        }  ...
      }   ...
   }
      ... // the remaining code is similar to generalised game, Section 5.4.1
}
```

The current user in the actual `action` mode, will click on any board cell to proceed with the game. Whenever the user is in the `request turn` mode, the game board is inactive and he/she can only click a specific button on the canvas to choose whether to request a turn or not.

5.4.4 Master

protocol master [server] [random | userdefined] [turn [timer] δ_{turn}] [overall [timer] $\delta_{overall}$] [silence [timer] $\delta_{silence}$] [[max] action σ_{action}] [rest [timer] δ_{rest}]

protocol master client η_{client} [turn [timer] δ_{turn}] [overall [timer] $\delta_{overall}$] [silence [timer] $\delta_{silence}$] [[max] action σ_{action}] [rest [timer] δ_{rest}]

turn set client η_{client}

For this protocol, the *master*, whose job is to determine the turn control, has all the power to set the turn. The default master is the server. The random choice indicates that the server determines the order of the user turn in no specific order or is randomly assigned. Like the *round robin* protocol, only the state □, can be changed, and no overwritten value is permissible. The following is the code segment for the `configuration` section of *dictator's entertainment* game A.7. Since this game has the same cell domain and rules except on the floor control protocol comparing to *generalised game* 5.4.1, both programming codes for these two protocols are the same except for the `protocol statement` in the `configuration` section.

```
configuration { ...
  // other declarations
  number of users 2;
  protocol master random turn 10 silence 5;// ◄ interaction protocol
} ...
```

This code segment shows the protocol *master server* with the default value is the server even though it is not stated in the statement. The timer options are the `turn timer` with the value of ten units of time and the `silence timer` is set to five units.

A more flexible option is `userdefined` for including a user-defined code segment in the server part of the program, providing JACIE programmers with the capability to incorporate an arbitrary server-based turn control mechanism into an application.

The other choice, the protocol *master client* allows one of the clients, η_{client}, to become the master, which can exercise the turn protocol in an application. Similar to *master server userdefined*, this protocol always requires the inclusion of a user-defined code segment for turn management, except that the code is defined in the client part of the program. This

protocol causes the *Protocol Manager* to be temporarily suspended and control passed to the user defined algorithm for interaction management.

The `userdefined` feature in the master protocol is supported by generic turn management statements, `turn pass`, `turn request`, and `turn set client`, as well as the `action start` and `action end` statements. In addition, protocol master can be modified by the turn timer, δ_{turn}, the overall timer, $\delta_{overall}$, the silence timer,$\delta_{silence}$, the action counter, σ_{action} and the rest timer,δ_{rest}. These options, which have the same meaning as defined in the round robin protocol, allow further parameterisation of a user-defined protocol. This in effect allows more creative JACIE programmers to design any complex interaction protocols. This flexibility is further extended with the user-definable option in the *group protocol* (Section 5.4.6.1) and the dynamic protocol feature (Section 5.6.1).

5.4.5 Tapping

```
protocol tapping
[turn [timer] δ_turn] [overall [timer] δ_overall]
[silence [timer] δ_silence] [[max] action σ_action]
[rest [timer] δ_rest]
```

In *tapping*, the order of the turn is determined by the client who currently gets the turn control. It is different from the protocol *master user* since in this protocol, every user is the master instead of just one master for all. The timer choices can also be made in this protocol.

To support protocol *tapping*, the `turn client` statement is used to enable the current user to set the next turn according to the user number. The user is free to decide whether to have another turn or to pass to the opponent.

```
JACIE {
   applet name  ncGBattle;
   configuration {
     ... // other declarations
     number of users 2;
     protocol tapping;   // ◄ interaction protocol
   }  ...
   client implementation {
      declaration { ... }
      ...
      on session { ...
        on MOUSECLICK { ...
           ... // process an action
           if (usernumber == 1)     // set the next turn to opponent
              turn set client 2;
           else
              turn set client 1;
           turn pass;
        }  ...
      } ...
   } ...
   ... // the remaining code is similar to generalised game, Section 5.4.1
}
```

In general, the program coding is also similar to the *generalised game* 5.4.1, but the slight differences are the interaction protocol setting in the `configuration` section and the inclusion of the `turn client` statement.

5.4.6 Group Protocols

```
protocol group [groupnumber grp]
[userdefined | random | roundrobin | master η_grp,mem]
[turn [timer] δ_turn] [overall [timer] δ_overall]
[silence [timer] δ_slience] [[max] action σ_action]
[rest [timer] δ_rest]
```

```
turn set group [groupnumber grp] η_grp,mem
```

In JACIE, besides handling an individual user's turn, the turn within a group is also considered because JACIE not only allows the collaboration among individual users, it also allows group collaboration. In the *group protocol*, the number of users K, is divided into a number of groups specified by the JACIE programmer in the JACIE `configuration` section. Let L groups, $grp_1, grp_2, \ldots, grp_L$, share a game. The JACIE group manager will select one member of a group to represent the current group in control, grp_l. While at the same time, the JACIE floor manager will select grp_l among the groups. All the groups can be assigned the same protocol or it is also possible to have different protocols for different groups.

The final statement `turn set group` permits selection of any group member to start a turn cycle. Otherwise, by default, the first player in a group will start the cycle. This is most significant for the *group round robin* protocol.

For the game rules and the domain cells, group protocol is similar to the *generalised game* 5.4.1 and the *gentlemen's battle* A.6. Therefore the implementation is the same as those except for the `protocol statement` in the `configuration` section. If the group number is not specified in the *protocol group* statement, all the groups will have the same within group protocol.

```
configuration {
  ... // other declarations
  number of users 4;
  protocol roundrobin;          // ◄ interaction protocol between groups
  number of groups 2;
  protocol group roundrobin;    // ◄ interaction protocol within each group
}
```

As an alternative, different groups could have different interaction protocols within the group as illustrated in the following example.

```
configuration {
  ... // other declarations
  number of users 4;
  protocol roundrobin;        // ◄ interaction protocol between groups
```

```
    number of groups 2;
    protocol group groupnumber 1 random;     // ◄ interaction protocol within each group
    protocol group groupnumber 2 master 1;  // ◄ interaction protocol within each group
    // ▲ group 1 uses the random order within the group,
    // group 2 uses master within the group, with player 1 acting as the master
}
```

This example uses the interaction protocol for all the groups as *round robin* but each individual group has its own in-group interaction protocol.

5.4.6.1 Protocol Group Userdefined

In JACIE, there is a set of built-in channels that include: canvas channel, message channel, chat channel and video channel where more than one channel can be used at the same time. For group collaboration, usually group members will communicate using these private channels. Since all the group members will have the turn control, they can decide who in the group will represent the group through the communication using one or more of the above mentioned channels. All members, p_k of the group grp_l will have $G_k(t)$ made active. When one of the members of grp_l has updated the board and passed the turn, then the turn of next group will be activated.

5.4.6.2 Protocol Group Roundrobin

The concept of the round robin for in-group protocol is very similar to the user turn protocol `roundrobin`. Each member of a group has an equal opportunity to represent the group in turn. At the grp_l's turn, only one member of the group will have the active game board to work on, the other members' boards will be inactive. When finished and the turn control is passed, the server will determine the next group.

5.4.6.3 Protocol Group Random

The selection of one member, who represents the group, is performed at random by the server. The chosen member will have the turn, and the rest of the members are inactive. After the turn is passed, the next group is selected.

5.4.6.4 Protocol Group Master $\eta_{grp,mem}$

For the protocol `group master`, only one member represents the group for every turn. The master of the group, $\eta_{grp,mem}$, acts like a leader of the group. The group leader can be set by the server or the server may ask the group to elect the leader. With the `turn set` statement, JACIE allows the flexibility to change the master by the server.

5.5 Language Enhancements

This section describes the improvements that have been made to JACIE I on managing interaction. We describe the changes to the lexicon and grammar required of the extended language, JACIE II. As JACIE II permits many more options and tags for the protocols, the corresponding grammar rules are necessarily more complex. For example, there are several options on timer selections and in a protocol type itself, which can consist of more than one choice such as *contention* or *contention hold* and *master server* or *master client.*

5.5.1 The Software Architecture for Managing Collaboration

As discussed in Section 5.3.2, an interaction protocol is essentially a discrete temporal function ψ, which is usually realised by a set of sub-functions operating in both the server and clients. In JACIE II, this is implemented as a set of software modules as shown in Figure 5.5. The rules for co-ordinating interactions largely reside in the *floor manager* for general inter-client turn control, and in the *group manager* for handling inter-group turn-control.

As communications between the server and clients are conducted over the *JACIE message channel*, several other JACIE modules are also involved in interaction management, including *Session Manager* for initialisation and message handling, and *Global Data Manager* for storing and managing all global variables and message identifiers including those used by interaction protocols.

Normally one would expect a complex set of pre-defined library functions or objects for managing collaborative activities and for interfacing with communication sub-functions. In JACIE, however, for an ordinary programmer, the programming interface to these predefined sub-functions is largely declarative, that is, in the form of protocol specifications.

Figure 5.5 illustrates that the main communication method between client and server is through the *JACIE message channel.* Each client has two main components, *Local Data Manager* and *Client Session Manager.* At the server, the predefined code segments have three major components, namely *Server Session Manager*, *Global Data Manager* and *Protocol Manager.* Both the server and client predefined code segments are actually the JACIE translated compiler codes that process the user-defined JACIE program, both the client and the server, at their upper level of the architecture.

All the clients will have the same copy of the client program. Since every client runs the program on different machines, all the data are kept locally and the same copy of the program code runs on each separate machine. The *Local Data Manager* is responsible for all the client's local data. Any shared data, which is also known as *global* data, can also be kept at client sites provided that the client is the owner of the data. Shared data must always reside on the server and the server has control of the shared data as well as the owner(s). Since every client has the same copy of the program, this does not necessarily mean that every client must have the same copy of shared data. They only have the same copy of *local* data. The detailed description of this topic is discussed in Chapter 6. The *Client Session*

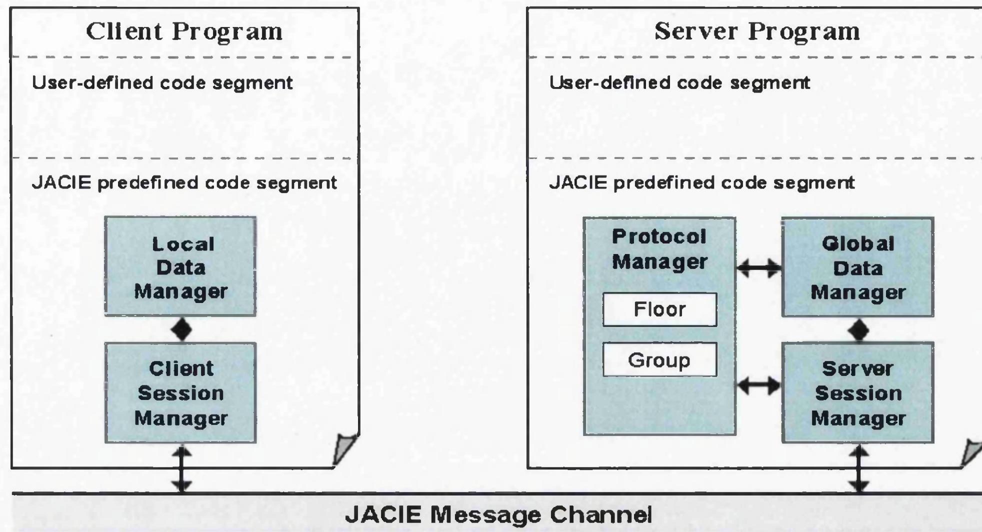


Figure 5.5: JACIE Collaborative Management.

Manager is a component that handles all the communication between client and server, at the client. It recognises the messages sent and received, to and from the server.

5.5.2 Additional Tokens and Productions

In adding the new interaction protocol constructs, new tokens are introduced. Below is an example of the code segments in `jacie.flex` file. It shows some of the tokens listed in the left column, and the specified name to be recognised for the next parsing process. All the new names for the next process must be written in the capital letters.

```
...
"hold"                     { return symbol(sym.HOLD); }
"master"                   { return symbol(sym.MASTER); }
"request"                  { return symbol(sym.REQUEST); }
"set"                      { return symbol(sym.SET); }
"rest"                     { return symbol(sym.REST); }
"action"                   { return symbol(sym.ACTION); }
"silence"                  { return symbol(sym.SILENCE); }
"overall"                  { return symbol(sym.OVERALL); }
...
```

The next step is to add the same tokens to the `jacie.cup` file to be recognised by the compiler tool JCup. Besides these terminal symbols, JCup requires the declaration of new non-terminal symbols to be used in the productions in the grammar. The non-terminals can be of any type such as *string*, *integer*, and many more, including Java *classes* defined for the code translation phase. The following code segment is the declaration for the new non-terminals in handling the interaction management. In the definition of the terminal symbols, all tokens are written in capital block letters as the JFlex compiler tools produce all the terminals in the same form.

```
...          // List of terminal symbols
terminal MASTER;
terminal REQUEST;
terminal SET;
...          // List of non-terminal symbols
non terminal protocol_user;  // ◄ non-terminal of no specific type as it produces other non-terminals
non terminal master_choice;
non terminal protocol_choice;
non terminal timer_choice;
non terminal timer_select;
non terminal contention_timer;
non terminal group_protocol_choice;
non terminal JACIECStatement start_action_statement; // ◄ type java class JACIECStatement
non terminal JACIECStatement end_action_statement;
non terminal JACIECStatement turn_client_statement;
non terminal JACIECStatement turn_request_statement;
...
```

The following example includes productions where some of the above non-terminals are used, for the `configuration` section and the initialisation section of the JACIE program. The `specify_protocol` production shows that the `protocol_choice` can be of a group or user protocol. For this production, the parser makes a call to a method, `protocolCheck()`, in `JACIEConfig class`. Both of the defined protocols have the same `timer_choice`, and they are the initial turn protocols to start a session.

```
specify_protocol::=  PROTOCOL protocol_choice timer_choice SEMICOLON
                     {: JACIECConfig.protocolCheck();  :}
                     ;

protocol_choice ::=  GROUP specify_group_protocol
                     |
                     protocol_user
                     ;
```

The following code segment is the user protocol choice. For every protocol type, the compiler sets the initial system configuration according to the specified protocol. For `contention` and `master` protocols, there exist several other options within the protocols. The option is denoted by an extra non-terminal next to the protocol name. These options result in several other protocol types that make this language flexible and comprehensive for implementing collaborative applications.

```
protocol_user::= CONTENTION contention_timer
                 {: JACIECConfig.protocol = JACIECFloorMgmtProtocol.contention; :}
                 |
                 ROUNDROBIN
                 {: JACIECConfig.protocol = JACIECFloorMgmtProtocol.roundrobin; :}
                 |
                 MASTER master_choice
                 {: JACIECConfig.protocol = JACIECFloorMgmtProtocol.master; :}
                 |
                 RESERVATION
                 {: JACIECConfig.protocol = JACIECFloorMgmtProtocol.reservation; :}
                 |
                 TAPPING
```

```
                {: JACIECConfig.protocol = JACIECFloorMgmtProtocol.tapping; :}
                ;
```

The group protocol that handles the user's turn within a group also has several options. The production, as shown below, sets the initial compiler configuration for handling the groups.

```
specify_group_protocol::=  group_protocol_choice SEMICOLON
                           ;

group_protocol_choice
         ::=  USERDEFINED
              {: JACIECConfig.groupProtocol = JACIECGroupProtocol.userdefined; :}
              |
              RANDOM
              {: JACIECConfig.groupProtocol = JACIECGroupProtocol.random; :}
              |
              ROUNDROBIN
              {: JACIECConfig.groupProtocol = JACIECGroupProtocol.rrobin; :}
              |
              MASTER
              {: JACIECConfig.groupProtocol = JACIECGroupProtocol.master; :}
              ;
```

As some turn protocols require several supporting statements, some new productions in the JACIE parser are added. The following code segment illustrates those statements. Since all of them involved new statements, the JACIE II parser has to invoke new Java classes to produce those new statements. In these example productions, the compiler tool, JCup, is not only checking the syntax of the statements, but also checking the semantics and producing the equivalent Java statements by invoking the associated new Java classes, such as `StartActionStatement` and `EndActionStatement`.

```
start_action_statement::=  START ACTION SEMICOLON
                           {: RESULT = new StartActionStatement(); :}
                           ;

end_action_statement  ::=  END ACTION SEMICOLON
                           {: RESULT = new EndActionStatement(); :}
                           ;

turn_pass_statement   ::=  TURN PASS user_option:uopt SEMICOLON
                           {: RESULT = new TurnPassStatement(uopt); :}
                           ;

turn_client_statement ::=  TURN CLIENT INTEGER_LITERAL:num SEMICOLON
                           {: RESULT = new TurnClientStatement(num.intValue()); :}
                           ;

turn_request_statement::=  TURN REQUEST BOOLEAN_LITERAL:num SEMICOLON
                           {: RESULT = new TurnRequestStatement
                                         (boolliteral.toString()); :}
                           ;
```

All the examples in this subsection show some of the new terminals and non-terminals introduced with the necessary productions added to the existing grammar. In addition, many new Java *classes* are also introduced that require more Java methods that are added to the

`JACIE code translator` program.

5.5.3 Additional Codes and New Java Classes

There are some new Java classes created to support the compiler for code translation. Table 5.2 lists all the new Java classes for the parser and the code translator.

CLASS NAME	**PURPOSE** (in code translation process)
RemindTurn	To stop the turn timer event
RemindSilence	To stop the silence timer event
RemindOverall	To stop the overall timer event
StartActionStatement	Resets silence timer
EndActionStatement	Count number of actions
TurnClientStatement	Allows to set client turn
TurnRequestStatement	Allows setting a request to a Boolean value
ChangeProtocolStatement	To change protocol during session

Table 5.2: New Java Classes for Interaction Management.

Major code modification and extensions can be found in the *Protocol Manager*. The *Protocol Manager* contains two files `JACIECFloorManagerTemplate.java` for handling user turn protocols and `JACIECGroupManagerTemplate.java` for the group protocols. Table 5.3 shows the changes on the user turn protocols.

VERSION	**CONTENT**
Original	1. Declaration of variables 2. Set initial values of all variables 3. Algorithm for protocol *round robin* and *contention*
New version	1. Declaration of variables 2. Set initial queue for protocol *reservation* or *contention hold* 3. Write a method containing next turn algorithm for all protocols 4. Write a method containing initial set values for all protocols 5. Write a method for queue manipulation on protocol *reservation* or *contention hold*

Table 5.3: Changes on FloorManagerTemplate.java File.

In JACIE I, only protocol *round robin* and *contention* were actually implemented while the rest of the user protocols, such as *reservation* and *tapping*, were defined in the previous design [140], and included in the file for initial program configuration.

In JACIE II, algorithms are implemented to handle all the protocols mentioned in the previous section 5.4. The *reservation* and *contention hold* protocol require the use of queues. In reservation, the queue holds the user number which made the request for turn control during the `TURN REQUEST mode`, while in *contention hold*, the function of a queue is simply to keep the order of the users' turns.

For the group protocol, there are major changes to the *Group Manager*, called the `JACIECGroupManagerTemplate.java`, the file where the codes reside. In JACIE I, the only protocol for groups is to select the group members, and there are no turn protocols for group interaction. The selection on the group members is set according to the arrival of the users. The first user is assigned to the first group and the next user is assigned to the next group following a cyclic order until all the users have been placed in the existing groups. Table 5.4 lists the changes made for the *Group Manager*. There are several additional methods in the JACIE II since there are four new choices of group protocols implemented.

VERSION	CONTENT
Original	1. Declaration of variables 2. Set group members according to the order of user arrival 3. Write a method that returns the member of the specified group
New version	1. Declaration of variables 2. Set group members according to the order of user arrival 3. Write a method that returns the member of the specified group 4. Write a method to set the same protocol for all groups 5. Write a method to set a protocol for a specified group 6. Write a method to set protocol initialisation values 7. Write a method to handle group roundrobin protocol 8. Write a method to handle group master protocol 9. Write a method to handle group random protocol

Table 5.4: Changes on JACIECGroupManagerTemplate.java File.

Since interaction management is also supported by other JACIE software components, such as *Global Data Manager*, *Local Data Manager*, *Server Session Manager* and *Client Session Manager*, there are also modifications and enhancements in the files associated with these components. The changes and the additions include the implementation of timer options and new supporting statements that are required in some of the protocol choices as described in Section 5.4.

5.6 Other Protocol Design Issues

In implementing the interaction protocol design into JACIE, several other issues arise. This section discusses those issues that include defining protocols dynamically which can be changed in an application and consideration of timer implementation either server based or client based.

5.6.1 Static and Dynamic Interaction Protocol Settings

With the above protocol declaration constructs, JACIE allows a programmer to set any of the protocols in two different sections in a JACIE program, the `configuration` section

and `onSession` section. Since the `configuration` section is at the beginning of a user program and known as the declaration part for both client and server, the setting protocol in this section is referred to as a *static* declaration. Once it is declared, the protocol throughout the session will remain unchanged if no other protocol statement is issued during the `onSession`. Contention is the default protocol if none is declared.

For flexibility, the JACIE programmer can change the interaction protocol dynamically by issuing the appropriate protocol declaration statements in the `onSession` section. This is called the *dynamic* feature, allowing changes in the protocol at anytime during a session. The protocol declarations for both static and dynamic settings shared the same syntax. However, this option is only applicable in the server program. If it happens that the programmer tries to include the statement in the client program, the attempt will be ignored by the JACIE compiler.

With the choice to use either static or dynamic protocol declarations, the JACIE II programmer has more flexibility in implementing various types of applications. For example, in implementing a card game, such as Bridge, at least two different protocols may be needed, which will be further discussed in Chapter 7.

5.6.2 Timer Implementation

There are four timer options: `overall`, `turn`, `silence` and `rest`. The default values are, ∞, ∞, 3 units and 0, respectively. The objective of `overall`, `turn` and `silence` options are to prevent users from holding their turns too long resulting in long waiting time for others. The rest timer is to reduce the over cluttering of the message queue at the server since a lot of actions may occur consecutively which result in sending a lot messages to the server.

In implementing the timers of JACIE II, a timer is a Java *Timer class* that creates an event for the compiler and executed as a thread. The following subsections discuss the alternative approaches for having either a server based or a client based timer.

5.6.2.1 Server Based Timer

By having the timer resides on the server, provides the opportunity to have centralised control. The timer can be considered a 'global clock'. For each client's turn, the server has to send messages at the start and at the end of the timer. This must be done for every timer option indicated in the interaction protocol statement. However, the possible delay in transmitting messages may result in inaccuracy in the actual time given to the clients. It is very unlikely that the time for the start timer message and the end timer message to reach the client, take exactly the same length. Many factors can influence this, such as the current network traffic that may be very unpredictable.

Furthermore, the server based timer requires the information at the server that can result

in a lot of message passing activities between server and the current client in control, especially when more than one timer option is chosen. As mentioned earlier in Chapter 4 that JACIE message is in the form of header and content, which message header is represented by an integer value (Figure 4.6). Therefore, to determine the start and end of a timer, messages must be sent by the server to the client who has the turn.

Below are examples of Java code generated by the JACIE compiler for handling the message identifiers. The message identifiers are integer constants and they are part of the declaration section in the JACIE *server session manager*. This code segment shows the message identifiers for start and end of all the timer options.

```
...
public static final int TURNTIMERSTART   = 21;     // the start of turn timer
public static final int TURNTIMERSTOP    = 22;     // the end of turn timer
public static final int OVRLTIMERSTART   = 23;     // the start of overall timer
public static final int OVRLTIMERSTOP    = 24;     // the end of overall timer
public static final int SLNCTIMERSTART   = 25;     // the start of silence timer
public static final int SLNCTIMERSTOP    = 26;     // the end of silence timer
public static final int RESTTIMERSTART   = 27;     // the start of rest timer
public static final int RESTTIMERSTOP    = 28;     // the end of rest timer
...
```

The following is an example of JACIE generated Java code for handling the `overall` timer. The code shows a Java *Timer class*, `ovrlTimer` is declared and activated when the call to the method `public void startOvrlTimer()` is made. When the timer duration is reached, an automatic call to another method, `public void run()` in `class RemindTimeTask extends TimerTask`, is made to stop the timer event. In both methods, `public void startOvrlTimer()` and `public void run()`, there are calls to the *server session* to activate message exchanged represented by two identifiers, `OVRLTIMERSTART` and `OVRLTIMERSTOP`.

```
class RemindTimeTask extends TimerTask {
  public void run() {
     System.out.println("Overall time is up!");
     myTurn = false;
     onSession();
     sessionAssistant.send            // a call to a server session component
       (ncsftimerServerSessionManager.OVRLTIMERSTOP+":");
        ... // ▲ message sent to client when timer stop
     ovrlTimer.cancel();
     onSessionEnd();
    }
 }

 public void startOvrlTimer() {
    ovrlTimer = new Timer();
    sessionAssistant.send            // a call to a server session component
      (ncsftimerServerSessionManager.OVRLTIMERSTART+":");
       ... // ▲ message sent to client when timer starts
    ovrlTimer.schedule(new RemindTimeTask(),90*1000);
    System.out.println("Start overall timer");
 }
```

5.6.2.2 Client Based Timer

The client based timer is an alternative approach where all the timer options are activated and run on the client's site. The server determines the turn control and once the client gets its turn, all the appropriate timers are started. The handling of all the timer events is entirely on the clients. Therefore, the client needs to inform the server of the end of each turn control. It is appropriate to have the timer running on the client's site rather than the server, especially for the `turn` and the `silence` timers. These two timers are activated once the client receives the turn. This will give a more accurate timing for the client program to run since the status of the network traffic will not have any influence.

In terms of storage, every client shares the burden and there will be little message passing between the server and clients to handle such timers. However, when the timer is a Java class event, many threads may exist. One problem which may occur is the existence of a timer interrupt to indicate the end of the user turn in the middle of sending other important messages from the client to the server. This is to be discussed in the next section.

5.6.2.3 Timer Interrupt

When there is a timer interrupt in the middle of sending a message, errors may occur at the server site which is waiting for a specific sequence of messages to be received. In this case, JACIE II introduces two new statements, `criticalsection start` and `criticalsection end`. The `criticalsection start` allows messages to be sent continuously to the server even though the time is up. When this statement is executed, the timer interrupt event in the JACIE II compiler, is ignored which enables messages in the client message queue to be retrieved and sent to the server until another statement, `criticalsection end` is executed.

The following JACIE code segment shows how these two statements, `criticalsection start` and `criticalsection end` are placed before and after several `send` statements, respectively.

```
...
on MOUSECLICK {
     if (myTurn) {
       gX = GETGRIDX;
       gY = GETGRIDY;
       if (GETGRID == board)
         if (boardP[gX][gY] == ""){
           criticalsection start;  // ◄ indicates the start of ignoring any timer interrupt
           send gridX gX;
           send gridY gY;
           send userNum usern;
           send symMark playerMark;
           criticalsection end;    // ◄ indicates the end of sending messages
           turn pass;
         }
       }
...
```

The effect of the statements, are for some flags in the Java generated code, from the JACIE compiler, to be set to ignore the interrupt from the timer event and also to reset the event to its default value. The following shows part of the Java equivalent codes to the above JACIE code segment.

```
...
if (boardP[gX][gY].equalsIgnoreCase(""))
  {
     criticalSection = true;       // ◄ actual flag setting to ignore any interrupt
     sessionAssistant.send(GRIDX+":"+gX);
     sessionAssistant.send(GRIDY+":"+gY);
     sessionAssistant.send(USERNUM+":"+usern);
     sessionAssistant.send(SYMMARK+":"+playerMark);
     sessionAssistant.send(sessionAssistant.PASSTURN+":");
     criticalSection = false;      // ◄ actual flag setting to be the default value of the event
     myTurn = false;
     turnTimer.cancel();
     setTimer = false;
  }
...
```

At the end of each timer event, some variables that act as flags are set to FALSE to indicate the end of certain actions. The following code segment illustrates flags such as `myTurn`, and `setTimer` being set to FALSE when the turn timer stops. The `criticalSection` that controls the timer interrupt is checked upon the occurrence of the interrupt. If there are messages to be sent to the server at this time, the `criticalSection` will have the value TRUE, resulting in the timer interrupt being ignored. The appropriate messages will be sent to the server in the other event section where all these relevant flags values are set.

```
...
class RemindTask extends TimerTask {
  public void run() {
    if (!criticalSection) { // ◄ check flag at the timer end event,
      myTurn = false;        // ◄ to change user turn
      sessionAssistant.send(sessionAssistant.PASSTURN+":");
      turnTimer.cancel();
      setTimer = false;
    }
  }
}
...
```

Figure 5.4 shows the screenshot of the *generalised game*, the client-based timer is used in its implementation. It is possible to display the remaining time of the turn in the JACIE `Local Message` section at the bottom of the screen. This may not be done with precise timer values if the timer is server-based.

Table 5.5 shows the summary of the comparison on the server and client-based timers. The comparison is made based on accuracy of the timer, the efficiency of the management, the disadvantage and advantage of the two options as mentioned above.

It is better to implement the timer options using the client-based approach as it gives accurate timing. Although it is totally managed by clients, the distributed management gives the

	Server-based	Client-based
Timing Accuracy	Depends on network traffic	Accurate
Management Control	Easy to manage (centralised)	Each client has control on the local timer
Disadvantage	Many message passing activities	Timer interrupt
Advantage	No separation between timer and turn control	local display of current timer values

Table 5.5: Timer Based Comparison.

clients full control in manipulating and displaying the timers. For example, in the case where the timer interrupt may occur during the execution of any code, an algorithm can be written to determine the interrupt and proper action can be taken. As the server-based timers have to rely on the network for sending and receiving information about the timers, it may not give the client the exact period of time specified. It is easy to manage the centralised timer setting especially if it is the `overall` timer where only a simple algorithm is needed to perform the calculations. There is also less chance of having interrupt problem because the server always has full control of the overall system and most activities occur at clients. Hence, additional codes, algorithms and message passing activities are needed at the server.

5.7 Summary

In this chapter, we have based our technical discussions around the noughts and crosses game and its variations, which serve as an 'abstract' collection of networked collaborative applications, and enable us to focus on the interaction management, rather than the context-specific details of the applications. We have presented a set of formal notations for modelling the spatio-temporal activities in a noughts and crosses game. The formal notations help address the main issues in interaction protocols, including the visual states and modes of the game boards, temporal consistency between clients and the server, and discrete events series. We have highlighted that an interaction protocol for managing collaborative activities is essentially a discrete temporal function, which in most cases, will have to be realised by sub-functions in both the server and clients.

Based on the formal notations and consideration of noughts and crosses games, we have developed a comprehensive collection of interaction protocols, and have incorporated them into JACIE. These protocols, including, *round-robin, contention, master, reservation, tapping* and *group*, are capable of addressing the protocol needs in all variations of the noughts and crosses game described, and thereby the related applications. Our main contribution in this respect is the adventurous attempt in providing language constructs for specifying a variety of interaction protocols. The implementation of such functionality otherwise typically requires the skills of experienced network programmers.

The implementation of the interaction protocols in the language has resulted a major ex-

tension to JACIE I. Many code modifications and additions are made. Contents of several existing files have been changed and some new files are introduced. This chapter also discusses other protocol design issues that include static and dynamic protocol declarations, and the implementation of timer options.

Chapter 6

Interest Management

Contents

6.1 Introduction 123
6.2 Related Work 124
6.3 Interest Management in JACIE 127
6.4 Language Constructs for Interest Management 138
6.5 Language Enhancements 142
6.6 Technical Considerations 145
6.7 Secret Switch 147
6.8 Summary 153

6.1 Introduction

Almost all collaborative applications involve the management of data distribution and filtering according to the needs of the receivers. *Interest management* is concerned with relevance-based data filtering in distributed and collaborative environments. The main objective is to avoid broadcasting data unless it is to be shared by all the processes, and to provide secured data transmission of a subset of information relevant to each process. In recent years, the issue of interest management has largely been considered in the context of large scale collaborative virtual environments [124]. In many ways, it is a long-standing issue that was previously considered in areas such as parallel and distributed computation, distributed operating systems and distributed database systems. However, the resurfacing of this issue clearly indicates the lack of generic support for interest management in programming languages and software development tools. In particular, it is not trivial to program dynamic data distribution and message communication with changing access control governed by interactive interaction needs.

There are numerous programming languages for developing Internet applications in general. These include general purpose conventional languages such as Java and C#, and scripting languages such as Perl, Python, VBScript and JavaScript. In addition, there are also

many domain-specific languages such as Distributed Oz [272] for network transparency, Yoix [102] for handling broadcast messaging, threaded communications, logging, and screen management, JCell [267] for distributed object and mobile code, and JACIE I [139] for prototyping collaborative environments. None of these languages yet feature any high-level constructs for interest management.

As pointed out by [95], commercial software systems, such as Lotus Notes, Novell Groupwise, Microsoft NetMeeting, O'Reilly Webboard and ICQ, largely employ store-and-forward transactions via a centralised server for facilitating data sharing among remote users. This raises the question as to how the decision on accessibility, security, mutual exclusion and data filtering can be incorporated into a collaborative application, ideally not to involve direct programming of a server. It is highly desirable, in most circumstances, to have high-level language constructs allowing the specification of interest management at the client without the need of programming a server.

Programming shared data is an indispensable task in the development of many collaborative applications. It is commonly implemented through a centralised database, where shared data is dynamically filtered according to access needs, restrictions and rules. While such an approach is technically effective, it relies upon a sophisticated database system, which supports reliable and protected concurrent access, in engineering such software environments. However, this approach that involves an extensive effort is usually implemented for large scale systems [159], such as collaborative virtual systems [205, 332]. Therefore, it is desirable for JACIE, as a development environment for collaborative applications, to provide program constructs that supports some basic interest management facilities without relying on a sophisticated database system.

6.2 Related Work

Since we define interest management to cover data sharing, filtering and access right issues, this section investigates several different implementations in other research.

6.2.1 Programming Data Sharing in Distributed Systems

In distributed systems, sharing of interest can be defined as the sharing of resources available on the network. Data may be the most popular resource that people share. It can be in any form of multimedia elements, object, component or variable. Other resources such as devices, memory and information can be also shared over the Internet.

From the perspective of programming languages, managing shared data becomes essentially the programming of 'variables'. However, conventional programming languages do not normally provide program constructs that support such *remotely shared variables*, except via network programming APIs (e.g., RPC, sockets, MPI). Note that so called *global*, *public* and *external* variables are mostly confined to a single process programming paradigm. Attempts were made to introduce high level constructs in some languages, including APL [184],

Scheme [195] and Orca [22]. These languages are mainly for sharing of memory and their communication facilities are supported by operating systems. Thus, the shared data is declared in these languages while the rest of the operations are performed by making calls to the specified operating systems. For example, Orca defines shared data as an object. The sharing of this object is performed through a `read` operation when this object is applied to a local copy, and a `write` operation ensures all copies are updated immediately using a reliable broadcast protocol. The syntax for declaring the shared data is shown below where the word `shared` is used to indicate its status.

```
identifier:  shared <object type>
```

The rest of the operations are performed by the use of a queue for message exchange [19] since the communication is supported by message passing activities.

Distributing shared memory is also an important approach to the distributed data management. Since a variable refers to a location in a memory, the data management mechanisms may be similar. However, the emphasis is more on the memory rather than the variable itself. Programming languages that deal with shared memory are often referred to as coordination languages. Examples are Linda [291], XMLSpace [316] and JavaSpaces [61] which provide shared space, called tuple space, for storing global variables. The security of the data essentially relies on the supported application and usually uses a database system.

6.2.2 Interest Management and Filtering Methods

Interest management in collaborative environments is often coupled with the replication and communication of data to be shared by users. For example, in very large virtual environments, it is necessary to manage *interest* by filtering out data that is of no interest to a particular user [124]. MASSIVE-3 [135] and ATLAS [191] utilise distributed databases to manage interest, SPLINE [326] facilitates interest management at the level of locales, while ATLAS supports interest management based on user interests and spatial distance.

As managing interest in collaborative environments is mainly concerned with data filtering, restrictions on whether or not shared data is accessible must comply with the level of interest of users and the level of access of owners. As many CVE systems like ATLAS and MASSIVE use aura and nimbus [33] to determine the areas of interest, there are many methods to determine the level of access rights as found in grid computing, e-commerce, database systems and wireless communications. Levels of access in these systems can depend on the Virtual Organisation [259], the present state and history of the user's behaviour [331], current event [38], knowledge hierarchy [32], location [198], user driven [328] or based on a user hierarchy [62]. However, none of these levels of access can be specified by high-level language constructs.

The networked systems as mentioned in Section 2.4.1 in Chapter 2, provide us with several different techniques of filtering. Largely, interest management is carried out by simulation techniques to test the system design effectiveness [224, 230, 205, 312, 323]. Here, we merely focus our attention to their implementation techniques in determining user interest.

Table 6.1 summarises a number of distributed system environments with the system objectives, the sharing method, the type of the data filtering and examples of real life applications using such systems. The filtering method, `distance`, is seen to be the most common technique that is dependent on the virtual distance between the sender and the receiver, which effects the network delay. While most systems propose several filtering techniques based on the user interest to reduce the amount of data transmitted from sender to receiver, in contrast, the main objective of parallel system environments is to reduce the amount of data to achieve better system performance and the data filtering has no concern with the user interest.

Environment	Objective	Shared method	Filtering method	Example applications
Network and CVE	Minimise network traffic	message [135, 124, 191]	distance [135, 124, 191]	chatting, CVE navigations, games, document download
Agent System	inter-operability	object [323, 263], information [298]	distance [323], activities [298], probability [263]	file system manipulation, resource determination, on-line auction
Parallel Systems	High performance, computation speed up	memory [180]	data layout [122], data placement [180], algorithm [180]	statistical analysis handling multimedia, arithmetic computation, massive databases
Distributed Database	Information sharing	information [53]	functional activities [53], distance [217]	e-billing, e-banking, reservation systems, question-answer UI
Network Language	Implement various and flexible applications	variable [22], object [13, 91], component [13, 91]	Interaction frequency [91], distance [13], preset [13]	e-learning, video conferencing, games, web camera-control

Table 6.1: Environment and Applications for Distributed Systems.

We can conclude that in any system that involves data sharing, there is a filtering method to ensure a structured and easy management for consistency and system efficiency. Even though, different environments have different objectives and sharing methods, the support of the data filtering helps in the design of such systems.

6.2.3 Access Control and Data Security

Chapter 2 and 3 discuss the issues of access control and security in existing collaborative systems and their implementation techniques using programming tools or languages. Here, the discussion based around the general approach that can usually be found in an operating system.

In operating systems, it is common that a shared resource is represented by a file. A file

is usually referred to by a name, structured in several ways (e.g. in bytes, records, *etc.*) and can have several data types such as text, images, *etc.* In managing file access, an operating system includes other information associated with a file called *file attributes* [223]. Several possible file attributes include a file creator with an identifier, an owner, read-only flag, lock flag, current size and maximum size. Different systems may require different file attributes. These attributes help the operating system in determining any access to such files. The operating system often has a list of users with its corresponding resources and operations such as read, write and execute [310].

The operating system also protects each individual user by a password scheme so that private resources cannot be shared by others. It is dependent on the user to set the password's value. A password is usually encrypted and compared to the previously stored password given to the operating system [310]. Nowadays, it is also possible to enhance security management using additional device such as *Personal Security Proxy* (PSP) [337] or a smart card [223] with encryption and decryption mechanisms.

6.3 Interest Management in JACIE

JACIE II allows for the sharing of resource through 'shared variables' with the level of access being specified by security rules. Users can set their level of interests in their accessible shared variables. Therefore, upon having to access some variables, it is the users' options to declare their level of interest so that the server can determine the filtering of broadcast communication messages and information. Information on the shared variable must always be kept secret from unauthorised and uninterested users to ensure efficient and secure management.

Although JACIE provides interest management for a collaboration of a small number of users, it is significant to reduce the number of message passing activities. Instead of sending 'extra' messages to some users who are not interested in using the data in the messages, users are guaranteed to receive the data that are relevant and significant to them. On top of this factor, with this interest management feature, a JACIE programmer can have a simple way to manage shared data without having to program details of message passing activities such as the use of `send` and `receive` statements.

Recall in JACIE I [139] that both server and client programs are defined in the same program, this feature can provide a consistent and coherent platform for introducing and controlling shared variables for interest management. The following subsections describe the management of shared variables in JACIE II that represents interest management.

6.3.1 Shared Variables and Attributes

In JACIE, the declaration of any variable to be read or written in a shared manner must start with the keyword `shared`. Hence, variables and shared variable can be declared using the following JACIE construct.

```
declaration { [shared] <data types>
      <variable declarator> }
```

Below is a JACIE code segment for declaring two variables of type *String*, namely `symbol` and `boardP`. The code shows that both variables are declared in the JACIE client program. The variable `symbol` is declared and managed locally as an independent variable by each client. On the other hand, `boardP` is a global variable shared among all clients, but managed by the server.

```
client implementation {
   declaration { ...
    string[2] symbol="X","O";
    shared string[8][8] boardP = "";
   } ...
}
```

For the local variable, the management is performed in a similar manner to variable manipulation in any programming language where an operation such as `read` or `write` can be performed anytime according to the scope of a variable in a program. In contrast, global variables require special management schemes to ensure that their values are always correct at any given time. This consistency issue is very important since it is shared remotely and may also be distributed in several places.

For each global variable, there are attributes attached to it. These attributes contain valuable information to ensure proper management when the global variable is being shared. Such information includes the variable's type and a list of users who have the permission to use it and these users are allowed to alter the data.

6.3.2 Management Framework

There are several tables used by the server and clients for managing shared variables. Initially, during the compilation process, the JACIE compiler creates several tables to keep useful information that consist of the following:

- All declared shared variables — All the shared variables are recorded in a table since their management is different from ordinary variables. When a client declares a shared variable, the variable is kept in a table called `Method Table`. Then, the compiler creates a similar type of table for the server program so that the declaration will appear in the server program instead of at client, because by default, the server is the owner.
- All shared variables that involve permissions — Since a programmer can include any variable in a permission statement, it is the compiler's job to check that each variable defined is in the `Method Table`. Any invalid variable usage will results in an error in the compilation stage.

Upon completion of the compilation process, several tables are created in the target program (which is referred to a user program). Figure 6.1 shows a diagram of the connection between compiler and user program components. The content of `Method Table` that exists during

compilation is copied to both the server and client programs using the same name. In the target program, the `Method Table` is used for searching the variable's index in determining the correct reference.

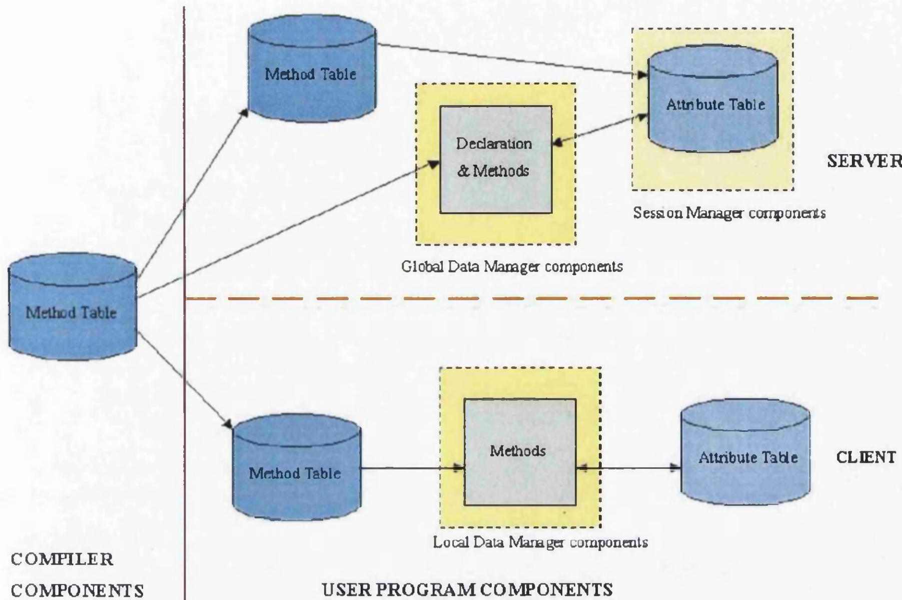


Figure 6.1: Tables for Managing Shared Variable.

In the user program, another table called the `Attribute Table`, keeps shared variable attributes for access validation that include user `access lists` and password checking if required. At the server, this table is handled by a *Server Session Manager*, while at the client the table is handled by the *Local Data Manager*. In general, the user program that contains both server and client programs, each has two types of table for managing useful information during runtime.

The *Global Data Manager* maintains a master copy of each shared variable for efficient and consistent data handling. A copy of each shared variable is also stored by the specific client owner. Each time the value is updated at the server, the updated copy will be sent to the owner, as well as at the clients who have the access. Since the shared variable's master copy resides and is automatically declared at the server, it is possible for the server to perform any variable manipulation 'locally' without informing the clients. Therefore, it is crucial that any operation on shared variables is performed through the JACIE's language constructs. Detailed descriptions of these language constructs are given in Section 6.4.

At the server, variable attributes are stored in a table by the *Server Session Manager* for the determination of access permissions and interest filtering. When a session starts, this table is empty. Once the server receives a permission instruction to use a shared variable, such an instruction is analysed and appropriate information is added to the table. This instruction is a permission statement that allows control to be transfered from a JACIE high level program to the *Server Session Manager* to assign variable attributes in the table shown

in Table 6.2.

ATTRIBUTE	PURPOSE
Variable name	As a reference of the shared variable
Owner's read password	To keep password for read validation
Owner's write password	To keep password for write validation
Variable type	To determine the data type
Variable presentation	To determine primitive or array type
Permission list	The original set of the permissible users
Access list	The actual set of users who get the access
User interest value	To keep user interests' values
Interest filter	To keep the owner interest filter value
Access list count	The total number of users in the access list
Permission list count	The total number of users in the permission list
Owner flag	To determine 'own' permission of a variable
Read access flag	To determine 'read' permission of a variable
Write access flag	To determine 'write' permission of a variable

Table 6.2: Shared Variable Attributes on the Server.

The permission list attribute keeps the original list of users who are given access permissions and is included in a list called the `user list`. This list is needed, since not all the users in the `user list` are automatically given the access permissions. Other factors, such as the user's interest and the password, if required, influence the granted access. Therefore, another list, the `access list`, is required to store the list of the users who have been granted access after going through the validation and interest filtering.

For every user in the permission list, the variable attribute table also stores each user's interest value. This is needed for interest filtering comparison. Users are free to change their interest values at anytime and the owners are also free to change the value of the interest filter. Hence, the number of users in the permission lists can always be changed. This results in the need to have a counter to cope with the changes. These changes are not only concern the content of the permission list, but also the content of the access list.

The variable attributes in the permission access list are kept using individual access name, own, read and write. `Owner` is a boolean flag with value true for `to own` or false for `not to own`. `Read access` and the `Write access` are of integer type with the value 0 for `not to read` or `not to write`, value 1 for `to read` or `to write`, and value 2 for `to read with password` or `to write with password`.

By default, the server can set permission rules during the start of the session. Otherwise, it is also possible to use a *selection* statement (e.g. `if` statement) in the JACIE program to determine the owner among the given users. If the owner does not include himself or herself in the permission list, the *Server Session Manager* will automatically include the owner and be added to the access list. If the shared variable has permission to be owned by all users in the permission list, then any of the owners can set the new permissions. Thus, updated

permissions will overwrite the previously defined permission.

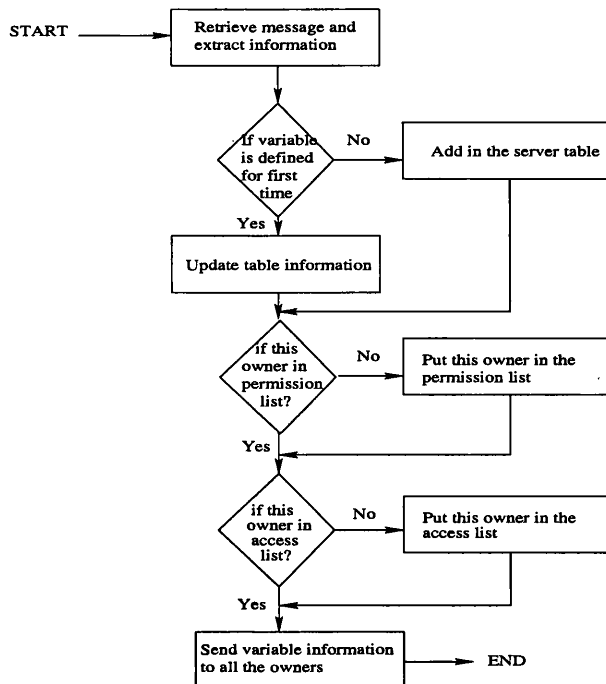


Figure 6.2: Flow Chart of the Server Control on Permission Setting.

Figure 6.2 shows the *Server Session Manager*'s operation during the setting of a permission rule. For every change of the table's content, the *Server Session Manager* will inform all the owners of the changes to the attributes. The variable attributes that are sent to the owners are the permission access, the own, read and write access, the shared variable name and the owner read password, if relevant.

The shared variable name is always required to be known by the *Server Session Manager* in order to get the index of the table in determining and setting other variable attributes. All the passwords obviously needed to be kept for validation when the read or write operation, at a later stage, requires the use of a password. The type of the shared variable, whether it is of typed integer, double, boolean or string, is needed for the write process where a value must be assigned to the variable. When the value is received by the *Server Session Manager*, it is always of type string. It is then the *Server Session Manager*'s job to pass the value in its original type to the *Global Data Manager* to actually change the shared variable's value. The variable presentation, whether it is primitive, one or two dimensional array must also be clearly indicated to be used to get or to set the particular element during the read or the write operation.

At every client site, the *Client Session Manager* also stores the shared variable's attributes in a table for any validation check by the client. The table information is shown in Table 6.3. This information is handled by the *Local Data Manager* in a manner similar to the *Global Data Manager* for the server. As in the data management at the server, this table of variable

attributes is also initially empty and data is added when the client receives any information regarding the shared variable in the standard JACIE message channel. All the variable's attributes in this table help prevent the client from sending unnecessary messages to the server that may result in too many message passing activities for data sharing and filtering.

ATTRIBUTE	PURPOSE
Variable name	As a reference of the shared variable
Owner flag	To determine 'own' permission of a variable
Read access flag	To determine 'read' permission of a variable
Write access flag	To determine 'write' permission of a variable
Owner's read password	To keep password for read validation
User read password	To keep password for read validation

Table 6.3: Shared Variable Attributes on the Client.

Like the server, the variable name is required to identify the location of the shared variable. In the JACIE compiler, the shared variable at the client is kept as another Java class generated during the compiling process. The shared variables are kept in an *ArrayList.* The order of the variables in the list is set according to the order of their occurrences during the declaration stage. The `Owner flag`, `Read access flag` and `Write access flag` are used by the *Local Data Manager* when the user who owns the shared variable performs validation during `read` operations. Similar to the server table, the `Owner flag` is a boolean type while the `Read access flag` and `Write access flag` are of type integer values. The `Owner's read password` and `User read password` are needed when the `read` operation requires password validation.

6.3.3 Assigning Value to Shared Variable

In order to manipulate shared variables, both server and client use a method called technique. Although at the server, the shared variable exists as an ordinary variable in the *Global Data Manager*, the management relies on the *Server Session Manager*. In this way, both of the server components must have a communication method between them. Therefore, there are several methods for retrieving and setting the value.

Below is a code segment that shows part of the *Server Session Manager* in determining the current user access rights. In the example, once the current user gets permission to `write access`, the `Method Table` is searched to find the variable index. Then, the given value is converted to its actual type before a call is made to actually change the shared variable's value. After this is carried out, the new value is copied to all the clients' sites.

```
if (accessYes) {              //  ◄ Current user has the access right
   ... //  Insert current user into the Access List in the Attribute Table
   boolean found = false;
   int k=0;
   while (!found && k<JACIECMethodTable.size()) //  ◄ Check index identifier in the Method Table
      if (idS.equals(JACIECMethodTable.getMethodName(k)))
         found = true;
```

```
        else k = k+1;
    int valint = 0;              //   initialisation before converting value to the actual type
    double valdouble = 0;
    boolean valbool = false;
    if (typeId.equals("1")) valint = Integer.parseInt(valuestring);
    else if (typeId.equals("2")) valdouble = Double.parseDouble(valuestring);
    else if ((typeId.equals("3"))&&(valuestring.equals("true"))) valbool = true;
    setACCESS(k,index1,index2,valint,valdouble,valuestring,valbool);
    ... // ▲ A method call to change a variable value
    ... //   Then the compiler sends a duplicate copy of variable to clients
}
```

In the client program, the same technique is applied when the *Local Data Manager* refers to the `Method Table` for a variable index reference and `Attribute Table` for access validation. There are also several methods in the *Local Data Manager* for accessing the copy of a shared variable kept in one of its component. In the case of updating the shared variable value, the event only takes place at server.

6.3.4 Access Control and Filtering Framework

As the design of interest management is incorporated into a high level language to support a programmer in building collaborative networked applications, it is desirable to acknowledge the type of applications related to the access constraints so that the rules and policies that can influence such design can be determined.

Table 6.4 shows example applications involving combinations of access control operations using `read` and `write`. The number of users involved at a time is divided into three categories: `All` (A), `Some` (S) and `One` (O). `All` refers to all the users, `Some` means the selected users and `One` simply denotes any one person from all the users, who is usually the owner in a distributed system. The actions, `Read` and `Write` are further classified into the actions verified by password. To identify a 'read only' situation, the 'No Write' condition is added. The 'No Write' is also useful to remove permissions on a user granted access. Dynamic access rights is significant due to the application requirements and conditions. All the examples are Internet based collaborative applications. The *number of users*, the specified *actions* and *password* affect the *access type* and *security* on some data.

Therefore, there are several important factors that include the following.

- Common rule access — It is common that sharing data usually includes `read` and `write` operations. Read access means that a party who owns a variable allows another party to `see` the value of the variable. Write access means that the other party is allowed to `change` the variable value. In a programming environment, data are not only seen and changed, but also another factor, called `execute`, is also included to enable raw data to be processed and converted into meaningful information. In distributed systems, to allow a user to process shared data, the `execute` factor is not appropriate, but instead, the word `own` is more suitable. In a way, the granted ownership leads to the replication of data on many sites on a networked system.

A : Permission to all users
S : Permission to some users depending on access granted by the owner
O : Only one user (or the owner) is allowed to get access
P : Short form for 'password'
w/: Short form for 'with'
X : Impossible condition

	COMMON ACCESS CONDITIONS						**ADDITIONAL CONDITION**
	Write (A)	Write (S)	Write (O)	Write w/ P (A)	Write w/ P (S)	Write w/ P (O)	No Write (A)
Read (A)	S-Whiteboard	Discussion	Web-camera	Forum	Blackboard	Booking system	Advertisements
Read (S)	X	Form Filling	QA UI	X	E-Learning	Dbase system	X
Read (O)	X	X	*Not Sharable*	X	X	Administrator	X
Read w/ P (A)	E-commerce	V-conference	Printing	Games	Games	Dbase system	Doc-download
Read w/ P (S)	X	Mgmt-info	Admin-info	X	E-banking	E-payment	X
Read w/ P (O)	X	X	Secret code	X	X	Secured data	X

Table 6.4: Example Applications Involving Read and Write Access Conditions.

- Access factors — It is very common that a password is used to restrict the access to shared data. The password is usually encrypted for actual communication.

  As password usage is a common way for restricting access, it is also necessary to limit the number of users who get access by a constraint. In general, the list of users with `all`, `some` or `one` which are mentioned above, is appropriate as a reference for the design of interest management in JACIE II.

6.3.4.1 The Access Control

Figure 6.3 shows the process of updating a shared variable in a `write` operation. Once a user tries to update any value, the server checks whether the operation is `write only` or it is a `write` operation that also involves a `read` operation. If the action involves both `read` and `write` operations, it represents that the shared variable is assigned to an expression that contains one or more shared variables. Therefore, first, the interest manager performs the `read` operation to evaluate the value on the right hand side of the assignment symbol. Then, it continues with the `write` operation.

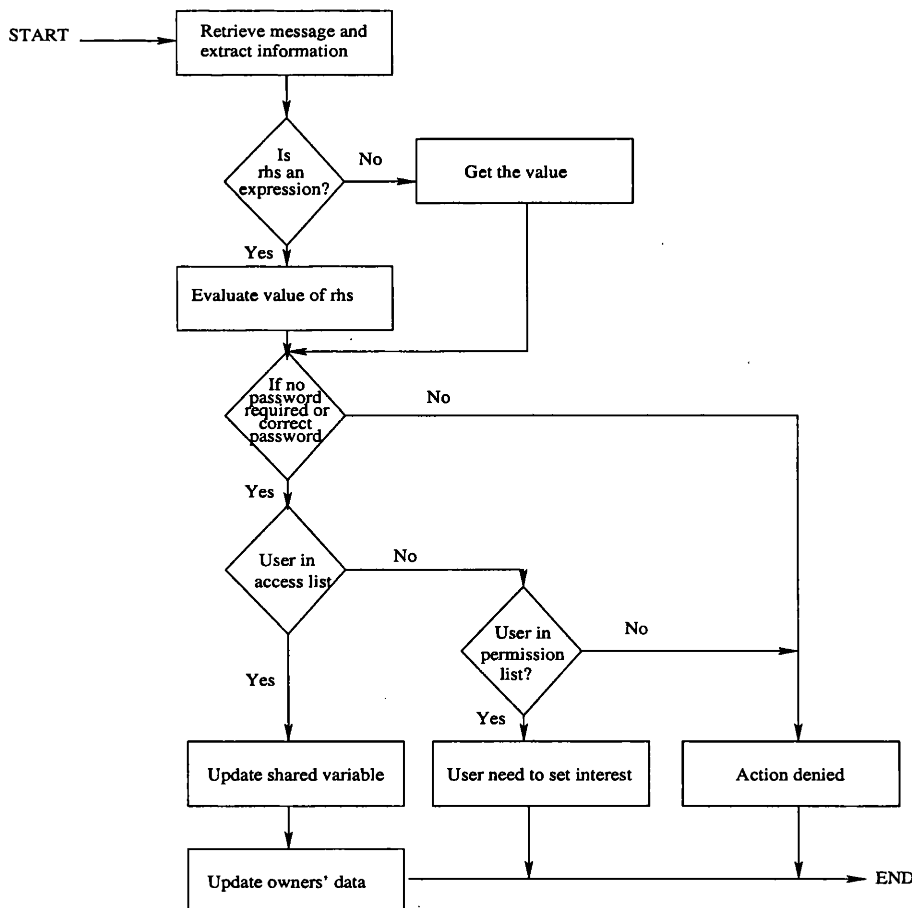


Figure 6.3: Flow Chart of Write Operation.

In the next step, the interest manager will check whether the `write` operation requires a password. If it does, the validation on the password given by the current user is performed by comparing with the owner's password kept in the variable attributes table. Upon correct validation, the interest manager will check the user `access list` in the table. The user is put in the `access list` only if the correct password is given and also the user interest to use the shared variable is already set and compared to the `interest filter`. On the other hand, without the required password, the user will be put into the `access list` for the first time on attempting to update the shared variable. Any changes in the shared variable's attribute table are sent to the owners for the shared variable's value and the client table updates.

In a `write` operation, the write access to update a shared variable is denied by the interest manager if the following cases occur. It is assumed that the user has referred to the correct shared variable in the variable attribute table.

- No password given - The interest manager via the *Server Session Manager* will first check this condition upon receiving the information on the shared variable. Upon failure to provide a password if required by the statement in setting the permission, the rest of the write operations will automatically be skipped .
- No permission on write access - At the time of password checking, the interest manager also looks at the write access permission in the variable attributes table. If there is no write permission, the write operation terminates and the rest of the write processes are skipped.
- User is not in the permission list - When a user who has an attempt to do the `write` operation and at the same time, he/she is not in the permission list, the program terminates since this action is not permitted.
- User has not give the interest set value - If the user is in the permission list but not yet in the access list, the password validation is required at this point. Upon receiving the correct password, the interest manager checks the user interest value. In the case where the user has not set any interest value, write access is denied and the `reminder message` is displayed on the server output panel.
- The interest set value is less than interest filter - In the condition where the user has given the correct password and the interest value in accessing such shared variable, the interest manager invokes the interest filtering operation. By having the interest set value less than the owner interest filter, the write operation is also denied. Detailed descriptions of the filtering operation is given in Section 6.4.4.

For the `read` operation, the process for determining the access can be at the server or at the client depending on the shared variable `own permission`. Figure 6.4 shows the server `read` operation. Similar to the `write` operation, the interest manager will first check whether the user has given any password if required for the reading access and also the shared variable `read permission`. If there is no `read permission` or the user does not provide any password when it is required, the interest manager skips the rest of the `read` operation. Otherwise, if the user is in the `access list`, the password validation is made

accordingly. Upon receiving the correct password, the value of the shared variable is sent to the user. At this stage, there is no need to check the user interest value since the inclusion or exclusion of the user from the `access list` already indicates that the filtering process has been made.

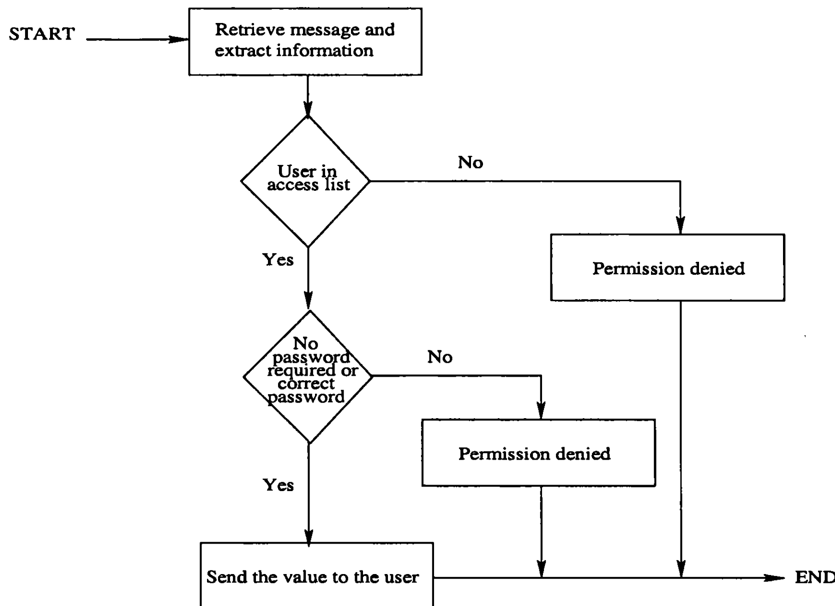


Figure 6.4: Flow Chart of Read Operation.

The `read` operation occurs at the client if the user is the owner of the shared variable. The `Read Flag` in the client's `Attribute Table` is always referred to when any validation process is needed and this process is similar for the server.

6.3.4.2 The Filtering

JACIE II chooses to implement filtering by a *preset* method. With *preset*, a user must give an `interest set` value to determine the interest desire, or otherwise, it is assumed that the user has no interest in using a shared variable. The owner of the shared variable may also need to set a value for the filtering of interest. The *Server Session Manager* determines this filtering process by comparing the `interest filter` value given by the owner and the `interest set` value by the user. The execution control throughout the filtering process is performed by the *Global Data Manager* and it is shown in Figure 6.5.

When the filtering process occurs, the interest manager checks whether the user `interest set` value is within the required range. Both the owner and user can change the respective `interest` values anytime during a session. Any changes made to the `interest filter` value must be validated upon the `Own Flag` attribute. Then, it is the interest manager's job to update the permission list or the `access list` and multicast the necessary changes to the appropriate clients.

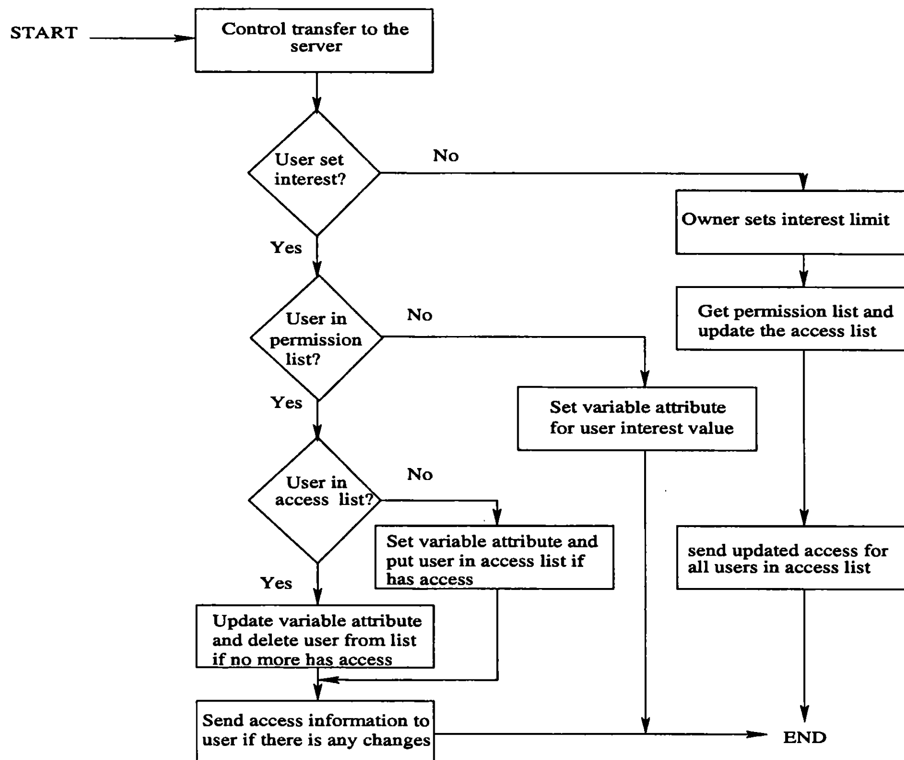


Figure 6.5: Flow Chart of Filtering Process.

For a user to give an `interest set`, firstly, the interest manager determines whether the user is in the user permission list. Upon exclusion from the list, the rest of the filtering operation is skipped. However, the value of the user interest is still stored in the table just in case there will be some changes in the later permission given by any of the owners.

If the permission list contains the current user, then the interest manager checks the content of the `access list`. The filtering process takes place where the user interest value is compared against the owner's `interest filter` value. If the user interest has a greater value than the `interest filter` value, the user must be included in the user access list if is not yet in the list. In contrast, for the case where the user is already in the `access list` and the user interest value is smaller than the interest limit, the user will be deleted from the `access list`. Any changes in the table content requires the interest manager to send the changes to the current user for updating the client variable attribute table.

6.4 Language Constructs for Interest Management

The following subsections discuss JACIE language constructs for the interest management. With similar design concepts to the previous interaction management language constructs, these constructs also allow several options and tags in a statement to facilitate fast scripting.

These constructs are designed based on the factors presented in Section 6.3. In summary, the constructs consist of the followings.

- An owner permission (`use` statement (Section 6.4.1)).
- A read operation (`set` and `check` statements (Section 6.4.2 and Section 6.4.3, respectively)).
- A write operation (`set` statement (Section 6.4.2)).
- A filtering operation (`filter` and `interest set` statements (Section 6.4.4)).

6.4.1 *Statement:* **use**

```
use <shared variable> by <user list>
     [to own | not to own]
    [to read [with password <String>] |
         not to read]
   [to write [with password <String>] |
                 not to write]
```

The `use` statement is employed by the *owner* of a shared variable to set its access permission. When a shared variable is first declared, by default, it is owned by the server, and can only be read or written to by the server. It is common for the server to re-assign the ownership at the initialisation stage of the server section of the program (i.e., `on session start`). The owner (or owners) of the variable can modify the access permission during a JACIE session, provided that appropriate mutual exclusion rules are observed.

The `<user list>`, which is the permission list, is one of the owner's access rule to indicate the number of users who are given the access rights. The list can be one of the following:

- `all` - All clients.
- `others` - All clients except the owner or owners.
- `group` *grp* - This is only applicable when a group interaction protocol is in use. It implies all the members of the group specified by the group identifier *grp*.
- `me` - The owner or owners.
- { $\mu_1, \mu_2, ...$ } - This is a list of client identifiers.

The access permission contains the common access states with several options that include:

- `to own` - All clients in the `user list` will become the owners of the variable.
- `not to own` - All clients in the `user list` will be stripped of the ownership. Regardless of the assignment of the ownership, the server will remain as the owner of any shared variable throughout a JACIE session.

- `to read [with password <String>]` - The variable will become readable by all clients in the `user list`. If the statement contains a `with password` option, clients are required to specify a password in `<String>` when invoking a `set` statement to obtain a copy of the shared variable. The same access right is given to the user to make any comparison on the shared variable in the `check` statement.
- `not to read` - The variable will not be readable by any clients in the `user list`.
- `to write [with password <String>]` - The variable will become writable by all clients in the `user list`. The same rule on password specification applies as `to read` specification. It is for invoking a `set` statement to assign a new value to the shared variable.
- `not to write` - The variable will not be writable by any clients in the `user list`.

Upon processing this statement, either at the server or at the client, the execution control must be transferred to *Server Session Manager* to assign the variable's attributes.

With the password option, the owner can specifically state the password by giving a string value in this statement. The JACIE compiler treats the password value as a string type. The password can be defined one or more times during a session. Once a password is defined for a shared variable, a client program can access the variable by including the password in the statements that require password validation. If the password is omitted in such a statement, JACIE will automatically prompt the user of the client program with an input instruction to enter the password every time the `use` or `set` statement is executed.

6.4.2 *Statement:* **set**

```
set [with password <String>] <local variable> =
<sv-expression>
```

```
set [with password <String>] <shared variable> =
<ordinary expression> | <sv-expression>
```

The `set` statement is used to handle any assignment statement that involves shared variables. Unlike an ordinary assignment, such an operation requires the validation of access permissions for all shared variables involved. The keyword `set` instructs the interest manager at the server (if the `read` operation is performed by any user who is not the owner) to validate the read-access permission for all shared variables in an expression involving shared variables (which is denoted as `<sv-expression>`) and the write-access permission of the variable to be updated if it is a shared variable.

At the right hand side of the assignment, any `<sv-expression>` that requires more than one password verification would need a list of passwords to be included at its left hand side. With this approach, the interest manager will make the 'mapping' between the password and its corresponding shared variable using the attribute table (Table 6.2). Furthermore, this

allows the compiler to parse JACIE expression (Appendix C.2.8) in a straight forward way. Therefore, the inclusion of these constructs into JACIE allows its main language structure, specifically its `<ordinary expression>` remains untouch.

Any failed validation on permission access will result in an error report in the JACIE standard message channel, and will cause the program to skip the entire `set` statement. However, such a failure will not cause the program to abort.

One can of course use the `set` for an assignment that involves no shared variable at all, that is, `set <local variable> = <ordinary expression>`. However this will incur additional processing and communication time for validation.

On the other hand, when one uses a shared variable in an ordinary assignment, the JACIE compiler will report this as a compilation error. This error is detected as the shared variable is not kept as a local variable with direct declaration in the Java equivalent program.

This `set statement` is like a write statement if the shared variable occurs on the left-hand side of the assignment symbol, and like a read statement if the shared variable only occurs at the right-hand side of the assignment. The write process is always validated by the server. However, the read process is dependant on the `own permission` access.

6.4.3 *Statement:* **check**

```
    check <conditional sv-expression> <statement>
 {else check <conditional sv-expression> <statement> }
       [else <statement>] [default <statement>]
```

The `check` statement is used in place of `if` when the conditional expression involves one or more shared variables. The statement causes the program to validate the read-access permission of all the shared variables. Therefore, the validation process is as in Figure 6.4.

Any failed validation will result in an error reported through the JACIE standard message channel, and will cause the program to skip the entire `check` statement, except for the `default` component of the statement if it is included in the `check` statement. The `default` component is activated only upon a failed validation of the read-access permission of shared variables in `<conditional sv-expression>`.

6.4.4 *Statement:* **filter** *and* **interest set**

There are two language constructs for managing data filtering.

```
filter <shared variable> <conditional operator> <value>
```

```
interest set <shared variable> <value>
```

The `interest set` statement allows users to set their interest on the shared variable. If it happens that the user, without any access permission, uses this statement, the action on this statement will be ignored. The statement `filter` allows the owner, who by default is the server, to set the level of access on any shared variable. The `conditional operator` can be any one of the relational operators: '=', '<', '<=', '>', '>=' or '<>'. The `value` specified must be 0 or 1 or any real value between 0 and 1. The value 0 will automatically indicate the denial of access or no interest in using such a shared variable. In contrast, the value 1 indicates the full access from the owner, and the very high interest of the user. Other values mean that some evaluations must be made.

For the interest level, only the user can determine the value. The user can give the value depending on how important is the shared variable in the design of the applications or simply based on the interest in reading or writing the shared variable. For the access level, the server can make the decision based on how important it is to protect the shared variable. The decision can also be made based on a certain percentage of the number of current users. However, if it happens that in an extreme case, all the users with access permission claim the full interest of a single shared variable, the server must fulfil the access. There is no limitation as the situation guarantees that no messages will be wasted when the server is sending the copy of the shared variable of any information regarding the variable's attributes. In contrast, if the user in the access list does not set any interest level, it is assumed that the user has no interest and no access will be granted. As a whole, the user interest `value` varies depending on the application to be implemented.

6.5 Language Enhancements

Interest management is new to JACIE with consequent changes to the language and major modifications to the compiler. In JACIE I, data sharing had to be carried out through user-defined messages using `send` and `receive` commands. With interest management, the number of message passed during a session is reduced with reduction in program size. The following subsubsections describe all the changes and the additions made to the JACIE I compiler.

6.5.1 Additional Tokens and Productions

The new tokens required for interest management are introduced into the *jacie.flex* file (Figure 4.2). Some of the example tokens are illustrated below.

```
"by"                    { return symbol(sym.BY); }
"set"                   { return symbol(sym.SET); }
"get"                   { return symbol(sym.GET); }
"check"                 { return symbol(sym.CHECK); }
"with"                  { return symbol(sym.WITH); }
"mine"                  { return symbol(sym.MINE); }
"password"              { return symbol(sym.PASSWORD); }
```

The *jacie.cup* file (Figure 4.2) has to be extended to deal with the parsing of the new language features. The following code segments show some declarations of the non-terminals of type `JACIECStatement` and `java.lang.String`. The non-terminals of type `JACIECStatement` are validated in Java *JACIECStatement Class* which generates the correct form of Java statements, while the `java.lang.String` represents any non-terminal symbols in JACIE II productions.

```
...
non terminal JACIECStatement permission_statement;// ◄ type java class JACIECStatement
non terminal JACIECStatement access_statement;
non terminal JACIECStatement filtering_statement;
non terminal JACIECStatement filter_choice;
non terminal JACIECStatement filter_statement;
non terminal JACIECStatement set_statement;
non terminal java.lang.String option_select;       // ◄ type java.lang.String
non terminal java.lang.String option_rwChoice;
non terminal java.lang.String option_wChoice;
non terminal java.lang.String option_to1;
non terminal java.lang.String option_to2;
non terminal java.lang.String option_to3;
non terminal java.lang.String option_not1;
non terminal java.lang.String option_not2;
non terminal java.lang.String option_not3;
non terminal java.lang.String password_choice;
non terminal java.lang.String str_choice;
non terminal java.lang.String permission_list;
non terminal java.lang.String user_list;
non terminal java.lang.String one_list;
non terminal java.lang.String more_list;
non terminal java.lang.String rel_sym;
...
```

The `check statement` construct for interest management is similar to an 'if' statement, so the production associated with this statement must be able to cope with multiple nesting. This is indicated in the following code.

```
statement     ::=  statement_without_trailing_substatement:swts
                   ...
                   |
                   while_statement:wstat              // ◄ the While statement
                   {: RESULT = wstat; :}
                   |
                   check_then_statement:ctstat        // ◄ Check statement
                   {: RESULT = ctstat; :}
                   |
                   check_then_else_statement:ctestat  // ◄ Check with Else statement
                   {: RESULT = ctestat; :}
                   |
                   ...
                   ;
```

The production for `use` statement, `set` statement and `filtering` statement are as follows.

```
statement_without_trailing_substatement ::=  compound_statement:compstat
                                             {: RESULT = compstat; :}
                                             |
                                             expression_statement:estat
```

```
{: RESULT = estat; :}
|
...
|
permission_statement:prmstat
{: RESULT = prmstat; :}
|
access_statement:accstat
{: RESULT = accstat; :}
|
filtering_statement:filstat
{: RESULT = filstat; :}
|
...;
```

Productions used in checking the syntax of some of the other interest management features appear in the next example. Some new Java classes are invoked for parsing and semantic processing.

```
permission_statement::=  USE IDENTIFIER:id BY permission_list:prsml
                                          option_select:ops  SEMICOLON
                         {: RESULT = new UseStatement(id,prsml,ops); :}
                                       // ▲ invoke the java Class UseStatement
                         ;

access_statement    ::=  SET password_choice:psdc assignment:assg  SEMICOLON
                         {: RESULT = new SetStatement(psdc,assg); :}
                                       // ▲ invoke the java Class SetStatement
                         ;

filtering_statement ::=  filter_choice:fc SEMICOLON
                         {: RESULT = fc; :}
                         ;

filter_choice     ::=  filter_statement:flim
                       {: RESULT = flim; :}
                       |
                       INTEREST set_statement:fset
                       {: RESULT = fset; :}
                       ;
                       // ▲ filtering options whether to limit or set the interest

filter_statement::=  FILTER IDENTIFIER:id rel_sym:rs FLOATING_POINT_LITERAL:ft
                    {: RESULT = new IntLimitStatement(id,rs,ft.doubleValue()); :}
                    ;

set_statement     ::=  SET IDENTIFIER:id FLOATING_POINT_LITERAL:str
                    {: RESULT = new IntSetStatement(id,str.doubleValue()); :}
                    ;
```

6.5.2 Additional Code and New Java Classes

New Java classes are introduced in the compiler for every new statement introduced and for handling several tables for data sharing. Table 6.5 lists some of these classes.

Most code modification is contained in the *Global Data Manager*, *Server Session Manager*, *Local Data Manager* and *Client Session Manager*. When a shared variable is declared in the

CLASS NAME	PURPOSE (in code translation process)
JACIECMethodTable	A table to keep shared variable index at runtime
JACIECSharedIdTable	A table to keep shared variable during compilation
JACIECSharedVarTemplate	A table for client shared variable attributes
JACIECSharedVariableTemplate	A table for server shared variable attributes
UseStatement	To handle `use` statement
SetStatement	To handle `set` statement
CheckStatement	To handle simple `check` statement
CheckElseStatement	To handle nested `check` statement
FilterStatement	To handle `filter` statement
IntSetStatement	To handle `interest set` statement

Table 6.5: New Java Classes for Interest Management.

client program, the code translator will generate the same copy of the shared variable in the *Global Data Manager* at the server using the same identifier name. This can be performed since all the shared variables are kept in the table `JACIECSharedIdTable` during the compilation process. The *Server Session Manager* handles the rest of the tasks for handling shared data and interest management.

At the client, the *Client Session Manager* only adds the new message header identifiers and receives any message associated with the shared variable from the server. The rest of the data handling is performed by the *Local Data Manager*.

6.6 Technical Considerations

Sharing global variables remotely requires proper management. Several issues must be considered to ensure that the variable has the same value even though it can be distributed in several places. The users who can manipulate this variable must be determined correctly. In this design of language constructs that involve global variables, the issues such as mutual exclusion, the management of permission list and access list are important.

6.6.1 Mutual Exclusion

As mutual exclusion requires only one process (user) uses a shared resource (variable) at one time and other processes (users) are excluded from doing the same thing [310], it is always related to a *critical section* [223]. Here, in our work, a critical section is a part of JACIE program where an owner is setting a shared variable permission and also when an authorised user is trying to do `write` operation on a shared variable.

In general, JACIE interaction protocols that provide proper scheduling of users' activities, are able to ensure that any attempt to do an update on a shared variable is in a mutually exclusive manner. Almost all the provided protocols allow only one of the users to have the turn control which allows the manipulation of shared variables. During a turn, the selected

user is performing any task while others are 'blocked' from doing any work. Therefore, in this way, it is not possible to have two or more users attempting to manipulate shared variables at the same time.

However, in protocol *contention* where there is no 'proper' turn scheduling, mutual exclusion can also be implemented by the use of a message queue at the server. In this case, whoever are fighting for the turn must send a kind of data using a JACIE message. Messages from users are added into the server queue and they are managed in *first in first out* order where only one message is retrieved at a time.

Since Java language supports multi-threading and JACIE messages can be processed in many different threads, a JACIE programmer is required to include a control, `if-then` statement in handling a *critical section.* This statement is used usually to check several conditions on the current environment to ensure mutual exclusion can be achieved. Furthermore, in executing these critical conditions, the control of execution in the JACIE program (user-defined codes) is transferred to the JACIE server program (pre-defined Java codes). In particular, this situation is like a call from a high level system to a lower level that can guarantee the objectives of the given tasks are achieved successfully.

6.6.2 Permission List Management

For variable sharing in a client/server system, a shared variable must belong to someone and by default, the server is the owner or otherwise, it can keep the copy of the variable and has the full control with joining ownership. At the start of a session, the owner of any shared variable must have declarations on the variables' permission rules. One of the rules is to state a list of users who can gain access. In JACIE, a permission list of a shared variable contains the identifier (`user number`) of all the permitted users. Even though JACIE supports group collaboration, this list still keeps the user's identifier instead of group identifier. In this case, the *JACIE group manager* can always provide the list of users in each group.

The content of the permission list is updated when the execution of the `use` statement (Section 6.4.1) is performed. During the execution, the control is transferred to the JACIE pre-defined codes in one of the *Global Data Manager* components. It is possible that a shared variable's permission is changed several times during a program execution. These actions require the *Global Data Manager* subcomponent to overwrite the content of the previous list and always keep the recent and updated information. Reference to the the content of this list is made through index references. When an item is added, the index reference is updated by incrementing the index counter, while the deletion of an item from this list can be made by removing the item from the list and decrementing the value of the index counter. In Java, this list is a *Vector* class which allows the operation regarding this list to be made through several methods.

In group collaboration, members are determined at the beginning of a session. JACIE does not facilitate the change of members throughout the application and its interaction protocols are concerned on the turn control of groups and the selection of a particular member to

represent the group during the turn control. Therefore, in the case where one of the users or group stops the collaborative work by terminating the connection to the server, a new session is started and the permission list is reset.

6.6.3 Access List Management

In addition to the permission list, JACIE also creates an access list to guarantee that the access to a shared variable is only performed by the appropriate user. Similar to the declaration of the permission list, this list is represented by the individual user. For group access, checking must be made by referring to the *group manager* for getting the list of group members. Like permission list, the implementation of this list also uses the same type of Java classes.

There are several conditions on adding or modifying the content of the access list which include the following.

- The execution of the `use` statement (Section 6.4.1).
- The execution of the `write` operation.
- A user in either the permission or access list changes the `interest set` value.
- The owner of the shared variable changes the value of the `interest filter`.

Unlike the permission list that requires the update operation on only the execution of JACIE `use` statement (Section 6.4.1), the access list can be updated at several parts of the program. During the execution of the `use` statement (Section 6.4.1), the user who is the owner of the shared variable is added into the access list. For other users who are listed in the permission list, their inclusion and removal from the list are determined according to the factors mentioned above.

6.7 Secret Switch

A secret switch is one of the variations of the noughts and crosses game. This game is implemented to test the language constructs for the data sharing mechanisms. The rules of the noughts and crosses game is modified in such a way that both players have to set their own password for the opponent to guess. The game turn alternates and each player has to guess the opponent's password. The correct password guess allows the player access to change the content of the board. The filtering of interest in this game can still be used but, it may be looked at as an added restriction to the board access.

This game was implemented in three different ways. First, it was implemented without the new language constructs and the winner of the game is determined by the server. The other two programs used the new interest management statements with one of them determining the winner at the server and the other checking the winner at the client. Below is the first implementation.

In the example, the board control and password verification must be done by the server with the support of the `send statement` and `receive statement`. Therefore, both players must send their passwords to the server to keep for later verifications. When they get the turn control, they must key in their guess passwords to be sent to the server and wait for the server to verify. The server sends an answer through the standard message channel using the user-defined type message that uses `send statement` and `receive statement` to instruct the player to play the game when the correct password is guessed. Otherwise the player's turn is terminated. In this implementation, there are many message passing activities involved.

```
client implementation {
  declaration {...
     string[8][8] boardP = "";          //  board declaration
     string my_pwd = " ";               //  owner's password
     string guess_pwd = " ";            //  guess password
  }
  on session start {...
     state = SET_PASSWORD;
  }
  on session {...
    if (opponentPWDanswer == correct)   //  check condition if need to reset a password
       state = SET_PASSWORD;
    ...
    on MOUSECLICK {
      gX = GETGRIDX;                    //  get a grid point on the canvas
      gY = GETGRIDY;
      if (state == PLAY) {
        boardP[gX][gY] = playSymbol;    //  update the board
        send gridX gX;                  //  send required data to the server
        send gridY gY;
        send symbol plySymbol;
        ...
      }
      else if (state == SET_PASSWORD) {
        print servermessage "Please set your new password for opponent to guess";
        input my_pwd;                   //  ◀ enter text using textinput bar
        send ownPwd my_pwd;             //  send new password to the server
        state = WAIT_PASSWORD;
        }
      else if (state == WAIT_PASSWORD) {
        print "Enter your guess for password";
        input guess_pwd;                //  ◀ enter text using textinput bar
        send clientPwd guess_pwd;       //  send guess password to the server and wait for verification
        ...
        receive ansPwd answer;          //  get the verification
        if (answer == correct) {
           state = PLAY;
           print "Correct password! Pick a position on the board";
        }
        else {
           state = WAIT_PASSWORD;
           print "Wrong password! Lose your turn";
           ...
        }
    }
  }
}
   server implementation {
      declaration {...}
      on session {...
        ...    //  validate password and determine the winner
```

```
    }
  }
```

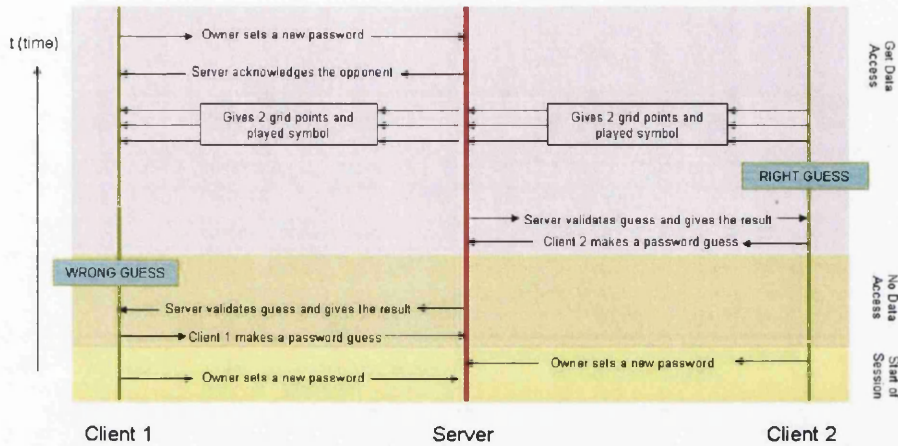
Below is the code segment that shows the implementation of the game with the second method. The password verification processes are carried out in the pre-defined JACIE code that automatically checks, verifies and makes any changes to the board game. The processes are performed through the JACIE standard message facility in the JACIE pre-defined compiler codes instead of user-defined type message that uses `send` statement and `receive` statement. As can be seen, this lower level verification uses none of these statements.

```
client implementation {
  declaration { ...
     shared string[8][8] boardP = "";      // declaration of a shared variable
     string my_pwd = " ";                  //  owner's password
     string guess_pwd = " ";               //  guess password
  } ...
  on session start {...
     state = SET_PASSWORD;
  }
  on session {
     if (opponentPWDanswer == correct)    //  check condition if need to reset a password
        state =  SET_PASSWORD;
     ...
     on MOUSECLICK {
        gX = GETGRIDX;                     //  get a grid point on the canvas
        gY = GETGRIDY;
        if (state = PLAY) {                //  try to update the board
          set with password guess_pwd boardP[x][y] = playerSymbol;
          ...
        }
        else if (state = SET_PASSWORD) { ...
          my_pwd = selected_pwd;           //  set new password for permission
          use boardP by all to own to read to write with password my_pwd;
          state = WAIT_PASSWORD;
        }
        else if (state = WAIT_PASSWORD) { ...
          guess_pwd = selected_string;
          state = PLAY;
        }
     } ...
  } ...
}
server implementation {
      declaration { ... }
      on session {
        ...       // determine the winner
      }
  }
```

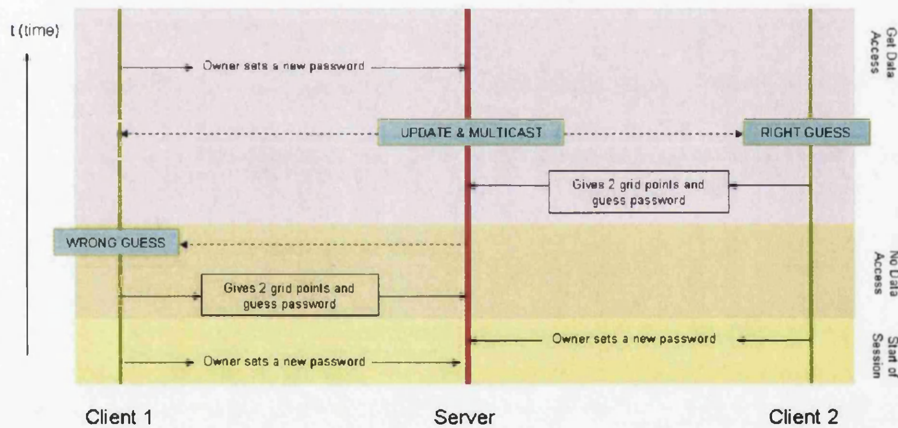
The third method is similar to the second method shown above. The slight difference is that the winner of the game is determined at the client, instead of at the server. After every event of the board game being updated, an algorithm that resides at the current client is executed to determine the winner. Once the winner is determined, the client sends a message to inform the opponent through the server before the game is ended.

In general, the implementation of these various methods allow us to test the significance and usefulness of the new language constructs. Therefore, the output for all the message

passing activities are printed out by the server program. Figure 6.6 shows the summary of those activities at a given time interval, where the upwards direction indicates time increasing. It also shows three different conditions which are represented by three different colours.



(a) **Message Passing Activities Without Interest Management**



(b) **Message Passing Activities With Interest Management**

Figure 6.6: Summary of Message Passing Activities.

The timing activity in Figure 6.6 can be described as follows.

1. Initial phase when password is first set up — At the beginning of a session, both clients have to set their passwords for the opponent to guess. By default, client 1 gets the turn control in the round robin protocol and sets a password. This password is sent to the server. Then, the same action goes to client 2 when client 2 gets the turn control. Here, this particular condition is referred to as *Start of Session*.

2. Client 1 makes a password guess and gets a wrong answer. Therefore, the board access is denied — In case (a), it shows that client 1 tries to guess the opponent's password by

sending the guess password to the server for validation. Later, the server responds and sends a reply for its result. For case (b), that involves interest management constructs, client 1 sends the guess password using the `use` statement. Then, the validation is performed automatically by the JACIE II predefined server program. Therefore, the server informs the result through the system defined message identifier. This condition is referred to as *No Data Access.*

3. Client 2 makes a password guess and gets it right so the game board is updated — When this condition occurs, there are more message passing activities in case (a) compared to case (b) because of the following actions.

   - Client 2 sends the guess password.
   - Client 2 receives server message for validation.
   - Client 2 sends three messages for the three different data in the game.
   - Server forwards these three new data to client 1.
   - Server sends a message to client 1 for the acknowledgement of changing the password because the current password is invalid.
   - Client 1 sends the new password.

   Therefore, there are a total of 10 messages to and from the server for both clients. In comparison, case (b) has only 4 message passing activities for the following actions.

   - Client 2 sends the required new data and the guess password in one message.
   - Server updates the `shared variable` and sends two messages that contain the required information (in one message) for both clients. This is also performed by the JACIE II predefined server program.
   - Client 1 has to send the new password to the server when receiving the updated `shared variable`.

   Here, this third condition is referred to as *Get Data Access.*

In executing the whole application, the first condition, *Start of Session*, occurred only once while the other two conditions, *No Data Access* and *Get Data Access*, may happen several times.

The number of message passing activities for these three conditions are determined when this secret switch game is implemented using the following methods.

- Method One: *Secret Switch without new language constructs*
- Method Two: *Secret Switch with new language constructs and determination of winner at the server*
- Method Three: *Secret Switch with new language constructs and determination of winner at the client.*

Table 6.6 compares these three methods for the above three different conditions and adds another factor called *Winner Info.* This factor is used to determine the number of message

passing activities when the winner determination algorithm is executed at server (for Method One and Method Two) or at client (for Method Three). In Method One and Method Two, the server sends each client a message that contains the information of the winner. In Method Three, the winner usually stops the game after sending a message to the server. Then, the server sends a message to the loser to end the game. In the table, the entities presented by letters 'P', 'S' and 'O' stand for the player, server and opponent, respectively. The numbers indicate the actual number of messages sent from those entities.

Besides looking into the number of message sending activities, we also calculate the number of lines in the programs. One way of doing the calculation is by running a Linux command `wc` in the Linux operating system environment. Therefore, the calculation is performed and added to this table.

METHOD	START OF SESSION			NO DATA ACCESS			GET DATA ACCESS			WINNER INFO			LINES
	P	S	O	P	S	O	P	S	O	P	S	O	
One	1	0	0	1	1	0	4	5	1	0	2	0	384
Two	1	0	0	1	1	0	1	2	1	0	2	0	330
Three	1	0	0	1	1	0	1	2	1	1	1	0	304

Table 6.6: Comparison on the Secret Switch Implementations.

From the data stated in Table 6.6, we can conclude that the interest management constructs reduce the number of message passing activities at either client or server when dealing with the `shared variable`. The main advantage is that the `set` statement allows several data to be transferred within one message. In terms of programming codes, the constructs help reduce the number of lines in the program by avoiding multiple `send` and `receive` statements.

Figure 6.7 shows a screenshot of the Secret Switch noughts and crosses game, which features the above example code segment of Method One. The game has two players, each sets a password at the end of his/her turn. The password, which is implemented as a string type variable to be passed to the server for access validation, can have a combination of four letters word, consisting of 'A', 'B', 'C' and 'D' (for this specific example). The opponent must guess the password correctly in order to have a turn.

There is a *Text Input bar* at the bottom of the canvas for the user to enter their own password or to guess the opponent password during the turn. The game instruction is displayed on the *JACIE Local Message bar.*

Since this example is simple involving only two players, it is not practical to apply the filtering methods provided. Chapter 7 provides a more practical e-learning application using these language constructs.

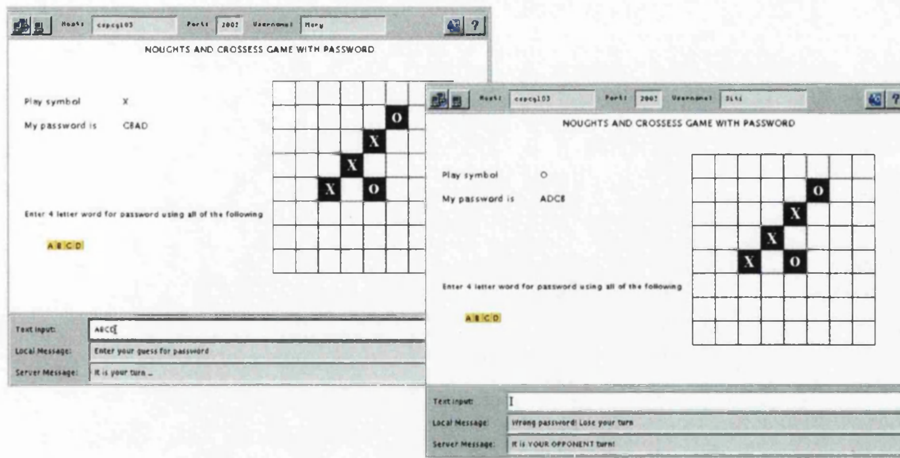


Figure 6.7: Screenshot of the Noughts and Crosses Game with Password.

6.8 Summary

In this chapter, we have presented the design of a set of language constructs for interest management, and our implementation through a major extension to an existing scripting language JACIE. This set of language constructs can provide network programmers with a high-level programming interface for managing shared variables among the server and different clients. This enables some common issues, such as data security, access control and information filtering to be addressed by a programmer in an intuitive manner. In addition, by incorporating such language constructs in a programming language, we are able to transform interest management from a traditionally hard-coded facility in specific applications to a general-purpose facility. With the rapid growth of network-based applications, we believe such a facility will become an dispensable feature in many programming languages in the future.

Chapter 7

JACIE Applications and Performance Analysis

Contents

7.1 Introduction 154
7.2 Bridge Game 155
7.3 Implementation of the Bridge Game 157
7.4 E-learning on Simulation of Network Trouble Shooting 166
7.5 Performance Analysis 177
7.6 Summary 186

7.1 Introduction

This chapter describes two examples of JACIE applications that have been implemented using the new language constructs for interaction and interest management. One application is a complex game, online Contract Bridge, that focuses on interaction protocols and the other is an example of an e-learning group exercise, the Simulation of Network Trouble Shooting, which is concerned with interest management. At the end of this chapter, the discussion on the JACIE performance is performed to verify the significance of this scripting language.

The implementation of the Bridge game signifies the need to have dynamic interaction protocol. In this chapter we discuss the relevant features of this game and its implementation using JACIE II, which include the overall program structure and several examples on the use of different interaction protocols in implementing the same application.

The initial e-learning groupware was implemented in JACIE I, which does not have any interest management features other than sharing information through a chat channel. Its focus was to show that JACIE I could provide multiple canvas displays. With multiple displays, a user could have a 'global' view of the network representing work spaces of all the

users in one canvas and also have a 'local' view of his/her own workspace represented by another canvas, as shown in Figure 1.2 in Chapter 1. Through discussion supported by a chat channel, which was a common approach adopted in many e-learning environments, the user could work without sharing his/her work space. He/She has total control and would make any changes according to the discussion and suggestions made during the collaboration. At any time, all users were free to switch back and forth between the 'local' view (working canvas) and the 'global' view (overall network diagram) during the discussion. With interest management, JACIE II allows the 'local' view of one's work space to be sharable with the access control monitored by the owner. In this way, flexible data manipulation for user collaboration at a high level can be achieved without relying on discussion through chatting.

7.2 Bridge Game

A contract bridge card game is played by four players in a partnership. Partners are determined by the seating arrangement; namely West, North, East and South. The person who sits at North is automatically the partner of the person at South and they play against the other partners, West and East. Although they are partners, no communication is allowed between them except via the bidding process seen by all players.

There are many types of bridge games [222]. Many books [178, 327], research papers [277], and websites as well as software (such as the GIB software) are devoted to this game. Most of them discuss the techniques of the bidding process because this is the most important part of the game that can lead the player to consider the tactics to win. There are also internet games of bridge that allow users to play against a computer. In this work, we concentrate only on the basic and common game rules. Figure 7.1 shows some examples of bridge game images that represent (quite similar) representation of the 'seating arrangement' of the users.

Bridge has two basic modes, the bidding process and the trick play, which both require different interaction protocols. The number of active players in both modes are also different. The bidding process has four players and then it is reduced to three as one of the players becomes the *dummy* in the trick play mode.

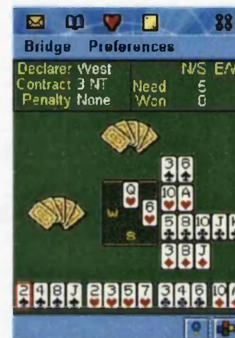
At the initial stages of this work, the game was implemented as a benchmark using JACIE I. This allowed us to see how JACIE I handles changes in the user interaction and controls the access to card information. We implemented the game with only one protocol setting throughout the game, which is due to the protocol setting in JACIE I that could be only be set at the beginning of a session. Information sharing was performed using the message passing facilities, *send* and *receive* statements. We then reimplement the game with JACIE II.

The bridge game is chosen as a benchmark because it has the relevant features that are very useful to test our design, especially as regards of interaction management. Some of the interesting game features are:

GIB 6.1.5 [26]

Bicycle Bridge V3.0. [14]

Pogo game [255]



SymMobile game [306]

Figure 7.1: Example of Bridge Games.

- **Team game** — Although the game divides the four players into two teams, the team members cannot share any information about the cards in their hands. Therefore, this rule requires the players to have bidding strategies, whether to place a bid or pass the bidding process that can enable them to win the trick play.
- **Game protocol** — The bidding process and the trick play require different sequences of turn control. In the bidding process, the turn follows a round robin in clockwise order. In the tricks play mode, the turn order depends on the bidding outcome for the first round and the rest relies on the outcome of the previous round of cards played. For the team that consists of the declarer and the dummy, only one member of the group can play. This results in a change in the interaction protocol.
- **Cards information sharing** — During the bidding process, information on all cards is private. However, when a dummy player is determined, the dummy's cards must be shown to all other players after the first card is played. In this way, the rule for information access has changed.
- **Complicated game rules** — This game has many rules that makes it a complicated game to program. For example, it requires different turn scheduling for different game mode, maintains information on the team members, stores the bidding history and keeps up with information on the individual player's cards, needs proper algorithms to

determine the dummy and decide the winner on every round in the trick play, updates display information and much more.

After implementing this game as the benchmark, some alterations and enhancements together with major extension to JACIE I have been undertaken. Several issues have arisen that concern programming this game in general that include dynamic interaction protocol (Section 5.6.1), code repetition for programming grid manipulation on the JACIE I canvas and many variable declarations that require many lines of codes. Some of JACIE I enhancements used are described in Chapter 4.

Since most of our case studies in this thesis are in terms of games, it is useful to compare game environments with collaborative working environments. At a glance, games can be viewed as simplified collaborative environments that are manipulative or active. They are represented either in a two or three dimensional presentations. Games can be used as approximations for real life applications. Throughout this thesis, games have been used as case studies for designing interaction management. We also use simple example of games for designing interest management features while most other research have used simulation techniques [8, 20]. Therefore, a game can be used as a research tool and a vehicle for understanding design actions in a restricted environment. The focus is usually on a specific aspect of the design as argued by Brandt and Messeter [42].

Like noughts and crosses, which uses cells in the game, Baughman *et al.* [27] provide cell-based protocols within local regions to reduce network traffic. Their work, carried out through simulation, purposely focus on centralised and serverless online games. Much research has also be undertaken using Conway's Game of Life [49], which is also a cell-based game that consists of a space-time model namely cellular automata (CA) [273]. In [149], this game is used as an evolution of a learning base in a character recognition system, while in [273], parallel techniques have been applied to the cellular automata algorithms. In general, collaborative working environments can be much larger and more complex than games.

7.3 Implementation of the Bridge Game

With JACIE, much less development skill and effort are required of the programmer for this task than what would be needed with Java, assuming no previous experience in either language. This section describes the implementation of this game highlighting some aspects of the program and focusing mainly on interaction management to test the technical feasibility and usefulness of the high level language constructs.

7.3.1 Program Flow and Game Layout

Once the program is compiled and executed at a specified web address, each user has to *log in* by entering the *hostname* and *username*. After four users have entered the correct information, the game starts by assigning the first user as the first player. We omit the usual rule

in determining the first player in order to concentrate on the more important features of the game. Figure 7.2 shows the screenshot of the game that represents the JACIE canvas viewed by the user at position 'W' (West). At the top bar is the *hostname* called `cspcgl03`, port number `2000` set by the programmer and the *username* called `Siti`. There are also several command buttons in the toolbar to perform actions such as connecting and disconnecting to the server and information regarding the application.

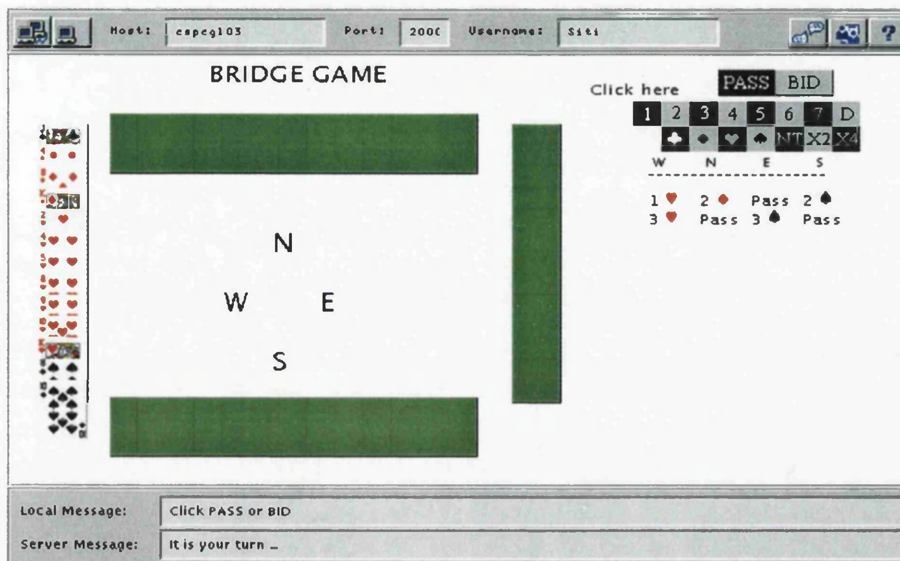


Figure 7.2: Bridge Game (The Bidding Process).

In JACIE, the game layout can be drawn on a canvas for each user where all the users see the same canvas view at the beginning of the game session as all users have the same copy of the program. Once the user is connected to the server, the server assigns an identifier (user number) to each user that is determined according to the user's arrival.

The program flow of this game is as follows:

- **Setting a player's cards —** The server generates 13 cards at random for each user and sends the card information to the respective user at the start of a game. Below is the example code segment in the server program where the card selection is performed in a JACIE method `drawCard`. The cards played by a user are presented by integer values. Since the total number of the cards played in this game is 52, any number from 0 to 51 is selected each time the method `drawCard` is called and the selected value is then removed from the original list `numberOfCards[]`. The selected value is sent to the client one at a time using the *send* statement.

```
server implementation {
  declaration { ...
    shared int drawCard() {                    // module to draw a card
       boolean drawn = false;
       int drawnCard = -1;
       int index;
```

```
        while (!drawn) {
          index = rnd(52);                    // ◄ random selection
          if (numberOfCards[index] !=-1) {
             drawnCard = numberOfCards[index];
             numberOfCards[index] = -1;
             drawn = true;
          }
        }
        return drawnCard;
     } ...
  }
  on session start {
     for (int i=0; i<13; i=i+1) send newCard drawCard(); // ◄ send a card to a user
     ...
  } ...
}
```

In a client program, the cards received are also represented by integer values, which are then illustrated on the JACIE canvas as images. Below is the example JACIE code that shows the client program which receives the cards and sorts them for the purpose of canvas display at a later time.

```
on session start { ...
   for (int i=0; i<13; i=i+1) {
      receive newCard currentCard[i];         // ◄ receive a card from the server
    }
   ...
   for (int i=0; i<12; i=i+1) {               // sort the cards
      for (int j=i+1; j<13; j=j+1)
         if (currentCard[i] > currentCard[j]) {
            temp = currentCard[i];
            currentCard[i] = currentCard[j];
            currentCard[j] = temp;
         }
   }
}
```

To include images of the cards, which are stored as *gif* files, JACIE provides a programmer with a variable of type `image`. The inclusion can easily be achieved in the declaration section of the client program. Below is the example code segment to show the declaration of some existing *gif* files that reside in the same directory as the client program. In the example, four different variables, `club`, `diamd`, `heart` and `spade`, are the arrays of type `image`. Each array consists of 13 elements.

```
declaration { ...
 image[13] club  = {"club2.gif","club3.gif",...,"clubK.gif","clubA.gif"};
 image[13] diamd = {"dia2.gif","dia3.gif",...,"diaK.gif","diaA.gif"};
 image[13] heart = {"heart2.gif","heart3.gif",...,"heartK.gif","heartA.gif"};
 image[13] spade = {"spade2.gif","spade3.gif",...,"spadeK.gif","spadeA.gif"};
}
```

The JACIE canvas channel displays the game according to grid specifications. Below is the example code segment that shows the grid for the location of all the card images, namely `nCard`, `eCard`, `sCard` and `wCard`, the players' title, and grid `playC` to display texts for labelling the users, 'N' (North), 'E' (East), 'S' (South) and 'W'

(West). This canvas specifications remain fixed throughout the program.

```
on canvas { ...
 draw string "BRIDGE GAME" at 140,20 size 18;
 draw grid nCard at 70,45 step 13,1 size 20,47 colour black width 1; // North
 draw grid eCard at 355,53 step 1,13 size 34,17 colour black width 1;// East
 draw grid sCard at 70,270 step 13,1 size 20,47 colour black width 1;// South
 draw grid wCard at 20,53 step 1,13 size 34,17 colour black width 1; // West

 draw grid playC at 150,110 step 3,3 size 34,47 colour white width 1; // player
 draw string grid playC "N" at 1,0 size 20;        // label for 'North'
 draw string grid playC "E" at 2,1 size 20;        // label for 'East'
 draw string grid playC "S" at 1,2 size 20;        // label for 'South'
 draw string grid playC "W" at 0,1 size 20;        // label for 'West'
 ...
}
```

Below is a JACIE method `griddap` that is responsible for identifying a card and transferring the appropriate image to the corresponding grid locations on the canvas.

```
  void griddap(grid2D card1,int x,int y) {
    int a=0;
    int b=0;
    for (int i=0; i<13; i=i+1) {  // ◀ A total of 13 cards for each player
     if (y == 0) a = i;
     else b = i;
     ...
     nom = currentCard[i] % 13;    // determine card number as the index of array for images
     if (currentCard[i]< 13) draw image grid card1 pcclub[nom] at a,b;
     else if (currentCard[i]< 26) draw image grid card1 pcdia[nom] at a,b;
     else if (currentCard[i]< 39) draw image grid card1 pcheart[nom] at a,b;
     else draw image grid card1 pcspade[nom] at a,b;
     ...
   }
 }
```

With the use of array `currentCard[]` to represent the card's value, this value is used as the reference to the index of the array of type `image`.

- **Bidding option** — After all users get the cards' information displayed, the game can begin the bidding process. Figure 7.2 shows the example game for the bidding process where all the players can see the bidding history.

  During the bidding process, several messages are displayed in the JACIE *message bar* (at the bottom of the screen in Figure 7.2) to instruct the current user. Grid cells can be chosen using a mouseclick that represents the bidding option. Below is the example code segment in the client program for the canvas settings for specifying the option buttons. There are two rows of buttons to do the bidding with the first row represented by `bid1` and `bid2` for bidding buttons in the second row. Grid `bidHis` is to display all the bidding histories.

```
on canvas { ...
 draw grid bid1 at 440,35 step 8,1 size 20,20 colour white width 1;
     ... // ▲ grid bid option button on the first row
 draw grid bid2 at 460,55 step 7,1 size 20,20 colour white width 1;
     ... // ▲ grid bid option button on the second row
 draw string " W        N         E         S" at 450,87 size 10;
 draw string "---------------------" at 450,97 size 10;
 draw grid bidHis at 450,105 step 11,15 size 12,15 colour white width 1;
```

```
    ... // ▲ grid bid history
}
```

The actual value of the bidding history is stored in a two-dimensional array for keeping track of the information and for display purposes. Below is the declaration of the variables that represent these values. The variable `bidHistory` stores all the necessary values of the bidding process while `bidHS1` and `bidHS2` are used to store the bidding information of the current user which is later copied into `bidHistory`.

```
client implementation { ...
    declaration { ...
        int[5][10] bidHistory = -1;
        int[5][10] bidHS1 = -1;
        int[5][10] bidHS2 = -1;
        ...
    }
}
```

The bid history is always displayed during the game and it is updated in every bidding made by each player. Below is the code segment that shows the printing of the bidding history during the bidding process where `uNumber` represents the user number and `rNumber` is the round number of the bidding.

```
...
for (int j=1; j<uNumber+1; j=j+1) {          // for the total users
   posX = (j-1) * 3;
   if (bidHistory[j][rNumber] == 0) {        // if user choose to 'pass' the bid
      draw string grid bidHis "Pass" at posX,rNumber size 12;
   }
   else {
      if (bidHistory[j][rNumber] == 1) { // if user is bidding
      ... // 'draw' the bidding history at the corresponding grid location
         if (bidHS1[j][rNumber]> -1 && bidHS1[j][rNumber]<7) {
            draw string grid bidHis points[bidHS1[j][rNumber]]
                        at posX,rNumber size 12;
         }
         if (bidHS2[j][rNumber]> -1) {
            if (bidHS2[j][rNumber]<4) {
               draw image grid bidHis cType[bidHS2[j][rNumber]]
                        at posX+1,rNumber;
            }
            else {
               draw string grid bidHis cT[bidHS2[j][rNumber]-4]
                        at posX+1,rNumber size 12;
            }
         }
      }
   }
}
```

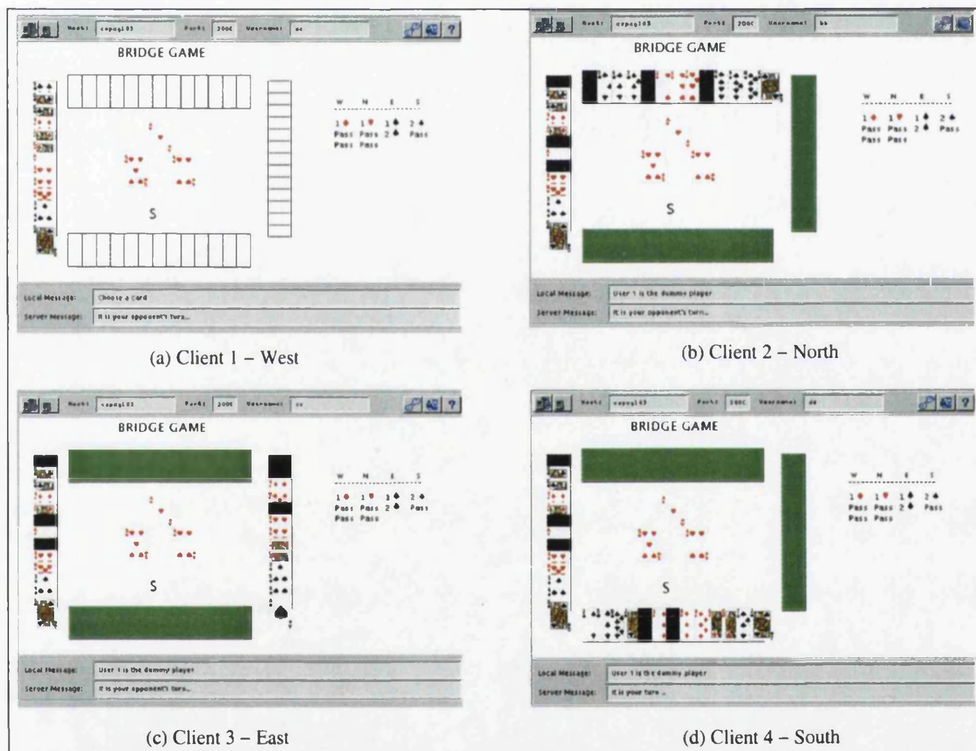
The bidding process ends when there are three consecutive 'passes' from three different users. Then, the game changes to the trick play mode.

- **Card selection during trick play —** During the trick play, a user can click on the selected card image to identify the card to play. Below is the code segment that shows how a player's selection on a grid is made by calling a JACIE method `getpoint()` to get the correct grid position. Before this call is made, the correct player number for the correct grid point is checked.

```
...
if (state==TRICK_PLAY && Card_to_Select==TRUE) {       // trick play mode
```

```
        if (cPlayer == 1) getpoint(2,1,0,1);              // determine current player
        else if (cPlayer == 2) getpoint(1,2,1,0);
        else if (cPlayer == 3) getpoint(0,1,2,1);
        else if (cPlayer == 4) getpoint(1,0,1,2);
        rCard = false;                                    // card is already selected
}
```

Once the correct grid cell is chosen, the cell is coloured with black to indicate the successful completion of the selection. Figure 7.3 shows a screenshot of the trick play for all four players. In this example, client 1 is the dummy player and all the others have the dummy's cards displayed on their sites. The dummy's partner, *i.e.*, client 3, can read the information and choose a card to play. Client 4 (South) has the current turn control. The player has two black squares on the cards deck, while other players have 3. This indicates that the game is on the third round in the trick play. After the thirteenth round, all the grid cells were selected and the game ends.



(a) Client 1 – West

(b) Client 2 – North

(c) Client 3 – East

(d) Client 4 – South

Figure 7.3: Bridge Game (The Trick Play).

The most significant difference between the bridge game and the noughts and crosses game and its variations is that the bridge game involves two interaction protocols, for the bidding and the trick play stages, respectively.

7.3.2 Dynamic Protocol Changes

In this implementation, the declaration of the protocols are not only in the *Configuration Section*, but also defined during the *Server On Session*. The game requires different types of protocol throughout the game session. The dynamic change of the protocols is described in Section 7.3.2.

The following subsections describe the methods used in managing the interaction protocol using several different protocol choices provided by JACIE II. Discussion will focus on the use of interaction protocols for the application without going into great detail about the complex rules of the game.

7.3.2.1 Method One: Protocol Round Robin Only

The game is treated as consisting of four individual players even though they are two pairs of partners, as no private communication is allowed to take place between partners. The round robin protocol is used for the entire game. The rules of the game are managed in a fully distributed manner, and there is little game-specific functionality implemented at the server side. Once the bidding mode has finished, the standard JACIE message channel is used for the winning bidder to issue instructions to the dummy partner, in view of all the other players. The dummy partner simply follows the instructions when it is his turn. This mechanism is very similar to that in a face-to-face bridge game.

```
...
configuration { ...
  number of users 4;
  protocol roundrobin;
}
client implementation { ...
  on TURN { ...
    if (state == BIDDING) {          // ◄ this is a local state
      ... // select an appropriate action on canvas
      ... // run an algorithm to determine if the bidding process finishes
      if (condition == BIDDING_END) {
        ... // run an algorithm to determine the dummy and the first player
        if (firstPlayer == ME) {   // is the bid winner
          player_counter = 0;      // initialise the player counter for a trick
          trick_counter = 0;       // initialise the trick counter for a game
          state = TRICK_PLAY;      // start a new trick, play now
        } else if (player_counter  > 0)
          state = TRICK_PLAY;      // a new trick has started, play now
        else {
          state = TRICK_PLAY;      // play next time when 'on TURN'
          turn pass;
        }
      }
    }
    if (state == TRICK_END) {
      ... // run an algorithm to determine the first player (based on the last trick)
      if (firstPlayer == ME) {
        player_counter = 0;
        trick_counter = tricker_counter + 1;
        state = TRICK_PLAY;        // start a new trick, play now
      } else if (player_counter < 4)
        state = TRICK_PLAY;        // a new trick has started, play now
      else {
```

```
        state = TRICK_PLAY;          // play next time when 'on TURN'
        turn pass;
      }
    }
    if (state == TRICK_PLAY) {
      if (dummy == ME) {
        ... // if it is the first trick, show all the cards
        ... // send a message to remind the bidder to issue an instruction
        ... // follow the bidder's instruction in the message window
        ... // play the card selected by the bidder
      } else {
        ... // select one card from my deck and play
      }
      player_counter = player_counter + 1;
      state = TRICK_END; // ◄ the end of a trick for the player concerned
    }
    turn pass;
  }
}
server implementation { ... }
...
```

Since the above algorithm involves a distributed decision process, most variables are local and every player has the same copy of code. However, some variables must be global and shared, for example, `player_counter` and `trick_counter`.

7.3.2.2 Method Two: Protocol Round Robin and Protocol Master User

The bidding process follows the same procedure as in Method One, except that the turn during the trick play is determined by the server. The interaction protocol is initially set to *round robin*, and after the bidding process finishes, it is changed to *master server userdefined.* The server program determines and sets the client turn based on a user-defined code segment for turn management. Thus the mechanism for managing the trick play mode uses a centralised approach, where almost all game specific functionality is implemented at the server side.

```
...
configuration { ...
  number of users 4;
  protocol roundrobin;
}
client implementation { ... }
server implementation { ...
  if (state == BIDDING) {
    ... // run an algorithm to determine if the bidding process finishes
    if (condition == BIDDING_END) {
      state = TRICK_PLAY;
      player_counter = 0;     // initialise the player counter for a trick
      trick_counter = 0;      // initialise the trick counter for a game
      protocol master server userdefined; // ◄ protocol change
      ... // determine the dummy and the first player based on the bidding sequence
      turn set client firstPlayer;
      player_counter = player_counter + 1;
    }
  } else if (state == TRICK_PLAY) { ...
    if (player_counter == 4) {
      player_counter = 0;
      trick_counter =  trick_counter + 1;
```

```
        ... // determine the player who wins the trick
        turn set client winnerPlayer;
      } else { ...
        ... // determine the order for the turn in the trick
        turn set client nextPlayer;
      }
      player_counter =  player_counter + 1;
    } ...
}
```

This example demonstrates a newly introduced feature of JACIE, namely dynamic protocol change within a JACIE session. This is particularly useful for implementing some complex interaction protocols. As long as such a protocol can be decomposed into a set of simpler protocols in a temporal order, one can use dynamic protocol change to facilitate the transition from one protocol to another. In this example, the transition involves a change from *round robin* to *master server userdefined*, and from four active players in the bidding stage to three in the trick play stage.

7.3.2.3 Method Three: Protocol Round Robin and Protocol Group

In this implementation, the bridge game is viewed as a group game. There are two groups (two players in each group) with no communication allowed between group members. In the bidding process governed by *protocol group round robin*, the flow of the game is exactly the same as the *protocol round robin* for four individual players. For the trick play, the interaction protocol for the winning bidder group changes to *protocol group master*, while the other group's protocol remains unchanged. As described in Chapter 5, *protocol group master* allows only one player of the group to be active and thus facilitates the change in the total number of active players in the game.

```
...
configuration {
  number of users 4;
  protocol roundrobin;          // protocol between group
  number of groups 2;
  protocol group roundrobin;    // protocol within a group, starts with first player
  ...
}
client implementation { ... }
server implementation { ...
  if (state == BIDDING) { ...
    ... // run an algorithm to determine if the bidding process finishes
    if (condition == BIDDING_END) {
      state = TRICK_PLAY;
      player_counter = 0;
      trick_counter = 0;
      ... // run an algorithm to determine the bidder group and the opponent group,
          // the bidder player (group master), the dummy, the first player
      protocol group groupnumber groupBidder master playerBidder;
      ... // ▲ change the protocol of the bidder group, and set the master
      turn set client groupOpnt;     // turn starts with the opponent group
      turn set group groupOpnt firstPlayer;
      player_counter = player_counter + 1;
    }
  } else if (state == TRICK_PLAY) { ...
    if (player_counter == 4) {
      player_counter = 0;
```

```
      trick_counter =  trick_counter + 1;
      ... // determine the player who wins the trick and the associated group
      turn set client winnerGroup;
      turn set group winnerGroup firstPlayer;
    }
    player_counter = player_counter + 1;
  } ...
}
```

Like the `turn set client` statement that sets one of clients to have a turn and in this example is the `winnerGroup` during the `TRICK_PLAY` state, `turn set group` statement allows the setting of one of the group members in the `winnerGroup` to become the first to play in the round.

7.3.2.4 Summary on the Bridge Game Interaction Protocol Implementations

As the bridge game requires changes in the turn protocol and the number of players during the game, we have implemented three possible different ways of interaction management provided by JACIE. Method One uses distributed management on the game rules supported by some algorithms and some involvements of global variables. JACIE facilitates this approach with the use of its standard message channel. The other two methods, Two and Three, provide the centralised functionality control based on the selected interaction protocols. Method Two requires the appropriate algorithms before the turn is assigned to the specific user, while method Three needs less program coding since it has to determine the turn order of only one of the groups that uses the group protocol roundrobin. The other group only has one active user. In general, method Three can be considered the best, as compared to method One that requires many message passing activities and method Two that needs quite a complicated algorithm.

7.4 E-learning on Simulation of Network Trouble Shooting

In collaborative applications, e-learning is one of the popular activities where people share information through *virtual classrooms* and *communities*. There are different methods used to help people share such information. Common ways to collaborate are shared whiteboard, chatting and video conferencing. Much research concentrates on the strategies and tools [77, 243] for these environments. Both the strategies and learning tools consist of the technology on improving the learning process such as the system's presentation [87] and presented materials [243]. In addition, electronic content of material can determine learning efficiency. Software systems and devices to support the learning systems are still ongoing research topics.

In this section we describe an application on data sharing and interest filtering implementations. In the variation of the noughts and crosses game namely Secret Switch, only the data sharing mechanism can be applied without the filtering mechanism since only two players are involved. The shared variable is the game board where all the players must have the read and write access, while in the bridge game, the variable sharing only occurs from the middle to the end of the game. According to the game rules, all players can only have read access

to one of player's cards for the game strategy except the bidding declarer has write permission. The filtering mechanism based on interest is also not appropriate for the bridge game because all players must share the dummy's card for the game tactic in the trick play. Since most work focuses on interaction management and language improvement, no data sharing constructs are implemented in this game, although the data sharing issue is also significant. So, in order to complete the testing of the interest management mechanisms, an e-learning network trouble shooting program is implemented and described in this section.

This e-learning environment consists of three on-line users who are assigned three different rooms, Room 1, Room 2 and Room 3. Each room has a network of computers and peripherals. The peripherals can be workstations, printers, hubs, a gateway, servers or network

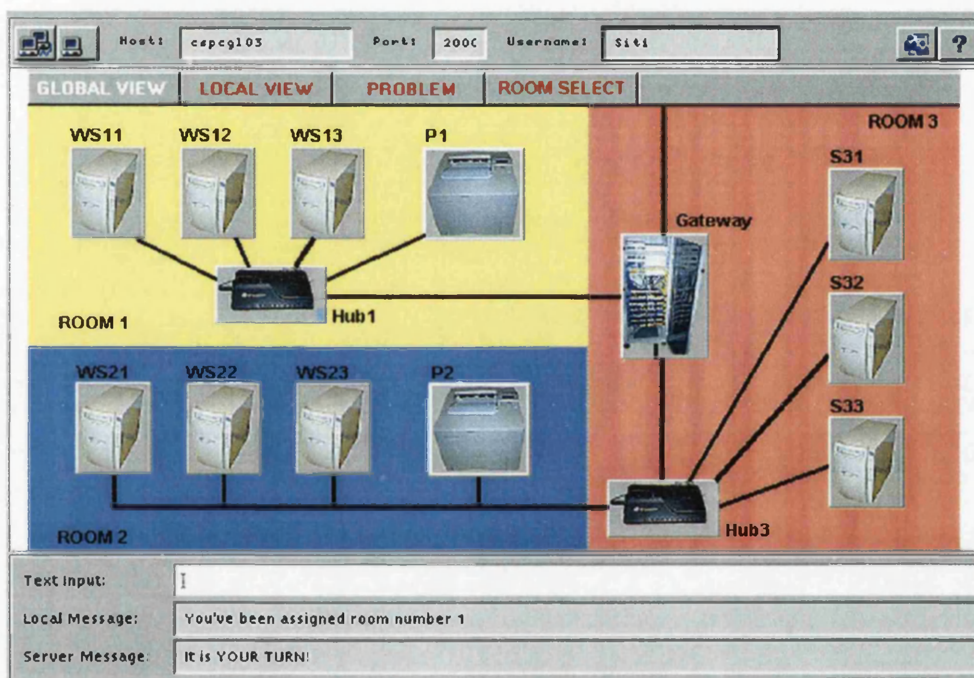
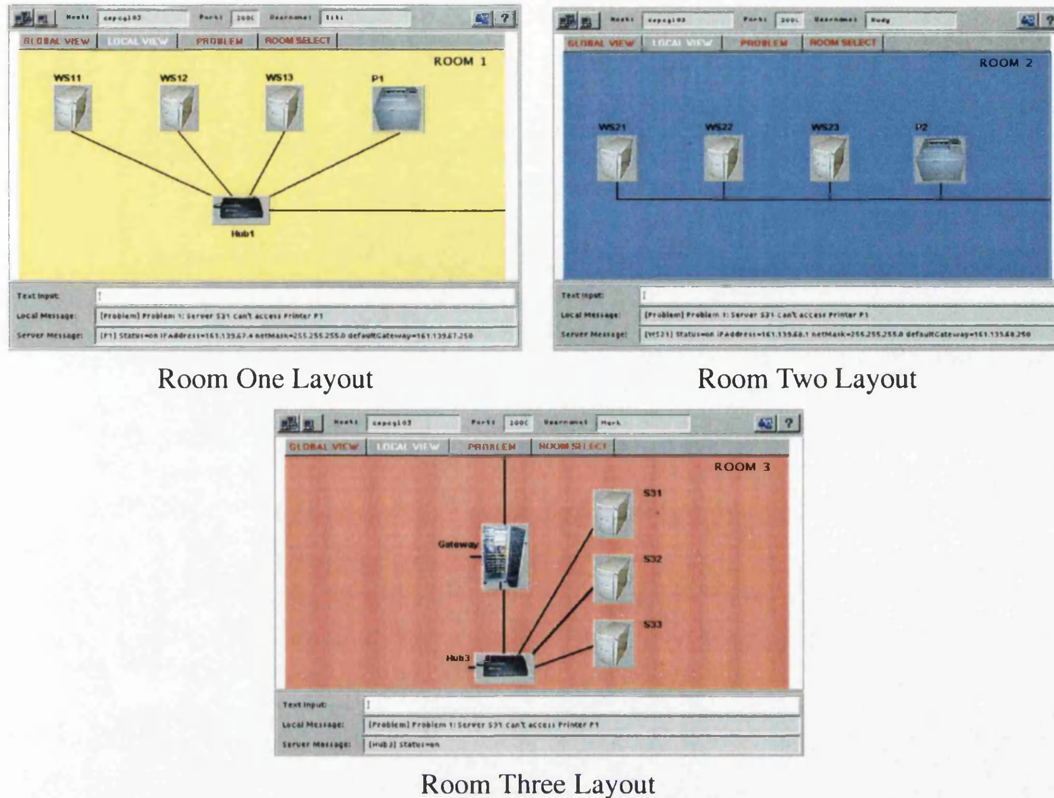


Figure 7.4: Overall Network Settings.

cabling. The network in all the rooms are inter-connected. Figure 7.4 shows the overall layout of the total network, which is the global view seen by the user in Room 1. At the start of a session, all users have the default view of the global layout. The main objectives of this application is for users to discover the cause of a network problem assigned by the server. In the process of trouble shooting, some of the network components must be checked for their status or properties before the problem can be detected, diagnosed and solved.

As described before in Section 7.1 that this application was already implemented before using JACIE I, but a different approach had been used and the user's local view was not sharable. The local view of every user is shown in Figure 7.5. A user can check the status of any device in the room by clicking on the specified device. Once the cause of the problem is known, the user can change the device property to solve the given problem by entering the command in the *Text Input bar* at the bottom of the user interface. This example requires



Room One Layout

Room Two Layout

Room Three Layout

Figure 7.5: Layout of the Individual Room.

only the contention protocol for the turn control for all the users to involve in the chatting process.

The local view of each room, Figure 7.5, illustrates the break down of the whole network set up. Room 1 consists of three workstations, a printer, a hub and five cables that connect the hub to all the devices. Room 2 shows only three workstations, a printer and a cable, while room 3 has a hub, a gateway, three servers and four cables that connect the hub to all the devices in the room. This figure also shows that all the users, 1, 2 and 3, have selected to view one of the network components in their assigned rooms. The *JACIE Local Message bar*, at the bottom of the canvas, displays the problem given by the server, and the *JACIE Server Message bar* prints the property of the selected component. For example, user 1 is clicking on the printer P1 in Room 1 which has status 'on', IP address is 161.139.67.4, netmask is 255.255.255.0 and default gateway is 161.139.67.250. User 2 is currently choosing workstation number 2 (WS21) with status also 'on', IP address 161.139.68.1, netmask is 255.255.255.0 and default gateway 161.139.68.250. User 3 is clicking on the hub in Room 3 and the information status is 'on'.

Turn control protocol is insignificant if the provided interest management feature is implemented. The chat channel is not necessarily the only way to collaborate. Therefore, we can implement the simulation on network trouble shooting with the data sharing mechanisms that allows all the users not only to view the assigned rooms as their local views, but also to

view any of other users' rooms as the local views. However, a user can only work on one room at a time.

In this application, there are four selection buttons at the top of the canvas. The first button is for viewing the global network layout, the second button is the local view selector, third, is the button to display the network problem on the *JACIE Local Message bar* and the last button is the global network layout with the room selection buttons. The last button is to allow every user to select and view other users' assigned rooms. Notice that the button is inactive or changed in colour once it is selected. For example, Figure 7.4, that shows the global view, has the `Global view` button inactive, Figure 7.5 shows the change in the `Local View` button and Figure 7.6 has the button `Room Select` inactive. Below is the example of code segment to show the display of the selection buttons on the top of canvas so that a user can choose to have the global view of the network, local view of a room, get information on the description of the trouble shooting problem to be displayed on the *message bar* or a button to view the options of room to be selected. Here, the button for the global view is currently 'on' while the other buttons are 'off'.

```
on canvas {
  draw grid ViewIcons at 10,0 step 5,1 size 100,25 colour black width 1;
  draw image grid ViewIcons globalViewOn at 0,0;    // global view button selection
  draw image grid ViewIcons localViewOff at 1,0;    // local view button selection
  draw image grid ViewIcons problemIcon at 2,0;     // problem description button selection
  draw image grid ViewIcons roomOn at 3,0;          // room selection button selection
}
```

A user can let others have read or write or both read and write access to the selected devices in his/her room. The interest management mechanism facilitates this feature. With the introduced data sharing, the user can choose to work on the assigned room or another room that he/she is allowed to have access. Figure 7.6 shows how each client can select any room to work on.

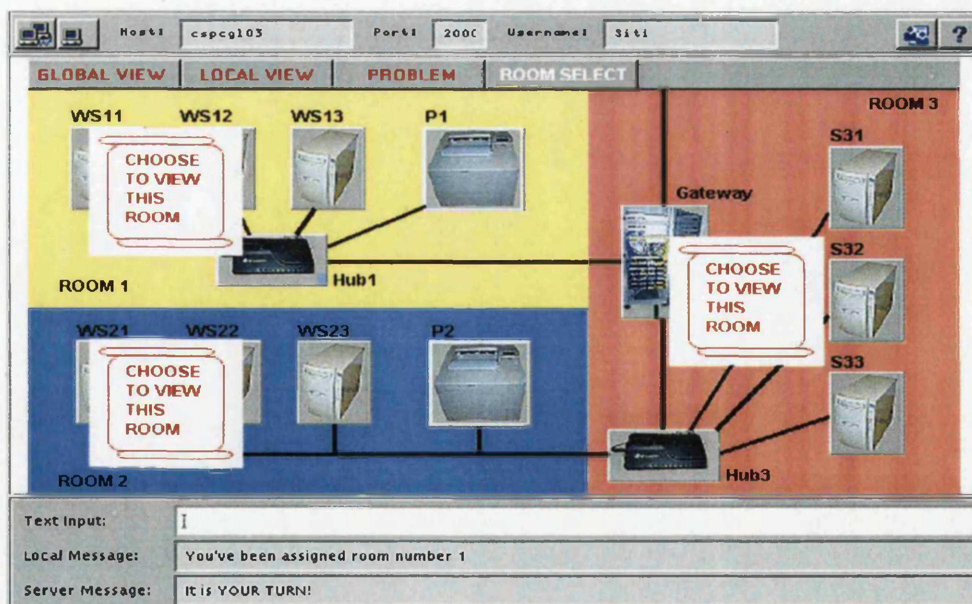


Figure 7.6: Room Selection.

In this figure, each room has the text, 'CHOOSE TO VIEW THIS ROOM'. By double clicking this text, the user will see the local view of the selected room. Viewing the selected room does not necessarily mean that the user has the control of the room, the control is dependent on the permission given by the original owner of the room. The code segment below gives the grid specifications on the button that presents the actual selection to the selected room.

```
...
draw grid roomC1 at 50,55 step 1,1 size 100,100;     // grid for button to select Room 1
draw grid roomC2 at 50,220 step 1,1 size 100,100;    // grid for button to select Room 2
draw grid roomC3 at 430,140 step 1,1 size 100,100;   // grid for button to select Room 3
draw image grid roomC1 chooseR at 0,0;               // Copy the image to the grid
draw image grid roomC2 chooseR at 0,0;
draw image grid roomC3 chooseR at 0,0;
...
on MOUSECLICK { ...
  if (currentView == 3) { ...                        // currently is on 'room select' button
     selectView=true;                                // a selection has been made
     if (GETGRID==roomC1) selectR=1;                 // select Room 1, or
     else if (GETGRID==roomC2) selectR=2;            // select Room 2, or
     else if (GETGRID==roomC3) selectR=3;            // select Room 3
     else selectView=false;
     refresh;
  } ...
}
```

With the data sharing feature, every network component is represented as a variable. In this application, not all components are shared. Therefore, only the shared component is referred to as a shared variable. All shared variables are declared as follows in the client program.

```
...
// devices and cables in Room 1
...
shared string h11 = "";         // represents Hub 1 in Room 1
shared string c14 = "";         // represents Cable 14
shared string p11 = "";         // represents Printer 1 in Room 1
// devices and cables in Room 2
...
shared string p21 = "";         // represents Printer 2 in Room 2
shared string c21 = "";         // represents Cable 21
// devices and cables in Room 3
...
shared string h31 = "";         // represents Hub 3 in Room 3
shared string c31 = "";         // represents Cable 31
shared string s31 = "";         // represents Server 31
...
```

In the server program, the above declaration of the shared variables are automatically generated by the declaration. The server can have full control over those variables. In the server program, each network component and its respective properties are kept in arrays. The information is kept in this way to ease the process of updating the specific properties. The following code segment shows part of the array declaration.

```
...
shared string[31] netDevice              // keeps device information
   = {"WS11","WS12","WS13","P1","Hub1",
      "Cable11","Cable12","Cable13","Cable14","Cable15",
```

```
      "WS21","WS22","WS23","P2",
      ...
      "Cable31","Cable32","Cable33","Cable34","Gateway"};

shared string[31][4] netConfig         // keeps status information
   = {{"on","161.139.67.1","255.255.255.0","161.139.67.250"},
      {"on","161.139.67.2","255.255.255.0","161.139.67.250"},
      {"on","161.139.67.3","255.255.255.0","161.139.67.250"},
      {"on","161.139.67.4","255.255.255.0","161.139.67.250"},
      {"on","na","na","na"},
      {"connected","na","na","na"},
      {"connected","na","na","na"},
      {"connected","na","na","na"},
       ...}
```

At the start of a session, the server initialises all the shared variables using the *set* statement by assigning the value from the `netConfig` array into the shared variable. The boolean variables, `firstSet`, `secondSet`, `thirdSet` and `startNow`, are used in the program to make sure that the initialisation of values are executed only once.

```
...
if (!firstSet && USERNUMBER==1 && startNow)        // user 1 is also the owner
 firstSet = true;
 set p11 =netConfig[3][0]+", IPAddress="+netConfig[3][1]+", defaultGateway="+
          netConfig[3][2]+", netMask="+netConfig[3][3];
 set h11 =netConfig[4][0]+", IPAddress="+netConfig[4][1]+", defaultGateway="+
          netConfig[4][2]+", netMask="+netConfig[4][3];
 set c14 =netConfig[8][0]+", IPAddress="+netConfig[8][1]+", defaultGateway="+
          netConfig[8][2]+", netMask="+netConfig[8][3];

if (!secondSet && USERNUMBER==2 && startNow)       // user 2 is also the owner
 secondSet = true;
 set p21 =netConfig[13][0]+", IPAddress="+netConfig[13][1]+", defaultGateway="+
          netConfig[13][2]+", netMask="+netConfig[13][3];
 set c21 =netConfig[14][0]+", IPAddress="+netConfig[14][1]+", defaultGateway="+
          netConfig[14][2]+", netMask="+netConfig[14][3];

if (!thirdSet && USERNUMBER==3 && startNow)        // user 3 is also the owner
 thirdSet = true;
 set s31 =netConfig[22][0]+", IPAddress="+netConfig[22][1]+", defaultGateway="+
          netConfig[22][2]+", netMask="+netConfig[22][3];
 set h31 =netConfig[25][0]+", IPAddress="+netConfig[25][1]+", defaultGateway="+
          netConfig[25][2]+", netMask="+netConfig[25][3];
 set c31 =netConfig[26][0]+", IPAddress="+netConfig[26][1]+", defaultGateway="+
          netConfig[26][2]+", netMask="+netConfig[26][3];

...
```

The following code segment is part of client program that shows how the permissions are set for shared variables that represent the devices in one's room. The checking condition on the `USERNUMBER` guarantees that only the owner has the right to determine the access permission. The flag `firstSet` ensures that the permission setting is performed only once during the whole session. In the example, user 1 is assigning three of the network components, cable number 14, a printer and a hub in his/her assigned room, Room 1, to be sharable. Cable number 14 can be owned and read, as well as write its property with a password. The only acceptable password value is `admin`. Printer and hub are given read access without write access. User in room 2 allows the printer in the room to be owned by all the users and cable number 21 to be read by user 3. User 3 allows server number 31 and

hub in room 3 to be owned, read and written, while cable number 31 can only be read by user 1.

```
if (USERNUMBER == 1) {  // ◄ Owner is user 1
  if (!firstSet) {
    firstSet = true;
    use c14 by all to own to read to write with password "admin";
    use p11 by 2 not to own to read;
    use h11 by 3 to own to read;
    // ▲ Use permission on three of the owner's variables
  }
...
if (USERNUMBER == 2) {  // ◄ Owner is user 2
  if (!firstSet) {
    firstSet = true;
    use p21 by all to own to read to write;
    use c21 by 3 not to own to read;
    // ▲ Use permission on two of the owner's variables
  }
...
if (USERNUMBER == 3) {  // ◄ Owner is user 3
  if (!firstSet) {
    firstSet = true;
    use s31 by all to own to read to write;
    use c31 by 1 not to own to read;
    use h31 by all to own to read to write;
    // ▲ Use permission on three of the owner's variables
  }
 ...
```

With the permission given by the owner, the selected users are allowed to read the information regarding the selected device such as the status, IP address, netmask and the gateway. The following code segment shows how a user sets the interest before the `read` operation in order to access the information. The *print* statement causes the specified string to be printed in the *JACIE Local Message bar*. It will display the name of the selected network component. The example below illustrates the `read` operation on variable `p11` that represents the printer in Room 1. The user sets the interest value to `1.0` to guarantee the read access. With the *set* statement, the `read` operation is invoked by assigning the property of printer 1 to a variable `displayI`. Upon checking the value of `displayI`, it is determined whether the `read` operation is successful or not. If the read operation is not granted, the appropriate message is displayed in the *JACIE message bar*, and in this example, the message is 'No access to the current device'. With the statement `print servermessage`, the status of the selected sharable device is displayed at the bottom of the screen in the *message bar*.

```
print "device = "+deviceName;
if (deviceName == "Printer1") {
  interest set p11 1.0;         // user sets interest to access shared variable
  set displayI = p11;           // read access statement
  if (displayI!="")             // ◄ check access condition
     print servermessage "[P11]"+displayI;  // display the shared data
  else
     print servermessage "No access to the current device";
  ...
```

Figure 7.7 is a screenshot of all the users who currently are viewing room 1 at the same time. There are three screens, user 2 is at the top, user 1 who is also the owner is in the middle and user 3 is at the bottom of the figure. User 2 is viewing the information on printer 1 and user

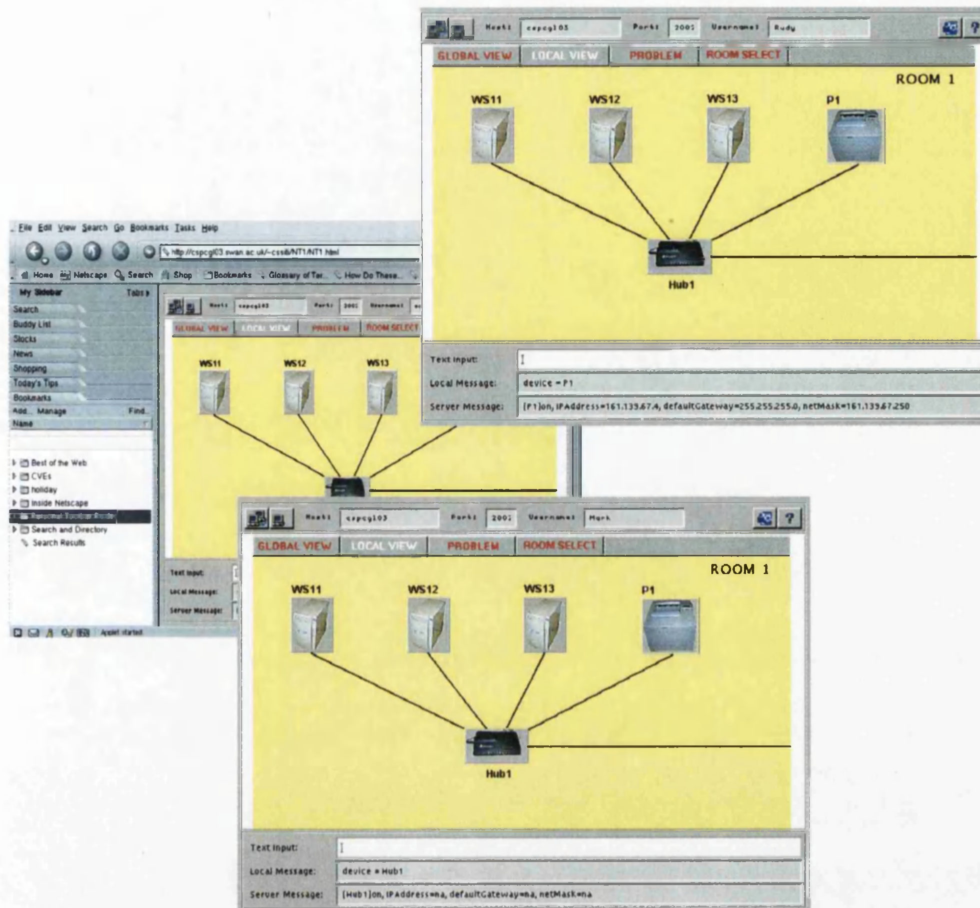


Figure 7.7: Read Operation on Devices in Room 1.

3 is viewing the information on the hub. All the users have the same button selection in the local view but each of them has different control over the network components in the room. User 2 can only access cable number 14 and printer for the `read` operation, user 3 can only access cable number 14 and the hub while user 1 has full control over all the components in this room.

For some variables that have write permission, there are a few selection buttons to take an appropriate action on changing the item's property. Figure 7.8 shows a screenshot of user 1 who is viewing room 3. The user who is allowed to have read/write access to the server number 31, is selecting the device for the `read` operation. The *JACIE message bar* displays the device status, and the user is ready to do any write operation if necessary. There are five option buttons appearing at the left of the network components to perform the selected action. The user can choose not to change the device's property, change the device status (to switch on or off and connected or disconnected for a cable), change the IP address, netmask or gateway by entering the text value at the *Text Input bar*. The user is also free to ignore these buttons and continue to view information on other permissible network components. If a cable is selected, any attempt to change IP address, netmask or gateway will be denied since it is not applicable.

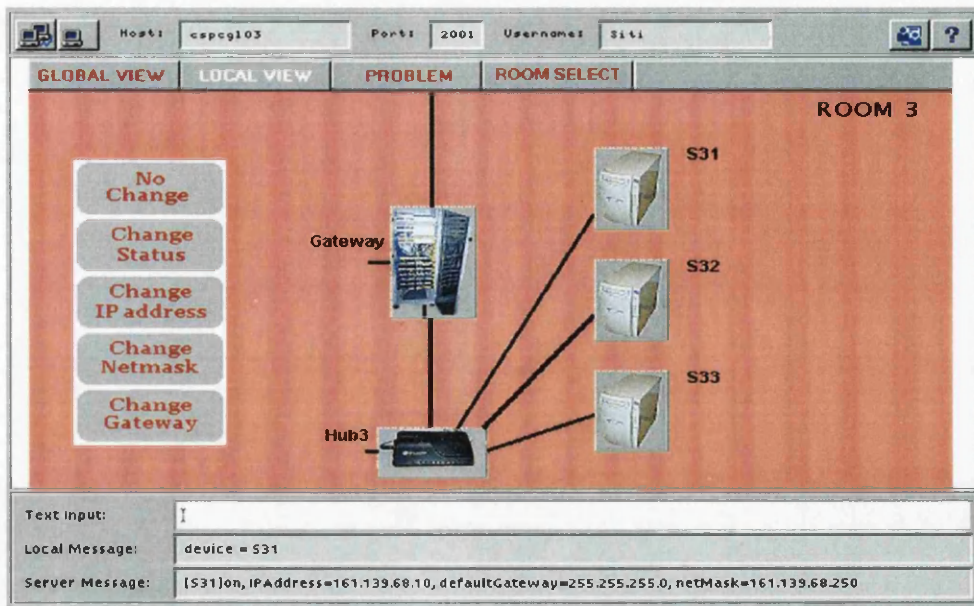
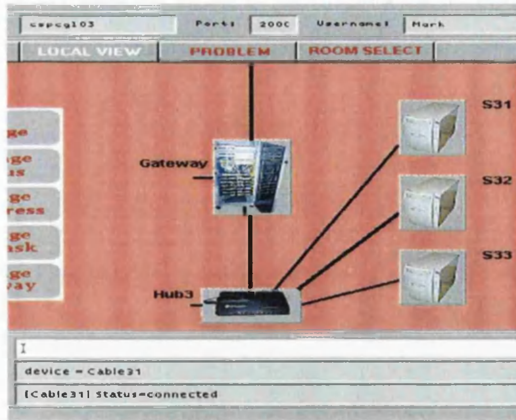


Figure 7.8: Ready To Do a Write Operation on a Device in Room 3.

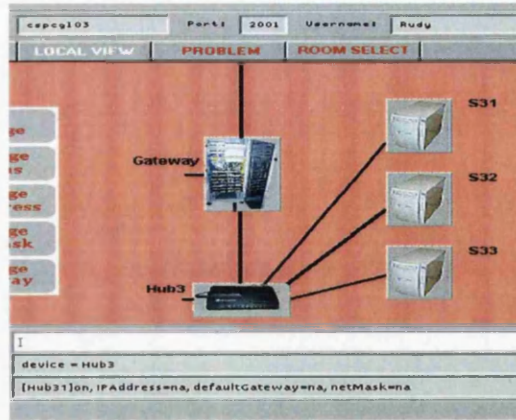
The problem to solve in this e-learning example is to find out the reason why Server 31 in Room 3 cannot access Printer P1 in Room 1. Figure 7.9 shows one of the possible options tried by all the users to find out the problem. In this example the turn control protocol is set to roundrobin to make sure that any change in the network component is mutually exclusive and a user can work on only one component in a turn. The order of the figure is not according to the user turn, but rather the connection of the specified devices starting from server S31 in Room 3. User 1 checks the status of server S31 in Room 3 (as shown in Figure 7.8), which is read/write by all users. Server S31 is on. User 3 clicks on cable 31 in Room 3 and finds out that the cable is connected. User 2 who has read/write access to Hub 3 in Room 3 sees that the hub's status is on. User 1 who has not given any access on Cable 15 in Room 1 to anyone else has the check on the status of this cable and it shows that it is connected. User 1 also clicks on Hub 1 and it is on. User 3 clicks on cable 14 in Room 1 and finds out that the cable is not connected. User 2 checks the printer status in Room 1 and it shows that the printer is on.

This shows that cable 14, selected by user 3 is the cause of the network problem. Since the device is read/write by all the users, user 3 can change its status from disconnected to connected. Figure 7.10 shows the screenshot of the action taken by user 3. When the action button 'Change Status' is clicked, the new device's status is sent to the server using the *set* statement. This is automatically verified by the server and the new status value is sent to all the users. Therefore, when any user who selects cable 14 afterwards, will find that the status is connected. The *Local Message bar* displays the process, while the *Server Message bar* shows the new status of the cable.

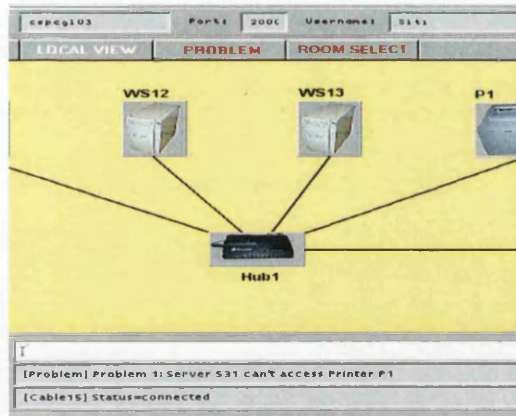
In the JACIE program, variable `c14` represents the cable14 device. Its value is set using



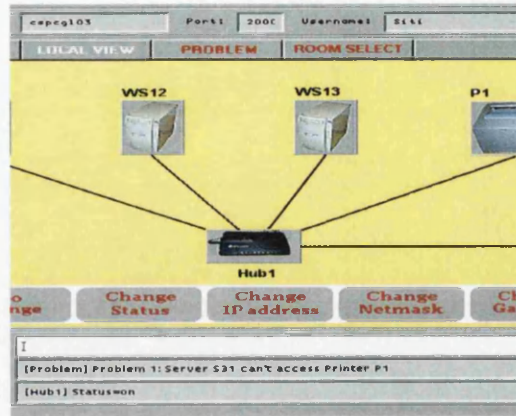
User 3 selects Cable31



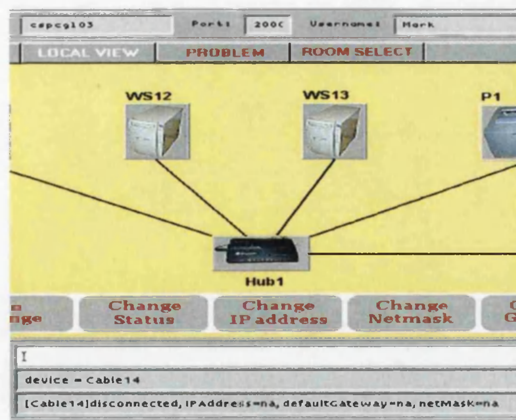
User 2 selects Hub3



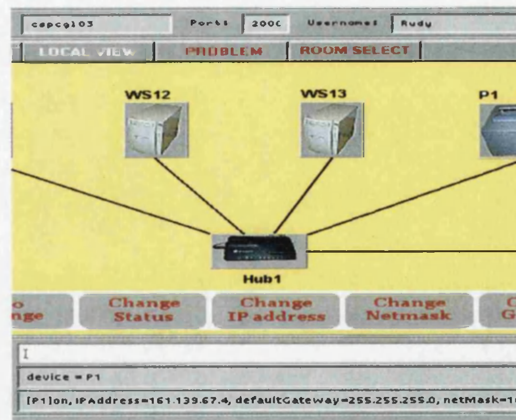
User 1 selects Cable15



User 1 selects Hub1



User 3 selects Cable14



User 2 selects device P1

Figure 7.9: Checking on Network Components.

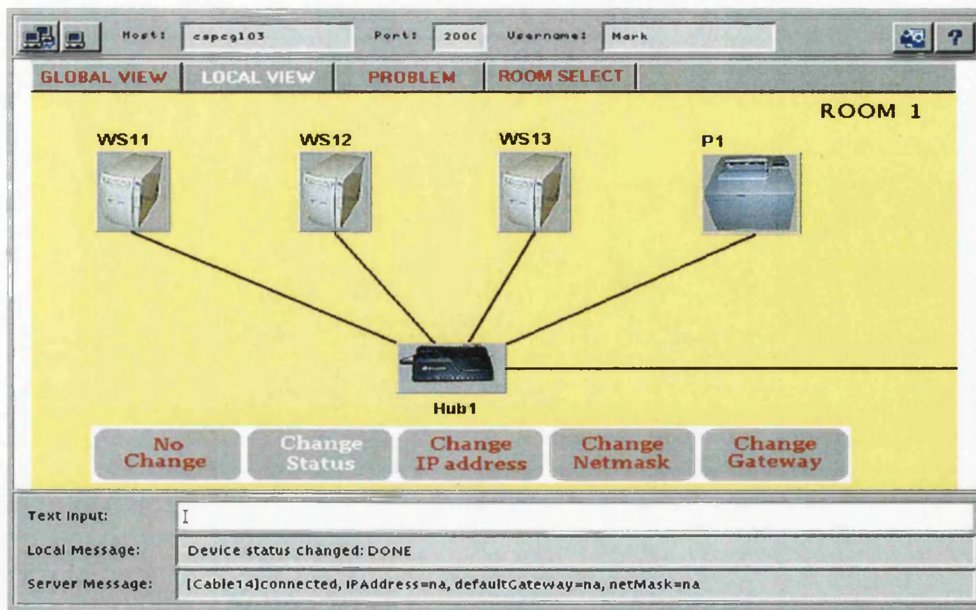


Figure 7.10: Detecting the Problem and Solve.

the new language construct. Below is the code segment that illustrates the use of the *set* statement. When the user is instructed to enter the device status, he/she must enter the value using *Text Input bar* by typing the word 'connected'. The variable is updated by the server and the server sends the new c14 value to all the users since all the users are the owners of the variable.

```
if (writeSelect) {                               // ready to perform 'Write' operation
   if (selectAction==noChange) { ...
      print "Do nothing to the selected device: DONE";
   }
   else if (selectAction==changeStatus) {        // Select button Change Status
      print "To change status, enter text";
      input change;                              // enter text in the textInput bar
      if (deviceName = "Cable14")
         set c14 = "Status="+change;             // ◄ change device status
      else if (deviceName = "P1");
         set p11 = "Status="+change;
      ...
      print "Device status changed: DONE";
   }
   else if (selectAction==changeIP) {            // Select button Change IP Address
      if (deviceType(deviceName)=="workstation" ||
           deviceType(deviceName)=="server" ||
           deviceType(deviceName)=="printer") { // only applicable on specific devices
         print "To change IP Address, enter text";
         input change;
         if (deviceName = "P1")
            set p11 = "IPAddress="+change;       // change device IP address
         else if (deviceName = "P2");
            set p21 = "IPAddress="+change;
         ...
         print "IP Address changed: DONE";
      }
      else print "Selection is not applicable";
```

```
    }
    else if (selectAction==changeNetM) {...
    } ...
}
```

At the server, when any of the devices' properties are changed, the 'reachability' of the associated devices that have connection to the current updated information is modified accordingly. The use of array variable `netConfig` is associated with another array called `reachability`. Below is part of the JACIE server program that handles the update of the network information.

```
if (problemNumber==1) {
    netConfig[8][0] = "disconnected";
    for (j=0; j<31; j=j+1)
      if (j!=3) {
        reachability[3][j] = 0;          // none can reach P1
        reachability[j][3] = 0;
      }
      problem = "Problem 1: Server S31 can't access Printer P1";
{
...
for (k=22; k<=24; k=k+1)                  // reassign reachability for S31, S32, S33
    if (reachability[25][k]==1)
        for (j=0; j<31; j=j+1)
          if (reachability[25][j]==1) {
              reachability[k][j] = 1;
              reachability[j][k] = 1;
          }
...
```

The example shows that the reachability factors of the devices associated with printer P1 in room 1 are set to zero at the start of session. This means that printer P1 is not accessible by all the devices that should have links to it. Later in the example, the code shows how the reachability of all the servers are set to 1 to indicate that connections between the specified devices are achieved.

7.5 Performance Analysis

Several experiments have been undertaken to find out the significance of this research. While Section 6.7 discusses interest management, this section focuses on the overall JACIE interaction management and delay problem in the presented example programs (e.g., the noughts and crosses games). Here, several results are presented to support the discussion.

7.5.1 JACIE vs. Java Translated Program

JACIE is translated into Java and then compiled to byte code for execution under the Java virtual machine. A comparison between JACIE and its translated Java programs is discussed to discover the differences in the amount of coding and compilation time. Using the Linux command `wc`, the byte, word and line counts for a program can be determined. Therefore, the above measures have been calculated to compare several JACIE programs and their

corresponding Java translated programs. The results are shown in Table 7.1. There are five JACIE programs selected for this purpose. `Hello.jacie` is a simple program that consists of one client connected to server to send a 'Hello' message and then the server sends a reply with the "OK: Hello" message to acknowledge the reception. `Vicious.jacie` and `Dictator.jacie` are the variations of the noughts and crosses games, vicious battle (Appendix A.5) and gentlemen's battle (Appendix A.6). `Bridge.jacie` is the bridge game example discussed in Section 7.3 and `Network.jacie` is the e-learning application example in Section 7.4.

	FILES		**BYTE**		**WORD**		**LINE**
1.	Hello.jacie		927		112		61
1(a)	Client Programs	24,570		1,974		816	
1(b)	Server Programs	12,244		1,046		417	
(a)+(b)	Total in Java		36,814		3,020		1,233
2.	Vicious.jacie		8,535		1,005		296
2(a)	Client Programs	39,696		3,230		1,253	
2(b)	Server Programs	26,522		2,200		763	
(a)+(b)	Total in Java		66,218		5,430		2,016
3.	Dictator.jacie		8,630		1,012		299
3(a)	Client Programs	40,413		3,262		1,265	
3(b)	Server Programs	28,221		2,303		807	
(a)+(b)	Total in Java		68,634		5,565		2,072
4.	Bridge.jacie		31,656		3,737		1,002
4(a)	Client Programs	81,929		6,013		2,301	
4(b)	Server Programs	36,134		2,738		973	
(a)+(b)	Total in Java		118,063		8,751		3,274
5.	Network.jacie		67,525		5,801		1,514
5(a)	Client Programs	110,927		7,170		2,828	
5(b)	Server Programs	92,134		7,073		1,962	
(a)+(b)	Total in Java		203,061		14,243		4,790

Table 7.1: Codes Length Comparison on JACIE and Its Java Translated Programs.

Referring to Table 7.1, the smaller size JACIE programs, produce Java programs with larger percentage size. For example, the simplest program, `Hello`, yields an equivalent Java program about 30 times larger, whereas the longest program, e-learning simulation (`Network.jacie`) produces Java codes about 2.5 to 3 times larger. This is due to the fact that a lot of Java code is required to prepare the graphical user interface for the output. Providing such interface takes about 1200 number of words for the client programs. On top of that, in the translation process, there are several Java files that contain the basic required codes. These files take about a minimun of 1000 number of words for the server programs and 300 words for the client programs.

We also look into the processing time of these five files. It includes the compilation time (from the start to the end) and the CPU time that they take during the compilation. This

calculation is made based on the Linux `time` command. Table 7.2 shows the data in seconds (sec). JACIE is designed to provide simpler codes in writing a networked collaborative application. Its compilation must go through two phases of translation, JACIE to Java, and then from Java to its virtual machine code. This table shows that the time to translate JACIE to Java does not significantly increase the overall compilation time. For example, it takes about 4 seconds for its 1,514 lines of program to complete a compilation. Within this time, the actual CPU time for the processing is 2.42 seconds.

	FILES		**COMPILE TIME (sec)**		**CPU TIME (sec)**
1.	Hello.jacie		0.76		0.33
1(a)	Client Programs	0.98		0.65	
1(b)	Server Programs	0.8		0.55	
(a)+(b)	Total in Java		1.78		1.20
2.	Vicious.jacie		0.83		0.42
2(a)	Client Programs	1.12		0.65	
2(b)	Server Programs	0.8		0.58	
(a)+(b)	Total in Java		1.92		1.23
3.	Dictator.jacie		0.83		0.41
3(a)	Client Programs	1.0		0.71	
3(b)	Server Programs	0.82		0.58	
(a)+(b)	Total in Java		1.82		1.29
4.	Bridge.jacie		1.78		0.66
4(a)	Client Programs	1.12		0.76	
4(b)	Server Programs	0.85		0.59	
(a)+(b)	Total in Java		1.97		1.35
5.	Network.jacie		1.89		0.93
5(a)	Client Programs	1.17		0.83	
5(b)	Server Programs	1.02		0.66	
(a)+(b)	Total in Java		2.19		1.49

Table 7.2: Processing Comparison on JACIE and Its Java Translated Programs.

In conclusion, the code length of the larger JACIE programs is about a third of the size of the translated Java code. Although the similar applications can be written in several different ways, one cannot avoid using a lot of Java classes in producing the same output needed for the graphical user interface. The Java classes such as Applets, Frame, Panel, Component and Container must be included in the client's program. As regards compilation, the extra time of the translation phase producing Java from JACIE, is acceptable.

7.5.2 Preliminary User Study on JACIE

A case study on introducing JACIE to a number of people was conducted. This study consists of two teaching and three program writing sessions. The first teaching session is designed to give the overall picture of this language. It is presented using four slides on the

following topics.

- All of the JACIE standard components.
- Configuration component.
- Client implementation component.
- Server implementation component.

The main contents of these slides are to provide information on network connections, message passing activities, variable declaration and assignment and other related issues on handling a single user. The JACIE user interface is also presented. After this teaching session, the participant is required to write a *hello program.* During this writing session, the participant is given the teaching notes together with a sample JACIE program. This sample program is the generalised game of noughts and crosses (Section 5.3.3) with the elimination of inessential detail. The required output is shown in Figure 7.11.

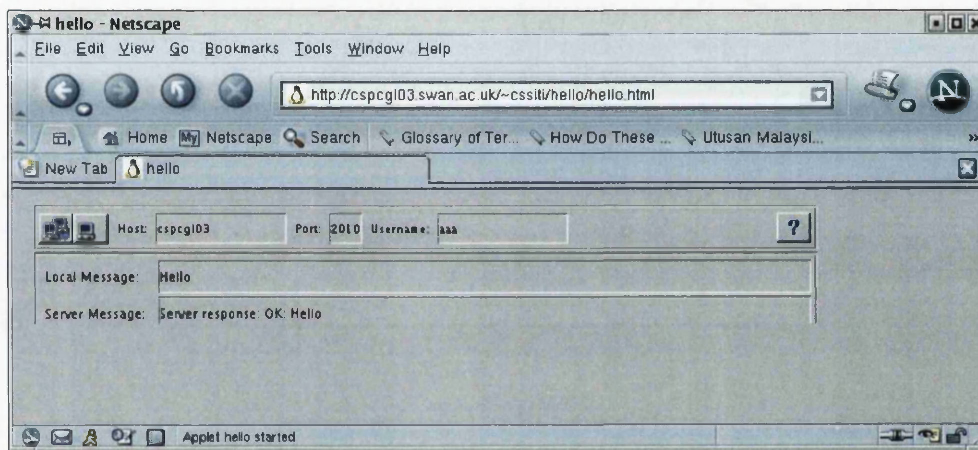


Figure 7.11: Output of JACIE Hello Program.

The second teaching session that uses three slides for its three program components introducing more JACIE statements. These include canvas manipulation, declaration of multiple users, interaction protocols specifically roundrobin and contention, JACIE events, message broadcasting and other related issues. The participant has to write a JACIE program that uses the roundrobin protocol to count and print the number of user's click on the canvas and the total number of clicks during the collaboration. The required output of this program is given to the participant. Similar to the previous session, the teaching notes and the same sample program are also supplied for the participant's reference.

The third session is very important to this research since its purpose is to test the significance of the interaction management design. In this session, the participant has to write the same program as in the second session but the interaction protocol is changed into contention. Figure 7.12 shows the screenshot of the required program. Here the total number of a user's click can differ, but the progam terminates when the total of 20 clicks have been made. This program only differs in the number of user's click if compared to the second program when both users make 10 clicks in a round robin fashion.

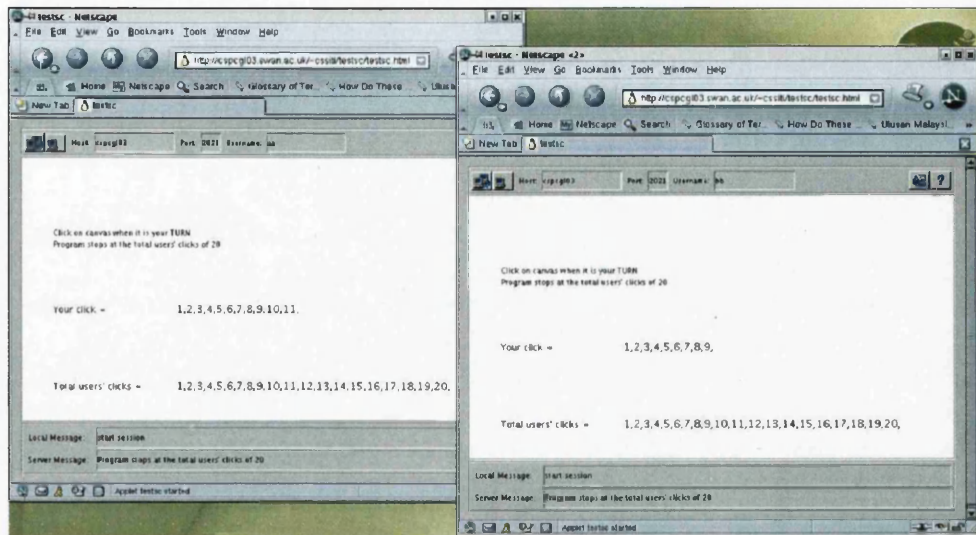


Figure 7.12: Output of Program with Contention Protocol.

There were five people involved in this case study, three from the computer science department (CS) and two from the engineering department (ENGR) at Swansea University. While two of these participants have some experiences in networked programming, others have none. However, all participants have a programming background, even though, the level of expertise may varies. Table 7.3 shows the data about all the participants' background obtained from the survey questions. A scale of 1 to 5 is used to indicate the level of experience, with 5 indicating the most experienced.

PARTICIPANT	DEPT	PROGRAMMING SKILL	NETWORK EXPERIENCE	TYPING SPEED (word/minute)
1	CS	5	5	22.5
2	CS	4	2	24
3	CS	3	1	22.5
4	ENGR	4	1	21
5	ENGR	2	1	10.5

Table 7.3: Participants Background Data.

Other data recorded from the study includes:

- The teaching time.
- The participant's program writing (typing) time.
- The participant's time on compiling (and editing) programs.

The teaching is conducted individually so that each participant can get the full attention and concentration. However, the method and teaching environment are the same for all.

Although the presentation of teaching materials takes about the same amount of time, the time for question and answers is also considered as that depends on the participant's understanding of such materials. The time is recorded after all participants are given the overview and basic concepts of the networked collaborative systems. Table 7.4 shows the recorded time to teach JACIE for its simple concept (T1) that can get the participant ready to write the first program (QI). It also shows the teaching time (T2) to introduce several additional statements to enable the participant to write the second program (Q2). The participant is required to compile both program one (Q1(Compile)) and program two (Q2(Compile)). Some additional program editing may be necessary when doing the compilation. Later, the participant has to write the last program (Q3) to determine the time taken to change the interaction protocol of the same application (the second program). The time measured for all the above work is in minutes (m). The program can be written in several different ways.

NO	T1	Q1	Q1(Compile)	T2	Q2	Q2(Compile)	Q3
1	8.6 m	9.2 m	2 m	7.9 m	13.5 m	3 m	12.3 m
2	8.2 m	7.5 m	3 m	8.5 m	25 m	6 m	13.5 m
3	11.2 m	11 m	3 m	13 m	17 m	7 m	14.8 m
4	11 m	11.5 m	3 m	12 m	18 m	6 m	14.5 m
5	12 m	16 m	6 m	15 m	24 m	8 m	17.7 m

Table 7.4: Learning and Testing Sessions.

Here, there are several factors that can influence the recorded time for writing the programs. For each participant, the background, programming experience, the understanding of networked collaboration, the ability to learn new knowledge and the typing speed may have the effects. From Table 7.4, we see that the longest time to write the 'Hello' program is 16 minutes for participant 5, who has the lowest level of skill. Participant 2, who has a very good programming skill, completes program 1 in the shortest time. However, participant 2 also has the fastest typing speed, so this could account for the good result for this simple program. Column T1, indicates that the simple concepts of JACIE can be taught within 15 minutes, allows the participant to write a client/server program. In writing the second program, participant 2, who is considered as a good programmer and has the fastest typing speed, takes almost the same time as participant 5. This is due to the fact that this particular participant has gone beyond the minimum requirement by writing the second program using long variable names, producing several user friendly program output and also adding several comments in the program. Participant 1,3 and 4 have similar 50 percentage increased in time to write this program compared to the first.

In writing the last program, all participants have shown a decrease in time for writing this program compared to the second. On average, all of them take about 1 to 1.5 minutes to determine the required changes in order to write the same program with different protocol as JACIE provides the interaction management statement. There are only two changes to make to the program,

- change 'protocol roundrobin' to 'protocol contention' and
- delete 'pass turn' statement.

Since the program is quite long, the length of time for each participant is constrained by the typing speed. The results have shown that all of them have very close time differences regardless of their experiences.

This case study has drawn several important findings. It concludes that JACIE is a simple programming language to learn based on the following factors.

- The number of presentation slides.
- The amount of teaching time.
- The number of provided materials for references.
- The amount of programming time.
- The amount of time to write a similar application with different interaction protocol.

With its templated-based feature, the structure of the program can be easily managed. This allows the participant to follow the flow of the program especially the networked support that hides the connection details and combining both client and server in a single program. Even someone with less programming skill can write a program in a short period of time similar to others when all of them are introduced to JACIE. With its interaction management features, one can easily modify one or two lines of code. Again, the programming experience has not much different since this feature eliminates the burden on the programmer to write quite a complex algorithm when changing the turn protocol. One of the participant had given a positive comment by stating that 'in JACIE, a programmer is only concerned with what to do rather than worry about how to do it'.

An attempt had been made to conduct a similar experiment in Java for comparative purposes. For this reason, some materials for teaching in Java is prepared. In order to keep the same number of slides as teaching JACIE, the output of the 'Hello' program has to be simple without the graphical user interface. Therefore, the client program only produces text-based line by line output. Figure 7.13 shows the output of both server and client.

The program is written in two different files that are compiled separately. Four slides are prepared that consist of the followings.

- Program layout, specifications and related features (e.g., object oriented, class definition, Java main method, Java syntax, etc.).
- The related programming features in Java (e.g., API packages, socket definition and connection, garbage collection, etc.).
- Server program requirements.
- Client program requirements.

The approximate teaching time without the question and answers, takes about 45 minutes to deliver (compared to 15 minutes for teaching JACIE for the 'Hello' program). A sample program that involves client/server is taken from the Bruce Eckel's book, 'Thinking in Java' [105]. This sample program is shown to participant 4, who has experienced in C language, without the object oriented features. This particular participant is reluctant to continue with

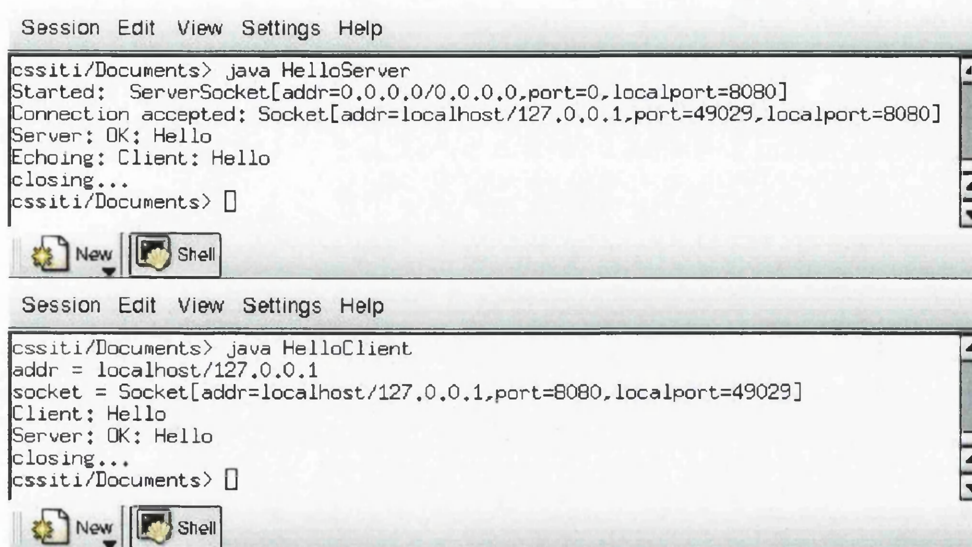


Figure 7.13: Output of Java Hello Program

the user study. The reason is that the Java syntax and the program flow are very hard to follow. Similarly to other participants, they are worried about the time factor involved and the complexity of the program. Therefore, with these responses, the case study on the Java language was not carried out.

In conclusion, this small case study shows that JACIE is a simple language to implement a networked application. It is able to keep a reasonable amount of time to learn, write and execute. The interaction management features help inexperienced programmers to manage user interaction in a program without concerning the details implementation of the protocols. Therefore, a programmer can focus more on the application's requirements.

7.5.3 Interaction vs. Transmission Delay

Interactive collaboratian requires the user to respond to the shared application in order for all the users to work together at the same time. Therefore, a few experiments have been carried out to determine the delay in the interaction and communication. Using the round robin and contention protocols in the variations of the noughts and crosses games, the time taken by a user (player) is recorded and printed in the output of a JACIE pre-defined server program. The relevant values (in millisecond) that are considered include the following.

- The packet transmission time between the sender and receiver.
- The time for a user to make a response to a computer.

We experiment with the transmission for two different conditions. First, the network transmission is calculated without any 'working data' using a `ping` command. Second, the transmission involving data in playing a generalised game (Section 5.3.3), where the transmission between client and server is recorded with first message from server to client. The

server records its time before this first message is sent. Upon receiving the first message, all the clients record their time and send back a message to the server. Then, when the server gets these messages, another recorded time is set. At the same time, the server also acknowledges the receiving message by sending another message to the corresponding client. In this way, both server and client can record the time for a transmitted message in a cycle.

MACHINE	CASE 1 (Game played)		CASE 2 (ping command)	
	SERVER	CLIENT	SERVER	CLIENT
1	1 ms	0 ms	0.026 ms	0.015 ms
	1 ms	1 ms	0.028 ms	0.018 ms
	46 ms	46 ms	0.027 ms	0.021 ms
2	4 ms	0 ms	0.176 ms	0 ms
	9 ms	10 ms	0.189 ms	0 ms
	1 ms	10 ms	0.181 ms	0 ms
3	1 ms	0 ms	0.191 ms	0 ms
	2 ms	0 ms	0.250 ms	0 ms
	1 ms	0 ms	0.211 ms	0 ms

Table 7.5: Transmission Delay Between Server and Client

Table 7.5 shows the results recorded for the transmission delay. Case 1 refers to recorded time of message transmission while executing the generalise game and Case 2 is the time recorded using the `ping` command. The unit of data is in milliseconds (ms). In Case 1, there are a lot of messages transmitted during the whole game. However, here, there are only three consecutive transmissions in three turns are shown. There is a sudden increase in the transmission time for the third data (Machine 1). It may be caused by the number of users working on the network or the number of concurrent jobs executing on the machine. Case 2 shows three consecutive packet transmissions. Since this test involves three different machines, such data can be influenced by the machine speed. Therefore, the concern is to compare the transmission rate of the same machine on two different conditions.

It is also significant to record the amount of time taken by a user to make a response to the computer. Here, we record the time for a user to click a mouse as quickly as possible. In comparison, a vicious battle (Appendix A.5) is played to record a message sent for every symbol put on the game board. A player clicks the board as quickly as possible to put five symbol in a row. Every action is recorded and printed in the output of the JACIE pre-defined server program.

Similar to the above approaches, Table 7.6 shows the results for message transmission during the vicious battle game for the same machines and the response time of the same player on the corresponding machine. With the same amount of data to be transmitted from either server or client as in the generalised game, this table shows the massive increase in the amount of time when the message is sent according to the number of user's action. This is represented by Case 1. Case 2 is the average amount of time for a user to do a mouseclick on the corresponding machine. The amount of time shows that the user response is about 1000 times higher than the message transmission rate. Both of the variations of the noughts and crosses games used do not involve a lot of calculations that can influence the delay. Table

7.5 shows the comparison that most of the transmission takes 0 to 46 milliseconds with little difference between the transmission without the game.

MACHINE	CASE 1 (Game played)	CASE 2 (Machine Interaction)
1	947 ms 1230 ms 1100 ms	201 ms 191 ms 193 ms
2	2258 ms 2054 ms 1645 ms	230 ms 223 ms 228 ms
3	980 ms 1372 ms 2233 ms	210 ms 250 ms 255 ms

Table 7.6: Interaction Delay on a User

In conclusion, both Table 7.5 and Table 7.6 show that the user's response has the larger impact on the delay for both games. Case 1 and 2 for Table 7.5 shows the small value in the results as this relates to the data transmission only while Table 7.6 has larger values when it represents the interaction. Therefore, network transmission time can be seen to be insignificant when compared to the user response time.

7.6 Summary

This chapter presented the implementation of the online Bridge game to test the design of the interaction protocols, and the simulation of network trouble shooting for testing the design of the interest management language constructs. Both applications have been successfully implemented and justified the significance of the design. In addition, several experiments have been undertaken for testing the JACIE language, interaction management features and the delay problem in some example programs presented in this research.

Online bridge, which is a complex and difficult game to implement usually requires an experienced network programmer if it is to be programmed using an ordinary general purpose language such as Java. JACIE has provided not only an easy to program feature, but also a collection of interaction protocols to fulfil the complicated interaction needs of the game in a structured manner. We have presented this game using three different protocol options. One is achieved using a built-in control protocol `round robin` throughout the game while the other two are implemented with dynamic changes using protocols `master user` and `protocol group`. The new language constructs can provide a JACIE programmer several protocol options in implementing an application.

The other application, that focuses on e-learning, shows how interest management can facilitates a programmer in implementing secure and efficient data sharing in a collaborative application. As most e-learning applications rely on a chat channel for communication, this

example shows that the chat channel is not the only possibility. With the data sharing facilities, one's resources can be shared and accessible by the users in accordance with the rules and specifications set by the owner. Therefore, it is an easy and helpful way for e-learning as information can be accessed directly from the owner's site.

The results on the experiments and conducted case study have shown that JACIE is a simple language. The interaction management is significantly helpful for programming multiuser networked systems. The presented example programs (the noughts and crosses games) show that the network transmission delay has insignificant effect.

Chapter 8

Conclusion

Contents

8.1 Summary of Contributions 189
8.2 Future Work 192

In this thesis, issues of interaction and interest management in interactive networked collaborative systems have been discussed. This work provides a set of new language constructs for interaction control protocols and data sharing incorporated into the scripting language, JACIE (*Java-based Authoring Language for Collaborative Interactive Environments*), and has achieved its objectives as outlined in Chapter 1. In particular,

- We have conducted an abstract study of the interaction activities and data transmission in networked collaborative environments through a large number of variations of the noughts and crosses game.
- We have designed a set of high level language constructs for specifying a variety of interaction protocols.
- We have designed a set of new high level language constructs for specifying secure variable sharing.
- We have implemented these language constructs in the form of a major extension of the scripting language JACIE I.
- We have developed two demo applications for demonstrating the technical feasibility and usefulness of the design.
- We have improved the JACIE I compiler and some of its language features.

In general, the enhancement of the language and its compiler allows a wider coverage and flexibility in designing and implementing various forms of interactive collaborative applications. For the interaction management, the new language construct provides the use of simple statements with several options. It gives further flexibility to JACIE II programmers to add or set their own interaction protocols. It is also allowable to dynamically change the protocol setting in the middle of a session. The interest management constructs allow a

fast and simple way for variable sharing among users. These new features that are added to JACIE, not only provide a secure means of resource sharing, but also interest filtering that helps prevent the server from sending unnecessary messages to clients.

8.1 Summary of Contributions

This work started by doing a comprehensive survey on the interaction and interest management in networked systems, programming languages and software tools. This study gives a broad understanding of commonly employed techniques in handling interaction needs and secure data sharing with filtering features. Several systems within networked collaborative systems have also been reviewed to look into the system features and related applications. The following subsections describe the work that we have contributed in order to provide the proposed interaction and interest management language constructs.

Selected parts of this thesis have been presented in the following publications:

- the IEEE International Symposium on Multimedia Software Engineering under the title *Managing Interaction for Multimedia Collaboration - through the keyhole of noughts and crosses game* [3], in Florida, USA.
- the Elsevier Journal of Network and Computer Applications under the title *Designing Interaction Protocols using Noughts and Crosses Type Games*, Volume 30, Issue 2, April 2007 [4].

8.1.1 The Collection of Interaction Protocols

We extended and elaborated the previous case studies on the noughts and crosses game [139]. The study of similar games to the noughts and crosses, either cell-based or in other forms had also been considered. Games, such as Gomuku, Connect-4 and Three Stones, are examples of related games. With some modifications to the game rules, we introduced many variations of the noughts and crosses game. At the same time, to have a realistic design, the real applications associated with the games variations have also been studied and proposed. Details on this case study are described in Chapter 5. Formal notations for modelling the spatio-temporal activities in a noughts and crosses game have also been outlined.

8.1.2 Interaction Management Implementation

Although the concept for interaction management was introduced in the JACIE I language, the constructs provided were not easily customisable. This became quite apparent when we attempted to implement many variations of the noughts and crosses game. These variations provided us with an effective means for identifying the many different types of protocols that can exist and the useful parameters for their customisation.

The language constructs that we have implemented include the protocols *round robin, contention, reservation, master, tapping* and *group protocols.* Within protocol *contention,* we also introduce another protocol *contention hold,* which is similar to 'first come first serve' and within protocol *master,* programmers can have several options whether to select the server (by default) to be the master or otherwise, choose one of the client as the master. The programmers can also choose to add their own design of application specific protocols without relying on the provided protocols. For handling groups, the protocol *userdefined, roundrobin, random* and *master* can be used to select one member from a group to handle inter-group collaboration. Some customisations that have been added to these protocols include timer options (*turn timer, overall timer, silence timer* and *rest timer*), number of actions per turn and several supporting statements (*turn pass, action start, action end, turn request Boolean, turn set client* and *turn set group*).

8.1.3 Interest Management Implementation

We have initiated an original investigation into the subject of interest management by reviewing the possible systems that have such mechanisms. The types of shared resources and the techniques involved have also been studied. We looked at several shared resources such as shared memory, shared object, variable, program component and database. The design and implementation of sharing such resources are applied in many systems such as parallel, concurrent, multi-agent, database and general networked systems. We have studied the interest filtering mechanisms in networked environments which are mostly implemented in collaborative virtual systems and simulation systems.

For designing the language constructs, several factors have been considered that include the *access rules* on sharing variables among users and the determination of *user list* that represents the list of users who can be granted access rights on a shared variable. The *access rules* include *to own, not to own, to read, not to read, to write* and *not to write.* Thus, these *access rules* facilitate a user to become the owner of a shared variable and the user can perform `read` or `write` operations. The *user list* consist of *all, others, group, me* and a specific list of users ({ $\mu_1, \mu_2, ...$ }). With the specific list, an owner of a shared variable has the flexibility to select any user to have variable access without depending on any user classifications. Other *user list* options can provide a quick setting to particular classifications of users. The security option on a shared variable is performed by a *password* protection that can be used in the read/write operation.

In the relevance-based filtering, a user is required to set an 'interest value' and this value will be compared to the owner's 'filter value'. Both of these values are between 0 and 1, inclusively. It is possible for both (user and owner) not to set any value that results in the assumptions that the user is 'not interested' and the owner allows 'no restriction'. The `filtering` operation causes the *user list* to be filtered into user *access list* that represents the actual users who are granted the access to a particular shared variable.

The new interest management statements include *use statement* for the owner to set shared variable's permission, *set statement* for read/write operations, *check statement* for selection

control on using the shared variable, and *interest set* and *filter* for filtering operations. Some of these statements are tested on implementing one of the noughts and crosses game variations, namely *Secret Switch.*

8.1.4 Major Language Enhancements

In adding several new statements into a language requires several new productions for the grammar. The new productions need several new non-terminal and terminal symbols. Therefore, modifications have to be made on the existing JACIE I compiler to accommodate these changes. For translating JACIE into Java, many new Java classes are introduced with additional new Java modules.

At the initial stage, the online Bridge game was implemented as a benchmark to identify major components of the JACIE I language and its compiler. Once the original design and details of the compiler constructions were known, the design of the language constructs were built upon JACIE. Chapter 4 has provided a comprehensive review of this language with its special features, compiler components, language architecture and other highlights of the language. The structured compiler construction of this language had allowed our design of the language constructs to enhance its features.

8.1.5 Demonstration Applications

Two applications namely an e-learning program for the Simulation of Network Trouble Shooting and online Bridge, were implemented using the extended JACIE language, JACIE II. The implementations have proven that our designs facilitate flexible and simple to program capabilities. Firstly, several protocols are tested in implementing the same game, online bridge. Three different protocols have been used that enable us to compare and distinguish the usefulness of the protocol designs. Secondly, we implemented an e-learning application to test the interest management design. It is apparent that the language constructs give a fast way of communicating while allowing users to share their workspaces rather than giving suggestions and discussions through the chat channel. Chapter 7 provides examples of the code segments of these two implementations, together with a number of the screenshots.

8.1.6 Minor Improvements on the Language and Compiler

In addition to the language extensions on the interaction and interest management, we had also foreseen other issues that had related to these language extensions indirectly, but it is significant to make some improvements for the overall language design.

In handling user collaboration concurrently, the race condition has become the issue that must be taken into account. We have detected some parts of the compiler that may face the chances of having the race condition and lead to the execution of thread to crash. Therefore, we have made some changes to some parts of the compiler program to reduce these chances.

We have introduced a new variable type, namely *Grid2D* so that any grid manipulation on JACIE canvas channel can be performed clearly with its declaration and manipulation are stated clearly. With this new grid type, the 'grid object' can be included in a parameter list of a method that results in shorter and organised program.

It is significant to provide a programmer with information on some illegal attempts in developing an application. Therefore, we have introduces several new messages that can be 'warning' messages or 'error' messages. With the 'warning' messages, the compilation of a user program continues by skipping the statements that contain the illegal actions, while any error would terminate the program compilation process. Details on this work is described in Chapter 4.

8.2 Future Work

The language JACIE, has now gone through its second major development. The first version was concerned more with the ability of the language to provide a tool to easily design and implement collaborative application with a set of multimedia channels, template and event-based programming. In this second development, the comprehensive set of language constructs for interaction management has been designed, implemented and tested. Even though the main objective to implement a set of comprehensive interaction protocols has been achieved, the group protocol needs further improvement, particularly on the selection of group members. Currently, there is no flexibility in choosing the group members. The selection is made by the JACIE compiler according to the users' arrival. Users are assigned to groups in a cyclic fashion. It is better for the protocol design to have some flexibilities for the users to choose their groups according to some rules and regulations. This will give a more realistic environment for group collaboration. The improvement on choosing the group is not only when a user starts a session, but also, it is practical to allow one, some or all the users to change their groups freely during a session.

Since the use of one or more types of timer options are permissible within one protocol, JACIE allows the timer to be set at the start of a session. No changes to a timer option could be done during the ongoing session. Therefore, there are no timer options for the dynamic change in the protocol setting. To add this feature, it requires quite substantial changes to the code translator. Having dynamic timer options along with the dynamic protocol settings can give great flexibility in the interaction management.

As for interest management, there are many rooms for further improvements since this feature is still new in JACIE. For example, the only filtering technique available is fixed and rigid with a preset value in $[0 \ldots 1]$. Other research has proved that there are a wide range of methods in determining users' interests such as user distance and interaction frequency. Including these choices of filtering methods into JACIE requires specific algorithms for fitting them to the JACIE language background.

The access method that we have implemented is achieved according to a certain set of user

lists. The user lists consist of *all*, *others*, *group*, *me* and *a specific list of clients*. It is possible to add the user list by introducing a combination of groups in the form similar to the *specific list of clients* that we already have.

In general, the language itself, JACIE has not focus on developing its visual representation and graphics capabilities. It caters for 2D graphics presentation. Before the start of the second version of language enhancement and development, we studied the possibilities of implementation under a different environment. The .Net Framework was our target. The idea behind this environment change was to ease enhancement of visual representation and graphics features provided by .Net. However, since building a language relies heavily on compiler construction, this aim was abandoned because there were no compiler tools provided under the .Net environmentat at that time. Eventually, the development of this language continued in Java.

In collaborative systems that involve concurrent processes, a race condition can happen. In JACIE, it has been detected that a race condition can happen at both server and client programs. In both programs, each has a queue to be used in message exchange between them. In the future, a locking mechanism can be included in the program when the action to add data into the queue and retrieve data from the same queue are to be made. It is to ensure that both actions can be performed in a mutual exclusive manner.

The language provides a set of multimedia communication channels. All the channels are designed and implemented to serve the basic needs of communication. Improvement of the advance features of such channels can provide the language with powerful tools in building interactive collaborative applications. For example, these channels could have the flexibilities to be set as private or public at any time. However, this improvement requires a lot of research and programming effort as Java can provide various options to implement very good, structured and flexible communication methods.

Appendix A

Variations of the Noughts and Crosses Games

The variations of the noughts and crosses games summarised in Table 5.1 are described in more detail in this appendix.

A.1 Five-in-a-line

Five-in-a-line game, which is called Wuzigi in China, Gomuku in Japan, Gobang or Goban in the West, is usually played with a Go board game set. It is commonly played in China, Japan and Korea as an initiation for beginners of Go. It uses a board, Figure 5.1, with 19×19 grid points (as pieces are placed on grid points rather than in cells), and black and white go pieces are used instead of noughts and crosses. A player, who first completes a 5-cell line horizontally, vertically or diagonally, wins the game. The game is usually associated with a more elaborate set of rules than noughts and crosses, for example, any player must announce if he/she reaches a condition that could lead to victory if undefended by the opponent.

In terms of interaction management, and related applications, this game can be considered as a special case of the generalisation in 5.3.3. It also uses round robin mechanism that can support similar applications as mentioned before, however, additional and probably more complicated rules can be included to the implemented applications. For example, in an automatic tele-information service, a user is allowed to have immediate contact to an appropriate person for help support in any special cases.

A.2 Connect-4

A connect-4 board is a vertically-standing (instead of flatly laid) grid of cells, typically of 7×6 cells. Two players are assigned to red and green pieces respectively. Each player takes it in turn to drop one of his/her pieces from the top of the board into one of the column

slots with empty cells, hence only the lowest empty cell in any column is 'modifiable'. A player who first completes a 4-cell line (horizontal, vertical or diagonal) wins the game. It is possible for a game not to have a winner.

This game can be generalised in a similar way to the traditional noughts and crosses games. From the perspective of interaction management, this game introduces the notion of 'unmodifiable' cells, ■, as illustrated in Figure 5.1. The mechanism for changing the state of an empty cell from ■ to □ is also interesting, as it can be facilitated by appropriate interest management and may require the incorporation of application-specific code segments.

The modifiability of an interactive component is not uncommon in practical applications. For example, complex online forms in web-based interaction often have items activated or deactivated after some other items are filled in. One can also imagine in a collaborative design environment (*e.g.,* computer-aided system design or computer-aided geometry design), available design options change during the design process according to the previous design steps taken.

A.3 Three Stones

This is a two-player game using an octagonal board as shown in Figure 5.1. Each player takes turn to draw a stone from a pouch but normally can only place the stone in any empty cell in the same row or column as his/her opponent's last played. In the case no empty cell satisfies this condition, the player can place the stone in any empty cell on the board. There are three different colours of the stones in the pouch, namely black, white and clear, with the clear stone considered as a 'wild card' for both black and white. Two players are assigned to black and white respectively. Every time a move results in three stones of the same colour in a row, a score is awarded to the player associated with that colour. When a player draws out a stone of the opponent's colour, he/she must also play, usually in a way to avoid awarding a score to the opponent. The player with the highest score when the board is completed filled wins the game.

In addition to the complexity introduced due to the wild card and the octagonal board, like connect-4 in A.2, the game also requires the management of the modifiability of empty cells. Therefore, in the application such as online form filling, users can have greater flexibilities in filling the form by having more selections of the 'activated' items to work on and they can also choose whether or not to give information on selected sections.

A.4 Hasty Battle

This is one of the imaginary games, designed to address the interaction needs of some collaborative applications. Similar to the generalised noughts and crosses game in 5.3.3, two players are assigned to nought and cross respectively. There is no turn control in this game. Both players place their assigned symbols, one each time, in any empty cells as fast as they

can. The player, who first completes a w-cell line horizontally, vertically or diagonally, wins the game.

Though the fact of not having any turn control seems to contrary to the essence of collaboration, this feature is common place in numerous real-life or computer based collaborations. It is usually an efficient and effective means for managing access to shared resources in a distributed collaborative environment, for example, accessing critical sections of a shared database, and queueing for printing facilities. It is this imaginary game that raises a number of design issues for the contention protocols in Section 5.4.2, and challenges the process for validating user actions, and resolving conflicts in interactions. In any of its applications, such as printing queueing, it is important to ensure that only one user is having access while other users must wait until the particular user finishes the printing and releases the printer. In this work, the term for this protocol is called *contention.*

A.5 Vicious Battle

This is an imaginary game that goes further than the hasty battle in A.4 in simulating unmanaged, or almost unfriendly, collaborations. The game has neither turn control, nor cell control. Like hasty battle, two players are assigned to nought and cross respectively, and the player who first completes a w-cell line wins the game. However, unlike hasty battle that has a cell control mechanism, players can overwrite symbols that are already in the cells. In other words, the game requires the use of three modifiable states, namely □, ⧇ and ⊠ as defined in Section 5.3.2 until the game ends.

One might think that there is no place for such an unmanaged and unregulated interaction protocol in collaborative activities. In fact, many shared applications in collaborative environments have been designed and developed in this manner. For instance, a shared whiteboard in video conferencing normally does not have any turn control or domain control. Users are free to overwrite anything drawn on the board previously, and are 'trusted' to self-regulate their interactive activities.

This game has also an interesting minor variation, where two players are required to swap their assigned symbols after a certain period of time ΔT until one wins the game. At the end of each period ΔT, the player who was playing with nought will play cross in the next period of ΔT, and vice versa. This can be considered as a simplified scenario where two authors are working on different aspects of a document collaboratively through a shared word-processor application and swap their roles after a period.

This game is also implemented using *contention* protocol, similar to hasty battle A.4, but vicious battle allows the resource, such as shared whiteboard, to be written to by any user at any time.

A.6 Gentlemen's Battle

Similar to the generalised game in 5.3.3, except that each player is required to inform the opponent of the cell where he/she is about to place a symbol. The player can place the symbol only after getting a permission from his/her opponent.

Although this is almost nonsensical and unworkable as a competitive game, it features as an essential protocol in any courteous human interaction and collaboration. It challenges the efficiency of protocol management with extra interactions between clients in a computing environment, despite the fact that it is often achieved in real life in a spontaneous, effortless but effective manner.

This interaction protocol is commonly found in a forum discussion where there is no proper order of user turn to be performed. Based on the matters being discussed in the forum, the current user can appoint any users (participants) to speak and it is important to make sure that only one can speak at any given time.

A.7 Dictator's Entertainment

This is perhaps the opposite of the gentlemen's battle in A.6. Two players are assigned to nought and cross respectively. Two players take it in turn to place their assigned symbols. The turn is determined by a 'dictator' (e.g., the server) at random. The player, who first completes a 5-cell line horizontally, vertically or diagonally, wins the game. A minor variation is to introduce a turn timer, which allows each player to play as many symbols as possible within a set period δ_{turn}.

Despite the less pleasant name of the game, we can find many collaborative activities are conducted, to a large extent, in such a manner, for example, in an e-learning class involving a teacher and several students, floor management of an online meeting, computer-assisted testing with automatic question selection.

In e-learning class, the 'teacher' role represents the 'master' who can control the turn of all the students in the class by a random order. Similar to an online meeting, the 'master' is the chairman who usually assigns the turn according to the matter being discussed and it can also be performed at random.

A.8 First-Come, First Served

Like the hasty battle in A.4, two players compete for their turns. However, unlike the hasty battle, once obtaining a turn, a player can hold on to it, and places as many symbols as he/she likes, until the player voluntarily gives it up. To prevent indefinite holding of a turn, a *turn timer* can be introduced to limit each player to have a maximum of $\delta_{turn} \in (0, \infty)$ seconds in each turn. Alternatively a *silence timer* can be used to seize the turn control back

after a predefined period of $\delta_{silence} \in (0, \infty)$ seconds within which the player fails to make any move.

As a competitive game, it would probably work only with very large values of N_x, N_y and w. However, in real life collaborative activities, this is a very typical playground protocol. It can also be deployed successfully in applications involving the control of shared resources, for example, camera control in collaborative surveillance [78]. In this application, a user who is fast enough to hold a camera control button, blocks others from getting the turn control. In this way, the user in control can view the videos that are presented by all the provided cameras, while other users can have other attempts once the camera control button is released.

A.9 Opportunity Knocks

Similar to the dictator's entertainment in A.7, this imaginary game also involves a 'master' (e.g., the server), which randomly sets 50% empty cells modifiable, □, and the other 50% unmodifiable, ■, at the beginning of each turn. Two players take it in turn to place their symbols, one each time in a modifiable empty cell. A minor variation of this game may also include opportunities, again randomly generated, for overwriting the opponent's symbol.

From the perspective of protocol design, this game requires dynamic management of an action domain as in the connect-4 game in A.2. In fact, the minor variation of the game involves the use of the full set of cell states, $S =$ {□, ■, ⧇, ▣, ⊠, ▩}. This simulates certain aspects of collaborative activities, where the operational domain of each member in a team is restricted but can change dynamically. The management of the changing operational domain is not a trivial task in any distributed software, leading to the discussions on interest management in Chapter 6.

This game is implemented using round robin and its management can be found in real applications such as in an emotional IQ test. This test can describe a person's ability to understand his or her own emotions and the emotions of others so that appropriate action can be taken based on this understanding [168]. The set of questions can be presented randomly and the score is based on the selected questions and answers.

A.10 Secret Switch

In this imaginary game, there is a special cell, called *secret switch*, on (or attached to) the game board. The secret switch is initially set, randomly and secretly, to either a nought or a cross. Similar to the generalised noughts and crosses game in 5.3.3, two players take it in turn to place their symbols. However, at the beginning of a turn, each player must first guess what is in the secret switch, either a nought or a cross. If the player guesses correctly, he/she continues to place an assigned symbol on the game board; otherwise no move is allowed. Before passing the turn to the opponent, the player resets the symbol in the secret switch to

either a nought or a cross, for the opponent to guess.

This can be seen as an abstraction of collaborative applications with security management, such as accessing distributed databases or other resources that are protected by passwords. In implementing these applications, it is important that user access rights must be determined before hand so that any attempt to read or modify data can be protected.

The secret switch can be implemented in many ways. We are particularly interested in how such a mechanism can be supported by an interaction protocol, and how the 'shared space' of the special cell can easily be specified with appropriate program constructs. Both of these two issues will be addressed in Sections 5.4 and Chapter 6.

A.11 Group Games

Most of above-mentioned games can easily be generalised to involve more than two players. Here we focus on a group game that is uncomplicated but introduces essential additional needs in protocol design. We consider two groups with 2 players in each group. In such a setting, there are two levels of protocol management, that is, *between groups* and *within a group*. Consider a simple round robin mechanism for managing turns between the two groups. Two groups are assigned to nought and cross respectively. Each group takes it in turn to place its assigned symbols. The group that first completes a w-cell line wins the game. We can have many different schemes for managing turns within each group. For instance, (a) players within a group may also take it in turn, (b) the first move whichever player plays becomes the group's move, (c) only one player (i.e., a leader) in each is allowed to make a move, or (d) players can be allowed to discuss the move and decide their move in consultative manner. Such a protocol can also be moderated using a *group turn timer*, especially when discussions are allowed in a group.

It is not difficult to imagine many applications that fit this particular model, as many collaborative activities involve multiple teams and groups, which work in a collective and often co-ordinated manner. Several users can form a group, then collaboration among the groups can be performed. There are several issues that can arise that include the determination of group members and the inter-group interaction protocol.

An example application that facilitates this protocol is in a group discussion. In this application, manipulation of timer options such as *round robin* with time out can be used for the case that requires several stages in discussing a subject matter. It is possible to 'reward' users with extra time if they have used less time at the beginning and more time towards the end. It can be imagined that users need a short duration of time in 'brain storming' session at the beginning of the discussion, and later, extra time may be required to discuss the conclusion.

Appendix B

Nomenclature

All the terms used throughout this thesis can be divided into two different tables. One consists of the common terms in the context of this research work and the other contains the mathematical notations. As there are a lot of terms with different meaning in different context, especially in areas of networked communication, Table B.1 is provided to clarify the meaning that this whole thesis is referred to.

Term	Definition
Data sharing	Any form of data that are globally used by people in a collaborative system.
Distributed system	A system that consist of more than one computers at distributed places and link together.
Environment	A situation where users are working
Group collaboration	The interaction between several users that form a group.
Interaction protocol	A rule that governs structured interactive activities.
Interest management	A filtering of secured data access.
Language construct	A part of a language that has syntactically determined elements to form a statement.
Networked Collaborative System	A system where several people work together remotely and are connected by a network.
Software tool	A provided software system to assist a system developer in implementing networked applications.
Shared variable	An identifier declared to commonly used in an application.
Structured activities	Users' activities that must be performed in a specific order.
System-defined program	A set of instructions written by a language developer/designer.
Un-structured activities	Users' activities that require no specific turn control mechanism.
Users	The people who involve in a collaboration.

User-defined program	A set of instructions written by a programmer.

Table B.1: Technical Terms.

The following table, Table B.2, lists mathematical symbols commonly used in this thesis.

Mathematical Term	**Definition**
G	A game board
N_x, N_y	Number of cells in x and y directions
$< i, j >$	A cell position
$\mathbb{N}$	Set of all natural numbers
Mod	Write access state; $\{mod, unmod\}$
$\mathbb{K}$	Set of all processors
p	A processor
act	A function to describe an action
$\mathcal{G}$	A homothetic state
S	A State
t	Time parameter
K	Number of clients
L	Number of groups
grp	A group
ψ	A discrete temporal function
δ	A timer variable option

Table B.2: Mathematical Symbols.

Appendix C

The JACIE II Language Specifications

JACIE II language specifications are divided into two sections that consist of its token specifications and the language constructs.

C.1 Token Specifications

JACIE II data have two types; primitive and compound. The primitive types are listed in Table C.1 and the compound type is an array that can have a collective data of primitive types.

Type	*Size/Format*	*Description*
int	4 bytes	Integer
double	8 bytes	Numbers with fractional parts
boolean	`true` or `false`	Boolean
image	`gif` or `jpeg` typed images	Image
string	a series of characters between double	Character string
grid2D	an object	a created Java Grid class

Table C.1: Primitive Data Type.

The operators for arithmetic operators are given in Table C.2 and relational and conditional operators are listed in Table C.3.

Type	*Description*	*Type*	*Description*
+	addition	/	division
-	subtraction/negation	%	modulus
*	multiplication		

Table C.2: Arithmetic Operators.

<table>
<tr><th>Type</th><th>Description</th><th>Type</th><th>Description</th></tr>
<tr><td>></td><td>greater than</td><td>|</td><td>or</td></tr>
<tr><td>>=</td><td>greater than or equal to</td><td>&</td><td>and</td></tr>
<tr><td><</td><td>less than</td><td>^</td><td>exclusive or</td></tr>
<tr><td><=</td><td>less than or equal to</td><td>||</td><td>logical or</td></tr>
<tr><td>==</td><td>equals to</td><td>&&</td><td>logical and</td></tr>
<tr><td>! =</td><td>not equals to</td><td>!</td><td>not</td></tr>
</table>

Table C.3: Relational and Conditional Operators.

JACIE comments are represented by symbol '//' for a one line comment and for multiple line comment, it starts with the symbol '/*' and ends with '*/'. A variable name must begin with a letter and be a sequence of letters or digits, which is similar to the variable declaration in the Java language.

C.2 Syntax Specifications

This section provides JACIE II syntax specifications (in terms of productions) divided into several components according to their features.

C.2.1 JACIE Program Body

<table>
<tr><td><JACIE program></td><td>::=</td><td>JACIE {
<create application> | <create applet>
}</td></tr>
<tr><td><create application></td><td>::=</td><td>application name <identifier> ;
<jaciecontent></td></tr>
<tr><td><create applet></td><td>::=</td><td>applet name <identifier> ;
[<create applet option>]
<jaciecontent></td></tr>
<tr><td><jaciecontent></td><td>::=</td><td>configuration {
<program configuration>
}
messages {
<message definition>
}
client implementation {
<client program implementation>
}
server implementation {
<client program implementation>
}</td></tr>
<tr><td><create applet option></td><td>::=</td><td>appletlauncher</td></tr>
</table>

		<text button launcher> | <image button launcher>
<text button launcher>	::=	**text** <string>
<image button launcher>	::=	**image** <string>

C.2.2 JACIE Configuration Section

<program configuration>	::=	<specify hostname>
		<specify port number>
		<specify username>
		<configuration statement>
<configuration statement>	::=	<specify channel> ;
		<specify about> ;
		<specify observers> ;
		<specify number of groups> ;
		<specify number of users> ;
		<specify protocol> ;
<specify hostname>	::=	**host**
		<string> | **prompt**
<specify port number>	::=	**port**
		<integer number> | **prompt**
<specify username>	::=	**username**
		<string> | **prompt**
<specify channel>	::=	**channel** <channel name> { , <channel name> }
<channel name>	::=	**canvas** | **chat** | **whiteboard** | **voice** | **video**
<specify about>	::=	**about**
		<string> | <about file>
<about file>	::=	**file** <string>
<specify number of users>	::=	**number of users**
		<integer number> | <specify minimum> |
		<specify maximum> | <specify range number>
<specify minimum>	::=	**minimum** <integer number>
<specify maximum>	::=	**maximum** <integer number>
<specify range number>	::=	<specify minimum> **upto** <specify maximum>
<specify observers>	::=	**number of observers** <integer number>
<specify protocol>	::=	**protocol** <protocol_choice> <timer_option>
<protocol_choice>	::=	**group** <specify group> | <specify user>
<specify user>	::=	**contention** [<contention_option>] | **roundrobin** |
		reservation | **master** <master_option> |
		tapping
<contention_option>	::=	**hold**
<master_option>	::=	**server** <master_choice> | **client**
<master_choice>	::=	**random** | **userdefined**
<timer_option>	::=	[**turn** <integer number>] [**rest** <integer number>]
		[**overall** <integer number>]

		[**silent** <integer number>]
		[**action** <integer number>]
<specify group>	::=	**userdefined** | **random** | **roundrobin** | **master**
<specify number of groups>	::=	**number of groups**
		<integer number> | <specify minimum> |
		<specify maximum> | <specify range number>

C.2.3 Message Definition

Below is the productions for declaring user-defined messages that are sharable between client and server.

<message definition>	::=	[<one or more identifiers>]
<one or more identifiers>	::=	<identifier> { , <identifier> }

C.2.4 Client Implementation and Server Implementation Sections

<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>	::=	<declaration list> ;
<declaration list>	::=	<variable declaration list>
		<method declaration list>
		<declaration list>

C.2.5 Variable Declaration

Below are productions for the declaration of variables both at the client and server program. The *keyword* 'shared' can be used in both programs. In the client program, this variable is treated as a 'global' variable that is sharable by all the users, while at server, a variable with 'shared' declaration is used globally only to all of the server program components.

<table>
<tr><td><variable declaration list></td><td>::=</td><td>{ [shared] <data types> <variable declarator></td></tr>
<tr><td><data types></td><td>::=</td><td>int | double | boolean | image | string | grid2D</td></tr>
<tr><td><variable declarator></td><td>::=</td><td><primitive type> | <compound type></td></tr>
<tr><td><primitive type></td><td>::=</td><td><variable> { , <variable> }</td></tr>
<tr><td><compound type></td><td>::=</td><td>[<value>] { [= <value>] } <variable></td></tr>
<tr><td><variable></td><td>::=</td><td><identifier> [<variable initialiser>]</td></tr>
<tr><td><variable initialiser></td><td>::=</td><td><value> | <array initialiser></td></tr>
<tr><td><array initialiser></td><td>::=</td><td>{ <variable initialiser list> }</td></tr>
<tr><td><variable initialiser list></td><td>::=</td><td><variable initialiser> { , <variable initialiser> }</td></tr>
</table>

C.2.6 Method Declaration Statements

<table>
<tr><td><method declaration list></td><td>::=</td><td>{ [shared] <method header> <method body></td></tr>
<tr><td><method header></td><td>::=</td><td><data types> | void | <method header list></td></tr>
<tr><td><method header list></td><td>::=</td><td><identifier> ([<formal parameter list>])</td></tr>
<tr><td><formal parameter list></td><td>::=</td><td><formal parameter> { , <formal parameter> }</td></tr>
<tr><td><formal parameter></td><td>::=</td><td><data type> <identifier></td></tr>
<tr><td><method body></td><td>::=</td><td><compound statement></td></tr>
</table>

C.2.7 Basic Statements

<table>
<tr><td><compound statement></td><td>::=</td><td>{ [<statement list>] }</td></tr>
<tr><td><statement list></td><td>::=</td><td><statement> | ; { <statement> }</td></tr>
<tr><td><statement></td><td>::=</td><td><comment statement>
| <expression statement>
| <control statement>
| <iteration statement>
| <input output statement>
| <graphics statement>
| <event control statement>
| <communication statement>
| <interfacing statement>
| <interaction statement>
| <interest statement>
| <compound statement></td></tr>
<tr><td><comment statement></td><td>::=</td><td><traditional comment>
| <end of line comment>
| <documentation comment></td></tr>
</table>

```
<traditional comment>       ::= /* {<comment content>} */
<end of line comment>       ::= // {<comment content>} <line terminator>
<documentation comment>     ::= /** {<comment content>} */
```

C.2.8 Expression Statements

```
<expression statement>   ::= <arithmetic expr>
                           | <conditional expr>
                           | <array access>
                           | <method invocation>
                           | <postfix expression>
<arithmetic expr>        ::= <arithmetic> | <unary expression>
<arithmetic>             ::= <expression> <arithmetic symbols>
                             <expression>
<arithmetic symbols>     ::= + | - | * | / | %
<unary expression>       ::= <unary symbols> <expression>
<unary symbols>          ::= + | - | !
<conditional expr>       ::= <expression> <conditional symbols>
                             <expression>
<conditional symbols>    ::= || | && | | | ^ | & | <relational symbols>
<relational symbols>     ::= == | != | < | > | <= | >=
<array access>           ::= <identifier> [ <expression> ] { [ <expression> ]}
<method invocation>      ::= <identifier> ( { <argument list> } )
<argument list>          ::= <one or more identifiers>
<postfix expression>     ::= rnd ( <expression> )
<expression>             ::= <unary symbols> <identifiers>
                           | <identifier> <relational symbols> <identifier>
                           | <expression list>
<expression list>        ::= <identifiers>
                           | <value>
                           | <system variable>
                           | <expression list> <arithmetic symbols>
                             <expression lists>
<system variable>        ::= USERNAME | USERNUMBER | GROUPNUMBER |
                             MESSAGEID | CURRENTTURN |
                             CURRENTGROUPTURN | GETTEXT | GETX |
                             GETY | GETGRID | GETGRIDX | GETGRIDY
```

C.2.9 Control Statements

```
<control statement>      ::= <if statement>
                           | <check statement>
                           | <return statement>
                           | <exit statement>
<if statement>           ::= if ( <expression> ) <statement>
```

		{ **else if** (<expression>) <statement> }
		[**else** <statement>]
<return statement>	::=	**return** [<statement>]
<exit statement>	::=	**exit**

C.2.10 Iteration Statements

<iteration statement>	::=	<for statement> | <while statement>
<for statement>	::=	**for** (<for init> ; <conditional expr> ; <for update>)
		<statement> | <compound statement>
<for init>	::=	<arithmetic expr> | <local var declaration>
<for update>	::=	<arithmetic expr>
<local var declaration>	::=	<data type> <variable>
<while statement>	::=	**while** (<conditional expr>
		<statement> | <compound statement>

C.2.11 Input Output Statements

<input output statement>	::=	<input stmt> | <print stmt> | <clear stmt>
<input stmt>	::=	**input** <receiver list>
<receiver list>	::=	<one or more identifiers>
<print stmt>	::=	**print** <message bar> [<output list>]
<clear stmt>	::=	**clear** <message bar>
<message bar>	::=	**servermessage** | **localmessage**
<output list>	::=	<expression list> [{ + <expression list> }]

C.2.12 Graphics Statements

<graphics statement>	::=	<canvas statement> | <draw statement>
		| <colour statement>
<canvas statement>	::=	<canvas size> | <canvas definition>
		| <canvas specify> | <refresh screen>
		| <clean canvas>
<canvas size>	::=	**canvas size** <pair expression>
<canvas specify>	::=	**use canvas** <identifier>
<refresh screen>	::=	**refresh**
<clean canvas>	::=	**clean**
<colour statement>	::=	**foreground** <draw colour>
		| **background** <draw colour>
<draw statement>	::=	<draw grid>
		| <draw image>
		| <draw string>
		| <draw line>

		| <paint grid>
		| <move grid object>
<draw grid>	::=	**draw** <identifier> <draw at> **step** <pair expression> <draw size> <draw colour> <draw width>
<draw image>	::=	**draw image** [<identifier>] <expression list> <draw at> <draw size> [**flip** <flip choice>]
<draw string>	::=	**draw string** [<identifier>] <expression list> <draw at>[**font** <font type>] [<expression>] [<font style>]
<draw line>	::=	**draw line** <identifier> **from** <pair expression> **to** <pair expression> [<draw colour>] [<draw width>]
<paint grid>	::=	**paint** <identifier> <draw at> <draw colour>
<draw at>	::=	**at** <pair expression>
<draw size>	::=	**size** <pair expression>
<draw colour>	::=	**colour** <draw colour>
<draw width>	::=	**width** <expression>
<flip choice>	::=	**horizontally** | **vertically** | **diagonally**
<font type>	::=	**arial** | **courier** | **times**
<font style>	::=	**plain** | **bold** | **italic** | **bolditalic**
<move grid object>	::=	**move to** <identifier> <pair expression>
<pair expression>	::=	<expression> **,** <expression>
<draw colour>	::=	**black** | **blue** | **green** | **cyan** | **red** | **magenta** | **yellow** | **white** | **gray** | **darkgray** | **lightgray** | **orange** | **pink**

C.2.13 Event Control Statements

<event control statement>	::=	<on event> | <pause statement>
<on event>	::=	**on** <event> <compound statement>
<event>	::=	**WAITING** | **OBSERVERCONNECTION** | **TURN** | **GROUPTURN** | **REQUESTCONTROL** | **RESERVATION** | **SERVERABORT** | **CLIENTABORT** | **NEWMESSAGE** | **MOUSECLICK** | **MOUSEPRESS** | **MOUSERELEASE** | **TEXTENTERED**
<pause statement>	::=	**wait** <expression>

C.2.14 Communication Statements

<send statement>	::=	**send** <identifier> <expression list> [**to** <send choice>] **;**
<send choice>	::=	**server** | <send list>
<send list>	::=	**all** | **others** | **group**

<receive statement>	::=	**receive** <identifier> [<receiver list>] **;**
<abort session statement>	::=	**abort ;**

C.2.15 Interface Statement

It is used to enable a programmer to include Java codes into JACIE. Below is the syntax rules.

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

C.2.16 Interaction Management Statements

Main parts of specifying these statements have been included in the *Configuration Section* above. Therefore, below are the productions for specifying the supporting statements and other related statements in managing user interaction.

<interaction statement>	::=	<pass turn statement> | <action statement> | <turnset statement> | <request statement> | <critical statement> | <dynamic statement>
<pass turn statement>	::=	**turn pass ;**
<action statement>	::=	**action start** | **end ;**
<request statement>	::=	**turn request** <boolean> **;**
<turnset statement>	::=	**turn set** <set_choice>
<set_choice>	::=	<turnset client> | <turnset group>
<turnset client>	::=	**client** <setting> **;**
<setting>	::=	<identifier> | <integer number>
<turnset group>	::=	**group [groupnumber** <setting>] <choices>
<choices>	::=	<setting> **,** <setting> **;**
<critical statement>	::=	**criticalsection start** | **criticalsection end ;**
<dynamic statement>	::=	<specify protocol>

C.2.17 Interest Management Statements

<interest statement>	::=	<permission statement> | <set statement> | <check statement> | <filtering statement>
<permission statement>	::=	**use** <identifier> **by** <user list> <access list> **;**
<user list>	::=	<send list> | **me** | { <choices> }
<access list>	::=	**[to own** | **not to own]** | **[to read [with password** <string>] | **not to read]** | **[to write [with password** <string>] | **not to write]**
<set statement>	::=	**set [with password** <string>] <identifier> = <expression> **;**
<check statement>	::=	**check** (<expression>) <statement> { **else check** (<expression>) <statement> }

		[**else** <statement>]
<filtering statement>	::=	<filter owner> | <filter user>
<filtering owner>	::=	**filter** <identifier>
		<relational symbols> <double> **;**
<filter user>	::=	**interest set** <identifier> <double>

Bibliography

[1] H.J. Abbink. An Ada–based script language for simulation applications. *ACM Ada Letters*, XVI(5):35–47, 1995.

[2] H. Abdel-Wahab, B. Kvande, O. Kim, and J.P. Favreau. An internet collaborative environment for sharing Java applications. In *Workshop on Future Trends of Distributed Computing Systems*, pages 112–117. IEEE, October 1997.

[3] Siti Z. Z. Abidin, Min Chen, and Phil W. Grant. Managing interaction for multimedia collaboration - through the keyhole of noughts and crosses games. In *International Symposium on Multimedia Software Engineering*, pages 132–135, Miami, Florida, December 2004. IEEE.

[4] Siti Z. Z. Abidin, Min Chen, and Phil W. Grant. Designing interaction protocols using noughts and crosses type games. *Journal of Network and Computer Applications*, 30(2):586–613, April 2007.

[5] Howard Abrams, Kent Watsen, and Michael Zyda. Three tiered interest management for large scale virtual environments. `http://watsen.net/Bamboo/papers/vrst98.pdf`, January 2002.

[6] M.A. Addison and H.W. Thimbleby. Networked interpersonal communications: The convergence of technology...with what? `http://citeseer.ist.psu.edu/302503.html`.

[7] Adit. The history of noughts and crosses. `http://www.adit.co.uk/html`, 1996.

[8] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *International conference on Management of data*, pages 86–97. ACM, 2003.

[9] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1985.

[10] M.L. R. Almendros, M.J. R. Fortiz, and M. G. Megias. A framework for modeling the user interaction with a complex system. In R. M-Diaz and F. Pichler, editors, *EUROCAST 2004*, pages 50–61. LNCS 2809, Springer-Verlag, 2003.

[11] G.R. Andrews. Distributed programming languages. In *Proceedings of the ACM conference*, pages 113–117, 1982.

[12] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[13] Miguel Antunes, Antonio Rito Silva, and Jorge Martins. An abstraction for awareness management in collaborative virtual environments. In *Symposium on Virtual Reality Software and Technology*, pages 33–39. ACM, November 2001.

[14] Reflexive Arcade. Card game bridge. `http://takegame.com/gamblings/pictures/bridge.jpg`.

[15] R. Athauda, N. Kodagoda, J. Wickramaratne, P. Sumathipala, L. Rupasinghe, A. Edirisighe, A. Gamage, and D.D. Silva. Integrating industrial technologies, tools and practices to the IT curriculum: An innovative course with .Net and Java platforms. In *SIGITE'05*. ACM, Oct 2005.

[16] D.L. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A domainspecific language for formbased services. *IEEE Transactions on Software Engineering*, 25(3):334–346, May/June 1999.

[17] J.S. Auerbach, D.F. Bacon, A.P. Goldberg, G.S. Goldszmidt, M.T. Kennedy, A.R. Lowry, J.R. Russell, W. Silverman, R.E. Strom, D.M. Yellin, and S.A. Yemini. High-level language support for programming distributed systems. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 173–196. ACM, Sept 1991.

[18] J.C.M. Baeten, H.M.A. van Beek, and S. Mauw. Specifying internet application with DiCons. In *SAC 2001*, pages 576–584. ACM, 2001.

[19] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Experience with distributed programming in Orca. In *Conference on Computer Languages*, pages 79–89. IEEE, March 1990.

[20] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[21] Henri. E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[22] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. *Computer Languages, Proceedings. International Conference*, pages 9–13, Oct 1988.

[23] Jakob E. Bardram. Temporal coordination: On time and coordination of collaborative activities at a surgical department. *Computer Supported Cooperative Work*, 9(2):157–187, May 2000.

[24] David Barron. *The World of Scripting Languages*. John Wiley & Sons, Ltd, Baffins Lane, Chichester, West Sussex PO19 1UD, England, 2000.

[25] John W. Barrus, Richard C. Waters, and David B. Anderson. Locales: Supporting large multiuser virtual environments. *Computer Graphics and Applications*, 16(6):50–57, November 1996.

[26] Adam Barth. `http://www.adambarth.org/images/bridge.gif`, August 2004.

[27] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-proof playout for centralized and serverless online games. In *INFOCOM*, pages 104–113, 2001.

[28] J. Begole, R.B. Smith, C.A. Struble, and C.A. Shaffer. Resource sharing for replicated synchronous groupware. *IEEE/ACM Transactions on Networking*, 9(6):833–843, Dec 2001.

[29] N.J. Belkin and W.B. Croft. Information filtering and information retrieval: Two sides of the same coin? *Communications of the ACM*, 35(12):29–38, Dec 1992.

[30] Pierfrancesco Bellini and Paolo Nesi. Communicating TILCO: a model for real-time system specification. In *Seventh IEEE International Conference on Engineering of Complex Computer Systems*, pages 4–14, Skovde, Sweden, June 2001. IEEE.

[31] A. Belokosztolszki, K. Moody, and D.M. Eyers. A formal model for hierarchical policy contexts. In *International Workshop on Policies for Distributed Systems and Networks*, pages 127–136. IEEE, June 2004.

[32] Marco E M Di Beneditto and Leliane N de Barros. Using concept hierarchies in knowledge discovery. In A. L. C. Bazzan and S. Labidi, editors, *Advances in Artificial Intelligence*, volume 3171, pages 255–265. Springer-Verlag Berlin Heidelberg, Sept-Oct 2004 2004.

[33] Steve Benford and Lennart Fahlen. A spatial model of interaction in large virtual environments. In *Third European Conference on Computer Supported Cooperative Work (ECSCW'93)*, September 1993.

[34] F. Bergenti and A. Ricci. Three approaches to the coordination of multiagent systems. In *Symposium on Applied Computing (SAC 2002)*, pages 367–372. ACM, 2002.

[35] T. BernersLee, R. Cailliau, A. Luotonen, H.F. Nielsen, and A. Secret. The worldwide web. *Communications of the ACM*, 37(8):76–82, August 1994.

[36] E. Bertino, P.A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3):191–223, August 2001.

[37] Lorenzo Bettini, Rocco De Nicola, Rosario Pugliese, and GianLuigi Ferrari. Interactive mobile agents in X-KLAIM. In *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE)*, pages 110–115. IEEE, June 1998.

[38] M. Bhide, S. Pandey, A. Gupta, and M. Mohania. Dynamic access control framework based on events: A demonstration. In *Conference on Data Engineering (ICDE'03)*, pages 765–767. IEEE, March 2003.

[39] R. Bisiani and A. Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, Aug 1988.

[40] CH. Bouras, E. Giannaka, and TH. Tsiatsos. Virtual collaboration spaces: The EVE community. In *Symposium on Applications and the Internet(SAINT'03)*, pages 48–55. IEEE, January 2003.

[41] J. Boyd. Floor control policies in multiuser applications. In *Companion on Human Factors in Computing Systems*, pages 107–108. ACM, April 1993.

[42] Eva Brandt and Jorn Messeter. Facilitating collaboration through design games. In *conference on Participatory design*, volume 1, pages 121–131. ACM, 2004.

[43] Tom Brinck. Groupware: Introduction. `http://www.usabilityfirst.com/groupware/intro.txl`, 1998.

[44] Paul Brns. Prolog programming a first course. `http://www.scre.ac.uk/personal/pb/prologbook`, May 1999.

[45] Barry Brown and Marek Bell. CSCW at play: 'There' as a collaborative virtual environment. In *CSCW'04*, pages 350–359. ACM, November 2004.

[46] Lee Bu-Sung, Yeo Chai Kiat, Soon Ing Yann, Lee Keok Kee, and Sun Wei. Design and implementation of a Java-based meeting space over internet. In *Multimedia Tools and Applications*, volume 20, pages 179–195, The Netherlands, June 2003. Kluwer Academic Publishers.

[47] Victor Budau and Guy Bernard. Synchronous/asynchronous switch for a dynamic choice of communication model in distributed systems. In *Proceedings on Parallel and Distributed Systems(ICPADS'02)*, pages 97–102. IEEE, Dec 2002.

[48] Wentong Cai, P. Xavier, S. J. Turner, and Bu-Sung Lee. A scalable architecture for supporting interactive games on the internet. In *Workshop on Parallel and Distributed Simulation(PADS'02)*, pages 54–61. IEEE, May 2002.

[49] Paul Callahan. What is the game of life? `http://www.math.com/students/wonders/life/life.html`, 2000.

[50] Cambridge. Dictionary online. `http://dictionary.cambridge.org`, 2006.

[51] Mary Campione, Kathy Walrath, and Alison Huml. *The Java(TM) Tutorial: A Short Course on the Basics.* Pearson Education Corporate Sales Division, New Jersey, USA, third edition, 2001.

[52] K.S. Candan, S. Jajodia, and V.S. Subrahmanian. Secure mediated databases. In *Conference on Data Engineering*, pages 28–37. IEEE, Feb–March 1996.

[53] Michael Capps and Seth Teller. Communication visibility in shared virtual worlds. In *Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, pages 187–192. IEEE, June 1997.

[54] D. Caromel. Toward a method of object–oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, Sept 1993.

[55] S. Carr, J. Mayo, and C.K. Shene. Race conditions: a case study. *Journal of Computing Sciences in Colleges*, 17(1):90–105, Oct 2001.

[56] F.A. Cavazos and J.C.L. Jarquin. A 3-tiered client-server distributed system component-based. In *International symposium on Information and Communication Technologies*, pages 1–6. ACM, 2004.

[57] R. Chandra, A. Gupta, and J.L. Hennessy. COOL: An object–based language for parallel programming. *Computer*, 27(8):13–26, Aug 1994.

[58] Carl K. Chang, Francis Quek, Lie Cai, and Seongwoon Kim. A research on collaboration net. In *Distributed Computing Systems. Proceedings of the sixth IEEE Computer Society Workshop on Future Trend*, pages 228–233. IEEE, October 1997.

[59] Somchai Chatvichienchai, Mizuho Iwaihara, and Yahiko Kambayashi. Secure interoperability between cooperating XML systems by dynamic role translation. In V. Marik et al, editor, *Database and Expert Systems Applications*, volume 2736, pages 866–875. Springer-Verlag Berlin Heidelberg, Oct 2003.

[60] DeQing Chen, Chunqiang Tang, Xiangchuan Chen, Sandhya Dwarkadas, and Micheal L. Scott. Multi-level shared state for distributed systems. In *Proceedings of the International Conference on Parallel Processing (ICPP'02)*. IEEE, 2002.

[61] Gang Chen, Zhonghua Yang, Hao He, and Kiah Mok Goh. Coordinating multi-agents using JavaSpaces. In *Conference on Parallel and Distributed Systems (ICPADS)*, pages 63–68. IEEE, 2002.

[62] T-Shyong Chen, Y-Fang Chung, and C-Sin Tian. A novel key management scheme for dynamic access control in a user hierarchy. In *Conference on Computer Software and Applications Conference*, pages 396–397. IEEE, Sept 2004.

[63] J.D. Choi, B.T. Jang, and C.J. Hwang. Collaborative interactions on 3D display for multi–user game environments. In M. Masoodian et al., editor, *Conference on Computer Human Interaction*, LNCS 3101, pages 81–90. SpringerVerlag Berlin Heidelberg, July 2004.

[64] T.W. Christopher and G. K. Thiruvathukal. *High Performance Java Platform Computing Multithreaded and Networked Programming*. Sun Microsystems/Prentice Hall, Feb 2001.

[65] W.S. Chung and D. McLane. Developing and enhancing a client/server programming for internet course. *Journal of Computing Sciences in Colleges*, 18(2):79–91, Dec 2002.

[66] C.L.A Clarke, P.L Tilker, A.Q-L Tran, K. Harris, and A.S. Cheng. A reliable storage management layer for distributed information retrieval systems. In *International conference on Information and Knowledge Management*, pages 207–215. ACM, 2003.

[67] Gail P. Clement. *Science and Technology on the Internet An Instructional Guide*. Library Solutions Press, Berkeley and San Carlos, California, June 1995.

[68] John Coggeshall. An introduction to PHP. `http://www.onlamp.com/php`, February 2001.

[69] IBM Corporation. Glossary. `http://publib.boulder.ibm.com/infocenter/adiehelp/topic/com.ibm.wsinted%.glossary.doc/topics/glossary.html`, 2000.

[70] Jupitermedia Corporation. Webopedia. `http://www.webopedia.com`.

[71] Jupitermedia Corporation. The JavaScript source. http://javascript.internet.com, 2005–2006.

[72] Microsoft Corporation. COM: Component object model technologies. `http://www.microsoft.com/com/default.mspx`, 2006.

[73] Microsoft Corporation. Microsoft. `http://www.microsoft.com`, 2006.

[74] Microsoft Corporation. Microsoft. `http://www.microsoft.com/windows/netmeeting`, 2006.

[75] Prismic Corporation. Sales forces. `http://www.facetofacemeeting.com/sales.htm`, 2004.

[76] Ramius Corporation. Knowledge management & healthcare. `http://www.ramius.net/healthcare.cfm=148987`, 2006.

[77] Rabelani Dagada. 'where have all the trainers gone?' e-learning strategies and tools in the corporate training environment. In *SAICSIT2004*, pages 194–203. ACM, October 2004.

[78] Gareth W. Daniël and Min Chen. Interaction control protocols for distributed multi-user multicamera environments. In N.Callaos, A.M. Di Sciullo, T. Ohta, and T.K. Liu, editors, *proc. of 7th World Multiconference on Systemic, Cybernetics and Informatics, SCI2003*, volume 1, pages 448–453. International Institute of Informatics and Systemics, July 2003.

[79] J. Daniel, B. Traverson, and V. Vallee. Active COM: an interworking framework for CORBA and DCOM. In *Symposium on Distributed Objects and Applications*, pages 211–222. IEEE, Sept 1999.

[80] A. Davis and D. Zhang. A comparative study of DCOM and SOAP. In *Symposium on Multimedia Software Engineering*, pages 48–55. IEEE, Dec 2002.

[81] J. C. de Oliveira ans Shervin Shirmohammadi and N. D. Georganas. Collaborative virtual environment standards: A performance evaluation. In *International Workshop on Distributed Interactive Simulation and Real-Time Applications*, pages 14–21. IEEE, October 1999.

[82] T.G. de Senna Carneiro and J.N. Cotrim Arabe. Load balancing for distributed virtual reality systems. In *International Symposium on Computer Graphics, Image Processing and Vision*, pages 158–165. IEEE, October 1998.

[83] J. DeFrancoTommarello and F.P. Deek. Collaborative software development: A discussion of problem solving models and groupware technologies. In *Conference on System Sciences*, pages 568–577. IEEE, Jan 2002.

[84] H.M Deitel, P.J. Deitel, J. Listfield, T.R. Neito, C. Yaeger, and M. Zlatkina. *C# HOW TO PROGRAM*. Prentice-Hall, Upper Saddle River, New Jersey 07458, 2002.

[85] Yves Demazeau, Olivier Boissier, and Jean Luc Koning. Using interaction protocols to control vision systems. *Systems, Man, and Cybernetics, 1994*, 2:1616–1621, Oct 1994.

[86] E. Denti and A. Omicini. A coordination infrastructure for agent-based internet applications. In *Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 230–235. IEEE, June 2000.

[87] Micheal Derntl and Renate Motschnig-Pitrik. Patterns for blended, person-centered learning: Strategy, concepts, experiences, and evaluation. In *Symposium on Applied Computing*, pages 916–923. ACM, March 2004.

[88] P. Dewan and R. Choudhary. A high level and flexible framework for implementing multiuser user interface. *ACM Transactions on Information Systems (TOIS)*, 10(4):345–380, Oct 1992.

[89] P. Dewan and H. Shen. Controlling access in multiuser interface. *ACM Transactions on Computer Human Interaction*, 5(1):34–62, March 1998.

[90] Prasun Dewan. An integrated approach to designing and evaluating collaborative applications and infrastructures. *Computer Supported Cooperative Work*, 10(1):75–111, March 2001.

[91] Dawei Ding and Miaoliang Zhu. A model of dynamic interest management: Interaction analysis in collaborative virtual environment. In *Symposium on Virtual Reality Software and Technology*, pages 223–230. ACM, October 2003.

[92] I. Djordjevic, C. Phillips, and T. Dimitrakos. An architecture for dynamic security perimeters of virtual collaborative networks. In *Network Operations and Management Symposium*, volume 1, pages 249–262. IEEE, April 2004.

[93] HansPeter Dommel and J.J. GarciaLunaAceves. Design issues for floor control protocols. `http://www.cse.ucsc.edu/ccrg/publications/peter.spie95.pdf`, 1995.

[94] Hans-Peter Dommel and J.J. Garcia-Luna-Aceves. Group coordination support for synchronous internet collaboration. *IEEE Internet Computing*, 3(2):74–80, April 1999.

[95] H.P. Dommel and J.J. Garcia-Luna-Aceves. A novel group coordination protocol for collaborative multimedia systems. In *International Conference on Systems, Man, and Cybernetics*, volume 2, pages 1225–1230. IEEE, October 1998.

[96] H.Peter Dommel and J.J. Garcia-Luna Aceves. Floor control for activity coordination in networked multimedia applications. `http://www.cse.ucsc.edu/research/ccrg/publications/peter.apcc95.ps.gz`, June 1995.

[97] H.Peter Dommel and J.J Garcia-Luna-Aceves. A coordination architecture for internet groupwork. In *Euromicro conference*, volume 2, pages 183–190. IEEE, September 2000.

[98] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *CSCW*, pages 107–114. ACM, 1992.

[99] Paul Dourish. The parting of the ways: Divergence, data management and collaborative work. In *Fourth European Conference on CSCW(ECSCW)*, pages 215–230, 1995.

[100] Wisconsin DPI. Science glossary of terms. `http://www.dpi.state.wi.us/standards/sciglos.html`, 2005.

[101] R. L. Drechsler and J.M. Mocenigo. The Yoix scripting language and interpreter. `http://www.research.att.com/sw/tools/yoix/doc/reserved`.

[102] Richard L. Drechsler and John M.Mocenigo. The Yoix scripting language as a tool for building web-based systems. In E. Gregori, L. Cherkasova, G. Cugola, F. Panzieri, and G.P. Picco, editors, *Web Engineering and Peer-to-Peer Computing: NETWORKING 2002*, volume 2376, pages 90–103, Pisa, Italy, May 2002. Springer-Verlag Berlin Heidelberg. network language.

[103] K. Drira, T. Villemur, V. Baudin, M. Diaz, and L. du Cnrs. A multiparadigm layered architecture for synchronous distance learning. In *Euromicro Conference*, pages 158–165. IEEE, Sept 2000.

[104] D. D'Souza, J.A. Thom, and J. Zobel. Collection selection for managed distributed document databases. *Information Processing & Management*, 40(3):527–546, May 2004.

[105] Bruce Eckel. *Thinking in Java*. Prentice-Hall, 3rd edition, December 2002.

[106] Stephen A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, April 2003.

[107] Abdulmotaleb El-Saddik, Shervin Shirmohammadi, Nicolas D. Georganas, and Ralf Steinmetz. Jasmine: Java application sharing in multiuser interactive environments. In *Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 214–226, 2000.

[108] C. Ellison and S. Dohrmann. Public–key support for group collaboration. *ACM Transactions on Information and System Security*, 6(4):547–565, Nov 2003.

[109] W.F. Elsworth and M.B.A. Parkes. Automated compiler construction based on top–down syntax analysis and attribute evaluation. *ACM SIGPLAN Notices*, 25(8):37–42, Aug 1990.

[110] Wolfgang Emmerich. Distributed component technologies and their software engineering implications. In *International Conference on Software Engineering*, pages 537–546. IEEE, May 2002.

[111] Ericsson. ERLANG. `http://www.erlang.org`.

[112] D.M. Etter. *Structured FORTRAN 77 for Engineers and Scientists*. Addison Wesley, Menlo PArk, CA, 5th edition, 1997.

[113] S.G. Fantar, S.M. Gammar, and F. Kamoun. Using SIP for floor control in a videoconference. In *Int. Conference on Information Technology Based Higher Education and Training*, pages 274–277. IEEE, June 2004.

[114] D. G. Feitelson. On the scalability of centralized control. In *Symposium on Parallel and Distributed Processing*. IEEE, April 2005.

[115] David Ferraiolo and Richard Kuhn. Role-based access control. In *Proceedingd of 15th National Computer Security Conference*, pages 554–563, 1992.

[116] F.G. Fiamingo. Unix system administration. `http://wks.uts.ohio\-state.edu/sysadm\-course/html/sysadm\-1.html`, August 1996.

[117] Robert E. Filman and Daniel P. Friedman. Shared variables. In *COORDINATED COMPUTING: Tools and Techniques for distributed software*, chapter 6, pages 57–72. McGraw-Hill Book Company, 1984.

[118] Fraunhofer FIT and OrbiTeam Software GmbH. BSCW. `http://bscw.fit.fraunhofer.de`, 1995–2005.

[119] Message Passing Interface Forum. MPI: Overview and goals. `http://www.mpi-forum.org/docs/mpi-11-html/node2.html`, August 1997.

[120] Piero Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(3):227–263, September 1999.

[121] Lidia Fuentes and Jose M.Troya. A JAVA FRAMEWORK FOR WEB-BASED multimedia and collaborative applications. *IEEE Internet Computing*, pages 55–64, March-April 1999.

[122] Masaru Fukushi and Susumu Horiguchi. A self-reconfiguration hardware architecture for mesh arrays using single/double vertical track switches. *IEEE Transactions on Instrumentation and Measurement*, 53(2):357–367, April 2004.

[123] U. Gall and F.J. Hauck. Promondia: a Java-based framework for real-time group communication in the web. *Computer Networks and ISDN Systems*, 29(9):917–926, Sept 1997.

[124] Pedro Garcia, Oriol Montala, Carles Pairot, Roberto Rallo, and Antonio Gomez Skarmeta. MOVE: Component groupware foundations for collaborative virtual environments. In *COLLABORATIVE VIRTUAL ENVIRONMENTS*. ACM Society, September 2002. http://ants.etse.urv.es/move.

[125] S.L. Garfinkel, D. Margrave, J.I. Schiller, E. Nordlander, and R.C. Miller. Email and security: How to make secure email easier to use. In *Conference on Human Factors in Computing Systems*, pages 701–710. SIGCHI, April 2005.

[126] P. Gaztin, B. Lerman, and M. Zeitoun. Distributed games and distributed control for asynchronous systems. In *Latin American Symposium on Theoretical Informatics*, volume 2976, pages 455–465. Springer(LNCS), April 2004.

[127] C.K. Georgiadis, I. Mavridis, G. Pangalos, and R.K. Thomas. Flexible team-based access control using contexts. In *Symposium on Access Control Model and Technologies*, pages 21–27. ACM, May 2001.

[128] W. Geyer, J. Vogel, Li-Te Cheng, and M. Muller. Supporting activity-centric collaboration through peer-to-peer shared objects. In *GROUP'03*, pages 115–124. ACM, November 2003.

[129] A. Geyer-Schulz and T. Kolarik. Distributed computing with apl. In *International Conference on APL APL '92*, pages 60–69. ACM, July 1992.

[130] J. Paul Gibson. A noughts and crosses Java applet to teach programming to primary school children. In *PPPJ 2003*, pages 85–88. ACM, 2003.

[131] H. Gilbert. Introduction to TCP/IP. `http://www.yale.edu/pclt/COMM/TCPIP.HTM`, Feb 1995.

[132] K. Goldberg and B. Chen. Collaborative control of robot motion: Robustness to error. In *Int. Conference on Robots and Systems*. IEEE/RSJ, 2001.

[133] Andrzej Goscinski. *DISTRIBUTED OPERATING SYSTEMS: The Logical Design*. Addison-Wesley Publishing Company, 1991.

[134] Chris Greenhalgh and Steve Benford. MASSIVE: a distributed virtual reality system incorporating spatial trading. In *conference on Distributed Computing Systems*, pages 27–34. IEEE, May–June 1995.

[135] Chris Greenhalgh, Jim Purbrick, and Dave Snowdon. Inside MASSIVE-3: Flexible support for data consistency and world structuring. In Elizabeth Churchill and Martin Reddy, editors, *ACM conference on COLLABORATIVE VIRTUAL ENVIRONMENTS*, pages 119–127. ACM SIGCHI, ACM SIGGROUP and ACM SIGGRAPH, ACM, September 2000.

[136] Object Management Group. CORBA BASICS. `http://www.omg.org`.

[137] Postgre SQL Global Development Group. PostgreSQL 8.1.4 documentation. `http://www.postgresql.org/docs/8.1/static/sql.html`.

[138] O. Hagsand. Interactive multiuser VEs in the DIVE system. In *IEEE Multimedia*, pages 30–39, 1996.

[139] Abdul S. Haji-Ismail, Min Chen, Phil W.Grant, and Mark Kiddell. JACIE–an authoring language for rapid prototyping net-centric, multimedia and collaborative applications. *Annals of Software Engineering*, 12:47–75, December 2001.

[140] Abdul Samad Haji-Ismail. *JACIE–A Scripting language for internet-based multimedia collaborative applications*. PhD thesis, Department of Computer Science, University of Wales Swansea, UK, 2001.

[141] Fred Halsall. *Data Communications, Computer Networks and Open Systems*. Addison Wesley, fourth edition edition, 1996.

[142] Fred Halsall. *Multimedia Communications*. Addison Wesley, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 2001.

[143] M. Handley, I. Wakeman, and J. Crowcroft. CCCP: A scalable base for building conference control applications. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 275–287. ACM, 1995.

[144] B. Hansen. Monitors and concurrent Pascal: a personal history. In *conference on History of programming languages*, pages 1–35. ACM, 1993.

[145] M. Hanus. Highlevel server side Web scripting in Curry. In *PADL'01*, LNCS 1990, pages 76+. Springer Verlag, 2001.

[146] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating system. *Communications of the ACM*, 19(8):461–471, August 1976.

[147] Frank Havemann. Collaboration and productivity of West-German biomedical researchers. `http://citeseer.ist.psu.edu/havemann02collaboration.html`, 2002.

[148] Micheal S. H. Heng and Aldo de Moor. From Habermas's communicative theory to practice on the internet. *Information Systems Journal*, 13(4):331–352, October 2003.

[149] Jean-Luc Henry. A k-nearest neighbour method for managing the evolution of a learning base. In *Conferencr on Computational Intelligence and Multimedia Applications*, pages 357–361. IEEE, Nov 2001.

[150] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Database replication using epidemic communication. In *6th International Euro-Par Conference*, pages 427–434. LNCS 1900, Springer, Aug/Sep 2000.

[151] S. Horrocks, N. Rahmati, and T. Robbins-Jones. The development and use of a framework for categorising acts of collaborative work. In *Conference on System Sciences*, page 13pp. IEEE, Jan 1999.

[152] Cay S. Horstmann and Gary Cornell. *Core JAVA Volume I-Fundamentals*, volume 1. Sun Microsystems Press, Palo Alto, California, 2001.

[153] Mojtaba Hosseini, Steve Pettifer, and Nicolas D.Georganas. Visibility-based interest management in collaborative virtual environments. In *COLLABORATIVE VIRTUAL ENVIRONMENTS*, pages 143–144. ACM Society, September 2002.

[154] H. Hua, L.D. Brown, and Chunyu Gao. Scape: Supporting stereoscopic collaboration in augmented and projective environments. *IEEE Computer Graphics and Applications*, 24(1):66–75, Jan-Feb 2004.

[155] Z. Huang, A. Eliens, and C. Visser. Implementation of a scripting language for VRML/X3D–based embodied agents. In *Conference on 3D Web Technology*, pages 91–100. ACM, 2003.

[156] Luke Hunsberger. Distributing the control of a temporal network among multiple agents. In *Conference on Autonomous Agents and Multiagents Systems (AAMAS'3)*, pages 899–906. ACM, July 2003.

[157] IBM. IBM REXX family. http://www306.ibm.com/ software/awdtools/rexx/.

[158] IBM. Lotus software. `http://www.lotus.com`.

[159] IFAC. `http://media.ici.ro/academia/ifac/ifac_tc54.htm`.

[160] ActiveState Software Inc. Tcl developer xchange. `http://www.tcl.tk`.

[161] Infinite Software Solutions Inc. VBScript. `http://www.devguru.com/Technologies/vbscript/QuickRef/vbscript_intro.h%tml`, 1999–2005.

[162] Sun Microsystems Inc. Java technology. `http://java.sun.com`.

[163] Yahoo Inc. Yahoo games. `http://games.yahoo.com/games/front`.

[164] Rahat Iqbal, Anne James, and Richard Gatward. A collaborative platform for heterogeneous CSCW systems: Case study of academic applications. In *International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, pages 462–467. IEEE, Sept 2003.

[165] iQTOYS. 4D tic-tac-toe. `http://www.iqtoys.au.com`.

[166] K. Isbister, H. Nakanishi, T. Ishida, and C. Nass. Helper agent: Designing an assistant for human-human interaction in a virtual meeting space. In *Conference on Human Factors in Computing Systems*, pages 57–64. ACM, April 2000.

[167] H. Ishikawa, Y. Yamane, Y. Izumida, and N. Kawato. An object–oriented database system Jasmine: Implementation, application and extension. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):285–304, April 1996.

[168] iVillage. What is your emotional intelligence quotient? `http://quiz.ivillage.com/health/tests/eqtest2.htm`, 1995–2006.

[169] L.S. Jackson and E. Grossman. Integration of synchronous and asynchronous collaboration activities. *ACM Computing Surveys*, 31(2es), 1999.

[170] James B D Joshi, Elisa Bertino, and Arif Ghafoor. Temporal hierarchies and inheritance semantics for GTRBAC. *Proceedings of the Seventh ACM symposium on access control models and techniques*, pages 74–83, June 2002.

[171] J.B.D. Joshi, R. Bhatti, E. Bertino, and A. Ghafoor. Access-control language for multidomain environments. *IEEE Internet Computing*, 8(6):40–50, Nov–Dec 2004.

[172] C. Joslin, T. D. Giacomo, and N. Magnenat-Thalmann. Collaborative virtual environments: From birth to standardization. *IEEE Communications Magazine*, 42(4):28–33, April 2004.

[173] C. Joslin, T. Molet, and N. Magnenat-Thalmann. Advanced realtime collaboration over the internet. In *Symposium on Virtual Reality Software and Technology*, pages 25–32. ACM, 2000.

[174] R. Jota, J. Martins, A. Rito-Silva, and J. Pereira. Experimenting with a flexible awareness management abstraction for virtual collaboration spaces. In *Symposium on Applications and the Internet(SAINT'03)*, pages 56–64. IEEE, January 2003.

[175] G.S. Novak Jr. Vocabulary. `http://www.cs.utexas.edu/users/novak/cs307vocab.html`, 2000.

[176] M.F. Kaashoek, R.V. Renesse, H.V. Steveren, and A.S. Tanenbaum. FLIP: An internetwork protocol for supporting distributed systems. *Transactions on Computer Systems*, 11(1):73–106, Feb 1993.

[177] Nadia Kausar and Jon Crowcroft. End to end reliable multicast transport protocol requirements for collaborative multimedia systems. In *Reliable Distributed Systems*, pages 425–430. IEEE, October 1998.

[178] Ronald L.G. Keith. *The Matic System or Bidding By Numbers for Contract Bridge*. Edurec Ltd, Spring Wood Lane, Burghfield Common, Reading, RG7 3DS, 1996.

[179] Peter J. Keleher. A high-level abstraction of shared accesses. *ACM Transaction on Computer Systems*, 18(1):1–36, Feb 2000.

[180] M.E. Khan, Ray Paul, Ishfaq Ahmed, and Arif Ghafoor. Intensive data management in parallel systems: A survey. *Distributed and Parallel Databases*, 7(4):383–414, October 1999.

[181] Hyung-Jun Kim, So-Hyun Ryu, Young-Je Woo, Yong won Kwon, and Chang-Sung Jeong. COVE: A design and implementation of collaborative object-oriented visualization environment. In *Groupware: Design, Implementation and Use, CRIWG 2003, LNCS*, volume 2806, pages 42–57. Springer-Verlag Berlin Heidelberg, September 2003.

[182] John Leslie King. Centralized versus decentralized computing: Organizational considerations and management options. *ACM Computing Surveys*, 15(4):319–349, Dec 1983.

[183] Elliot B. Koffman. *Problem Solving and Structured Programming in PASCAL*. AddisonWesley, Reading, Mass, 1981.

[184] Thomas Kolarik. Extending the two-partner shared variable protocol to n partners. In *Proceedings of the international conference on APL*, pages 124–133. ACM Press, 1993.

[185] F. Kon, R.H. Campbell, M.D. Mickunas, K. Nahrstedt, and F.J. Ballesteros. 2K: a distributed operating system for dynamic heterogeneous environments. In *Symposium on HighPerformance Distributed Computing*, pages 201–208. IEEE, Aug 2000.

[186] Jean-Luc Koning and Marc-Philippe Huget. Interaction protocol design: Application to an agent-based teleteaching project. In *Conference on Cognitive Informatics(ICCI'03)*, pages 8pp–. IEEE, 2003.

[187] B.I. Kumova. Software design concept of a distributed simulation kernel. In *Conference on Parallel, Distributed and Network–Based Processing*, pages 34–39. IEEE, Feb 2004.

[188] B.I. Kumova. Dynamically adaptive partition–based data distribution management. In *Principles of Advanced and Distributed Simulation*, pages 292–300. IEEE, June 2005.

[189] K.B. Lassen. Colored petri nets. `http://www.daimi.au.dk/CPnets/intro`, Dec 2002.

[190] Chungnan Lee, Chuanwen Chiang, and Minfong Horng. Collaborative web computing environment: An infrastructure for scientific computation. *IEEE Internet Computing*, March-April:27–35, April 2000. access control or load distribution.

[191] Dongman Lee, Mingyu Lim, and Seunghyun Han. ATLAS: a scalable network framework for distributed virtual environments. In *ACM Conference on COLLABORATIVE VIRTUAL ENVIRONMENTS*. ACM, September 2002.

[192] Gunhee Lee, Hongjin Yeh, Wonil Kim, and Dong-Kyoo Kim. Web security using distributed role hierarchy. In M. Li et al, editor, *Grid and Cooperative Computing*, volume 3032, pages 1087–1090. Springer-Verlag Berlin Heidelberg, December 2003 2004.

[193] J.H. Lee, A. Prakash, T. Jaeger, and G. Wu. Supporting multi–user, multi–applet workspaces in CBE. In *CSCW'96*, pages 344–353. ACM, 1996.

[194] L.Q. Lee and A. Lumsdaine. The generic message passing framework. In *Symposium on Parallel and Distributed Processing*, page 10pp. IEEE, April 2003.

[195] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 479–492, Charleston, South Carolina, 1993.

[196] H. Lei, D. Chakraborty, H. Chang, M.J. Dikun, T. Heath, J.S. Li, N. Nayak, and Y. Patnaik. Contextual collaboration: Platform and applications. In *Conference on Services Computing*, pages 197–206. IEEE, Sept 2004.

[197] U. Leonhardt and J. Magee. Towards a general location service for mobile environments. In *Workshop on Services in Distributed and Networked Environments*, pages 43–50. IEEE, June 1996.

[198] U. Leonhardt and J. Magee. Security considerations for a distributed location service. *Journal Of Network and Systems Management*, 6(1):51–70, March 1998.

[199] P. Li, B. Ravindran, H. Cho, and E.D. Jensen. Scheduling distributed real-time threads in Tempus middleware. In *International Conference on Parallel and Distributed Systems*, pages 187–194. IEEE, July 2004.

[200] Wanjiun Liao and Victor O.K. Li. Synchronization of distributed multimedia systems with user interactions. *Multimedia Systems*, 6(3):196–205, May 1998.

[201] Oliver Liechti. Awareness and the www: an overview. *SIGGROUP Bulletin*, 21(3):3–11, December 2000.

[202] J.M. Linebarger, C.D. Janneck, and G.D. Kessler. Shared Simple Virtual Environment: An object-oriented framework for highly interactive group collaboration. In *International Symposium on Distributed Simulation and Real-Time Applications*, pages 170–180. IEEE, October 2003.

[203] Chen Ling, Chen Gen-Cai, and Chen Chun. A knowledge-based adaptive message filtering technique for collaborative virtual environment. In *Conference on Signal Processing*, volume 1, pages 266–271. IEEE, Aug 2002.

[204] Antonio Liotta, George Pavlou, and Graham Knight. A self adaptable agent system for efficient information gathering. In S. Pierre and R. Glitho, editors, *MATA 2001*, volume LNCS 2164, pages 139–152. Springer-Verlag Berlin Heidelberg, 2001. agent.

[205] Chia-Hao Liu, Chen-Hsing Wen, and Hsing-Lung Chen. Tracking-needless grouping: An efficient and scalable grouping scheme in networked virtual environments. In *Consumer Communication and Networking Conference (CCNC 2004)*, pages 477–482. IEEE, Jan 2004.

[206] E.S. Liu, M.K. Yip, and G. Yu. Scalable interest management for multidimensional routing spaces. In *VRST'05*, pages 82–85. ACM, Nov 2005.

[207] Z. Liu, X. Du, and N. Ishii. Integrating databases in internet. In *Conference on Knowledge-Based Intelligent Electronic Systems*, pages 381–385. IEEE, 1998.

[208] Enginuity LLC. Three stones board game. `http://www.educational-child-toy.com`.

[209] Veryard Projects Ltd. Notions of software componentry. `http://www.users.globalnet.co.uk/~rxv/CBDmain/cbdnotions.htm`, December 2001.

[210] Zakaria Maamar. Commerce, e-commerce, and m-commerce: What comes next? *Communications of the ACM*, 46(12):251–257, December 2003.

[211] Micheal R. Macedonia, Micheal J. Zyda, David R. Pratt, Donald P. Brutzman, and Paul T. Barham. Exploiting reality with multicast groups: A network architecture for large-scale virtual environments. In *Virtual Reality Annual International Symposium*, pages 2–10. IEEE, March 1999.

[212] Isabel Machado, Rui Prada, and Ana Paiva. Bringing drama into a virtual stage. In Elizabeth Churchill and Martin Reddy, editors, *ACM conference on COLLABORATIVE VIRTUAL ENVIRONMENT*, pages 111–117. ACM SIGCHI, ACM SIGGROUP, ACM SIGGRAPH, ACM, September 2000.

[213] David A. Maluf and Peter B. Tran. Articulation management for intelligent integration of information. *IEEE Transactions on Systems, Man and Cybernetics*, 31(4):485–496, Nov 2001.

[214] Tony Manninen. Conceptual, communicative and pragmatic aspects of interaction forms - rich interaction model for collaborative virtual environments. In *International Conference on Computer Animation and Social Agents(CASA'03)*, pages 168–173. IEEE, May 2003.

[215] G. Mark, J.M. Haake, and N.A. Streitz. Hypermedia structures and the division of labor in meeting room collaboration. In *Computer Supported Cooperative Work*, pages 170–179. ACM, 1996.

[216] Dave Marshall. Remote procedure calls (RPC). `http://www.cs.cf.ac.uk/Dave/C/node33.html`, 1999.

[217] Michal Masa and Jiri Zara. Generalized interest management in virtual environments. In *Proceedings on Collaborative Virtual Environment*, pages 149–150. ACM, Sep–Oct 2002.

[218] M.L. Massie, B.N. Chun, and D.E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

[219] A.K. Mattas, I.K. Mavridis, and G.I. Pangalos. Towards dynamically administered role-based access control. In *Workshop on Database and Expert Systems Applications (DEXA'03)*, pages 494–498. IEEE, Sept 2003.

[220] Martin Mauve. TeCo3D – sharing interactive and dynamic 3D models. In *Multimedia Tools and Applications*, volume 20, pages 283–304, The Netherlands, August 2003. Kluwer Academic Publishers.

[221] P F McKee, I W Marshall, and I D Henning. Research directions in distributed systems. *BT Technology Journal*, 17(2):137–144, April 1999.

[222] John McLeod. Contract bridge. `http://www.pagat.com//boston/bridge.html`, January 2002.

[223] MediaWiki. Wikipedia encyclopedia. `http://en.wikipedia.org/wiki/Software_agent`, 2006.

[224] P. Messina, S. Brunett, D. Davis, T. Gottschalk, D. Curkendall, L. Ekroot, and H. Siegel. Distributed interactive simulation for synthetic forces. In *Heterogeneous Computing Workshop (HCW'97)*, pages 112–119. IEEE, April 1997.

[225] B.R. Millard, D.S. Miller, and C. Wu. Support for Ada intertask communication in a message-based distributed operating system. In *Conference on Computers and Communications*, pages 219–225. IEEE, March 1991.

[226] I. Mirbel, B. Pernici, T. Sellis, S. Tserkezoglou, and M. Vazirgiannis. Checking the temporal integrity of interactive multimedia documents. *The VLDB Journal*, 9(2):111–130, July 2000.

[227] A. L. Moran, J. Favela, A. M. Martinez-Enriquez, and D. Decouchant. Before getting there: Potentials and actual collaboration. In J.M Haake and J.A Pino, editors, *Groupware: Design, Implementation and Use*, pages 147–167. LNCS 2440, Springer-Verlag, 2002.

[228] G. Morgan, F. Lu, and K. Storey. Interest management middleware for networked games. In *Symposium on Interactive 3D Graphics and Games*, pages 57–64. ACM, April 2005.

[229] Graham Morgan and Fengyun Lu. Predictive interest management: An approach to managing message dissemination for distributed virtual environments. In *Proceedings of the First International Workshop on Interactive Rich Media Content Production: Architectures, Technologies, Applications, Tools (Richmedia2003)*, 2003.

[230] Katherine L. Morse. Interest management in large-scale distributed simulations. `http://citeseer.ist.psu.edu/morse96interest.html`, 1996.

[231] K.L. Morse and M. Zyda. Multicast grouping for data distribution management. report, ESS and MAG, 2550 Fifth Avenue, Suite 724, San Diego, CA, USA, 2001.

[232] J.P. Munson and P. Dewan. Sync: a Java framework for mobile collaborative applications. *Computer*, 30(6):59–66, June 1997.

[233] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[234] N-Tier.com. Client/server and the n–tier model of distributed computing. `http://n\-tier.com/articles/csovervw.html`, 1999.

[235] J.C. Navas and M. Wynblatt. The network is the database: Data management for highly distributed systems. In *International conference on Management of data*, pages 544–551. ACM, May 2001.

[236] Zsolt Nemeth and Vaidy Sunderam. Characterizing grids: Attributes, definitions, and formalisms. *Journal of Grid Computing*, 1(1):9–23, 2003.

[237] T. Niemi, M. Junkkari, K. Jarvelin, and S. Viita. Advanced query language for manipulating complex entities. *Information Processing & Management*, 40(6):869–889, November 2004.

[238] Anton Nijholt. Meetings in the virtuality continuum: Send your avatar. In *Conference on Cyberworlds (CW'05)*, pages 75–82. IEEE, Nov 2005.

[239] Inc Novell. Groupwise. `http://www.novell.com/products/groupwise`, 2006.

[240] Marco Oey, Keon Langendoen, and Henri E. Bal. Comparing kernel-space and user-space communication protocols on Amoeba. In *Conference on Distributed Computing Systems*, pages 238–245. IEEE, May–June 1995.

[241] EECS(University of Michigan). The DistView collaboratory toolkit. `http://www.eecs.umich.edu/distview/userdocs.html`, 1998.

[242] Tetsuro Ogi, Takura Kayahara, Toshio Yamada, and Michitaka Hirose. MVL toolkit: Software library for constructing an immersive shared virtual world. In *Virtual Reality*, pages 249–250. IEEE, March 2004.

[243] Masaya Okada, Hiroyuki Tarumi, and Tetsuhiko Yoshimura. Collaborative environment education using distributed virtual environment accessible from real and virtual worlds. *Applied Computing Review*, 9(1):15–21, April 2001.

[244] Zan Oliphant. Programming Netscape plug-ins. `http://docs.rinet.ru/Plugi/appc.htm`, 1996.

[245] Emil Ong. MPI Ruby: Scripting in a parallel environment. *Computing in Science & Engineering*, 4(4):78–82, July/August 2002.

[246] J.K. Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, March 1998.

[247] C.C. Pan, P. Mitra, and P. Liu. Semantic access control for information interoperation. In *SACMAT'06*, pages 237–246. ACM, June 2006.

[248] Ted Panitz. A definition of collaborative vs cooperative learning. `http://www.lgu.ac.uk/deliberations/collab.learning/panitz2.html`, 1996.

[249] Marcin Paprzycki and Janusz Zalewski. Ada in distributed systems: An overview. *ACM Ada Letters*, XVII(7):67–81, Mar/Apr 1997.

[250] parentsays. `http://www.parentsays.com/images/benefit1.gif`.

[251] Sungju Park, Dongman Lee, Mingyu Lim, and Chansu Yu. Scalable data management using user-based caching and prefetching in distributed virtual environments. *Virtual Reality Software and technology*, pages 121–126, November 2001.

[252] M. Patino-Martinez, R. Jimenez-Peris, and S. Arevalo. Synchronizing group transactions with rendezvous in a distributed ada environment. In *Symposium on Applied Computing*, pages 2–9. ACM, February 1998.

[253] David Pinelle and Carl Gutwin. Task analysis for groupware usability evaluation: Modeling shared-workspace tasks with the mechanics of collaboration. *ACM Transactions on Computer-Human Interaction*, 10(4):281–311, December 2003.

[254] Claudio S. Pinhanez and Aaron F. Bobick. Interval scripts: a programming paradigm for interactive environments and agents. *Personal and Ubiquitos Computing*, 7(1):1–21, May 2003.

[255] Pogo. `http://www.pogo.com`.

[256] R. Prada and A. Paiva. Intelligent virtual agents in collaborative scenarios. In T. Panaylotopoulos *et al.*, editor, *Int. Working Conference on Intelligent Virtual Agents*, LNAI 3661, pages 317–328, Sept 2005.

[257] A. Prakash and H.S. Shim. DistView: Support for building efficient collaborative applications using replicated objects. In *Conference on CSCW*, pages 153–164. ACM, Oct 1994.

[258] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. A survey of distributed shared memory systems. In Trevor N. Mudge and Bruce D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 74–84, Los Alamitos, CA, USA, jan 1995. IEEE Computer Society Press.

[259] Weizhong Qiang, Hai Jin, Xuanhua Shi, and Deqing Zou. A novel VO-based access control model for grid. In H. Jin, Y. Pan, N. Xiao, and J. Sun, editors, *Grid and Cooperative Computing*, volume 3251, pages 293–300. Springer-Verlag Berlin Heidelberg, Oct 2004.

[260] Ruibiao Qiu, Fred Kuhns, and Jerome R. Cox. A conference control protocol for highly interactive videoconferencing. In *Global Telecommunications Conference, GLOBECOM '02*, volume 2, pages 2021–2025. IEEE, Nov 2002.

[261] C. Qu and W. Nejdl. Constructing a web-based asynchronous and synchronous collaboration environment using WebDAV and Lotus SameTime. In *SIGUCCS '01*, pages 142–149. ACM, Oct 2001.

[262] D. Radosevic and B. Klieek. Development of a higherlevel multimedia scripting language. In *Information Technology Interface*, pages 201–208. IEEE, 2001.

[263] R.R. Raje, D. Zhu, S. Mukhopadhyay, L. Tang, and M. Palakal. COBioSIFTER – A CORBA-based distributed multi-agent biological information management system. *Cluster Computing*, 7(4):373–389, October 2004.

[264] P.K. Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):154–169, Feb 2004.

[265] S.P. Reiss. A component model for internetscale applications. In *ASE'05*, pages 34–43. ACM, Nov 2005.

[266] Mathias W. Richter. Iava: yet another interpreter for scripting within the Java platform. *Software-Practice And Experience*, 30:81–106, 2000.

[267] Ran Rinat and Scott Smith. Modular internet programming with cells. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming:16th European Conference*, pages 257–280, Malaga, Spain, June 2002. Springer-Verlag Berlin Heidelberg. LNCS 2374.

[268] M. Ringel, K. Ryall, C. Shen, C. Forlines, and F. Vernier. Release, relocate, reorient, resize: Fluid techniques for documents sharing on multi-user interactive tables. In *CHI 2004*, pages 1441–1444. ACM, April 2004.

[269] Kirrily Skud Robert. Perl documentation. `http://www.perldoc.com/perl5.8.0/pod/perlintro.html`.

[270] K.W. Ross and J.F. Kurose. Connectionless transport: UDP. `http://www\-net.cs.umass.edu/kurose/transport/UDP.html`, 1996–2000.

[271] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. `http://www.mozart-oz.org/papers`, 2003.

[272] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5):804–851, September 1997.

[273] R. P. Saldana, W. C. Tabares, and W.E. S. Yu. Parallel implementations of cellular automata algorithms on the agila high performance computing system. In *International Symposium on Parallel Architectures, Algorithms and Networks*, pages 110–116. IEEE, May 2002.

[274] Ravi Sandhu and Jaehong Park. Usage control: A vision for next generation access control. In V. Gorodetsky et al., editor, *MMM-ACNS 2003*, pages 17–31. Springer, 2003.

[275] R.S. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating schemes. *Journal of ACM*, 35(2):404–432, Apr 1988.

[276] R.S. Sandhu. The typed access matrix model. In *Symposium on Security and Privacy*, pages 122–136. IEEE, 1992.

[277] Manish Sarkar and B. Yegnanarayana. Rough-fuzzy set theoritic approach to evaluate the importance of input features in classification. In *International Conference on Neural Networks*, volume 1, pages 438–443. IEEE, June 1997.

[278] Herbert Schildt. *Modula2 Made Easy*. McGrawHill, Berkeley, California, 1986.

[279] K. A. Schneider and James R. Cordy. Abstract user interfaces: A model and notation to support plasticity in interactive systems. In C. Johnson, editor, *DSV-IS 2001*, pages 28–49. LNCS 2220, Springer-Verlag, 2001.

[280] Jennifer M. Schopf. Grids: The top ten questions. *Scientific Programming: Special issue on Grid Computing*, 10(2):103–111, August 2002.

[281] C. Schuckmann, L. Kirchner, J. Schummer, and J.M. Haake. Designing object-oriented synchronous groupware with COAST. In *CSCW*, pages 30–38. ACM, 1996.

[282] H. Schuldt, H. Schek, and M. Tresch. Coordination in CIM: Bringing database functionality to application systems. `http://citeseer.ist.psu.edu/schuldt98coordination.html`, 1998.

[283] Timothy K. Shih, Lawrence Y.Deng, I-Chun Liao, Chun-Hung Huang, and Rong-Chi Chang. Using the floor control mechanism in distributed multimedia presentation system. In *Distributed Computing Systems Workshop*, pages 337–342. IEEE, April 2001.

[284] S. Shirmohammadi and N.D. Georganas. JETS:a Java–enabled telecollaboration system. In *Multimedia Computing and Systems*, pages 541–547. IEEE, June 1997.

[285] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, January 1991.

[286] Jon Siegel. CORBA BASICS. `http://www.omg.org/gettingstarted/corbafaq.html`, May 2002.

[287] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, New York, 6th edition, 2002.

[288] Hugo Simpson. Mascot real time networks in distributed system design. In *IEE Colloquium on Building Distributed Systems*, pages 1/1–1/10. IEEE, Nov 1990.

[289] A. Sinha. Client/server computing. *Communications of the ACM*, 35(7):77–98, July 1992.

[290] Smalltalk.org. Smalltalk. `http://www.smalltalk.org/main`.

[291] M.L. Smith, R.J. Parsons, and C.E. Hughes. View-centric reasoning for Linda and tuple space computation. In *IEE Proceedings on Software*, volume 150, pages 71–83. IEEE, April 2003.

[292] Allan Snavely, Greg Chun, Henri Casanova, Rob F. Van der Wijngaart, and M. A. Frumkin. Benchmarks for grid computing: A review of ongoing efforts and future

directions. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):27–32, March 2003.

[293] P.G. Soares. On remote procedure call. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 215–267. ACM, 1992.

[294] Eiffel Software. Eiffel software. `http://www.eiffel.com`.

[295] Sandy Knoll Software. Tic-tac-toe. `http://secure.downeast.com`, May 2003.

[296] Diane H. Sonnenwald, Mary C. Whitton, and Kelly L. Maglaughlin. Evaluating a scientific collaboratory: Result of a controlled experiment. *ACM Transactions on Computer-Human Interaction*, 10(2):150–176, June 2003.

[297] C. Stapleton, C. Hughes, M. Moshell, P. Micikevicius, and M. Altman. Applying mixed reality to entertainment. *Computer*, 35(12):122–124, Dec 2002.

[298] A. Di Stefano, G. Pappalardo, C. Santoro, and E. Tramontana. SHARK, a multi-agent system to suppport document sharing and promote collaboration. In *International Workshop on Hot Topics in Peer-to-Peer Systems*, pages 86–93. IEEE, October 2004.

[299] Karl E. Steiner and Tom Moher. Encouraging task-related dialog in 2D and 3D shared narrative workspaces. In Elizabeth Churchill and Martin Reddy, editors, *ACM conference on COLLABORATIVE VIRTUAL ENVIRONMENT*, pages 39–46. ACM SIGCHI, ACM SIGGROUP, ACM SIGGRAPH, ACM, September 2002.

[300] R.C. Steinke and G.J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, Sept 2004.

[301] M. Young Sung and D. Hyung Lee. A Java-based collaborative authoring system for multimedia presentation. In K. Aizawa, Y. Nakamura, and S. Satoh, editors, *Advances in Multimedia Information Processing*, volume LNCS 3332, pages 96–103. Springer-Verlag, Nov–Dec 2004.

[302] U.J. Sung, J.H. Yang, and K.Y. Wohn. Concurrency control in CIAO. In *Virtual Reality '99*. IEEE, 1999.

[303] R. Sureswaran, S. Noori, R. Budiarto, and S. Rao. Scalable and reliable multisession document sharing system. In *Conference on Information and Communication Technologies: From Theory to Applications*, pages 613–614. IEEE, April 2004.

[304] H. Suzuki and R. Huang. Virtual realtime 3D object sharing for supporting distance education and training. In *Advanced Information Networking and Applications*, pages 445–450. IEEE, 2004.

[305] Irene Sygkouna, Maria Strimpakou, Francisco Valera, Anastasia Kaltabani, Luis Bellido, Enrique Vazquez, and Miltiades Anagnostou. Seamless incorporation of agents in an e-commerce intermediation platform. *Springer-Verlag Berlin Heidelberg*, pages 292–301, 2002. LNCS2521.

[306] SymMobile. `http://www.symmobile.com/img/news/bridge_p800.gif`.

[307] Toshiyuki Takeda. A design for computer supported collaborative learning using concerns oriented model. In *Conference on Creating, Connecting and Collaborating Through Computing*, pages 89–95. IEEE, Jan 2003.

[308] M. Tambe, W.M. Shen, M. Mataric, D. Goldberg, P.J. Modi, Z. Qiu, and B. Salemi. Teamwork in cyberspace: Using TEAMCORE to make agents teamready. `http://citeseer.ist.psu.edu/8634.html`, 1998.

[309] The Tams11. Tams11 gaming lobby. `http://www.tams11.com/pokersquares/pokersquares.jpg`.

[310] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Upper Saddle River, N.J., 2nd edition, 2001.

[311] Paul Tarau. Towards inference and computation mobility: The Jinni experiment. In J. Dix, L. Farinas del Cerro, and U. Furbach, editors, *JELIA'98*, pages 385–390. Springer-Verlag Berlin Heidelberg, 1998. LNAI 1489.

[312] Simon J.E. Taylor, Jon Saville, and Rajeev Sudra. Developing interest management techniques in distributed interactive simulation using java. In *Proceedings of the 1999 Winter Simulation Conference*, volume 1, pages 518–523. IEEE, 1999.

[313] Thuan Thai and Hoang Q. Lam. *.NET Framework Essentials.* O'Reilly & Associates, 1005 Gravenstein Highway North Sebastopol CA 95472, second edition, February 2002.

[314] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[315] Roshan K. Thomas. Team-based access control (tmac): A primitive for applying role-based access controls in collaborative environments. In *Workshop on Role-based Access Control*, pages 13–19. ACM, November 1997.

[316] Robert Tolksdorf and Dirk Glaubitz. XMLSpaces for coordination in web-based systems. In *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE)*, pages 322–327. IEEE, June 2001.

[317] W. Tolone, G.J. Ahn, T. Pai, and S.P. Hong. Access control in collaborative systems. *ACM Computing Survey*, 37(1):29–41, March 2005.

[318] D.M. Traill, J.M. Bowskill, and P.J. Lawrence. Interactive collaborative media environments. *BT Technology Journal*, 14(4):130–140, October 1997.

[319] Yuh-Min Tseng. A scalable key-management scheme with minimizing key storage for secure group communications. *International Journal of Network Management*, 13(6):419–425, Nov-Dec 2003.

[320] UIUC. Habanero. `http://www.isrl.uiuc.edu/isaac/Habanero`.

[321] Bill Venners. The philosophy of Ruby. `http://www.artima.com/intv/rubyP.html`, September 2003.

[322] W3C. W3c world wide web consortium. `http://www.w3.org`, 1994–2006.

[323] Lihua Wang, S.J. Turner, and Fang Wang. Interest management in agent-based distributed simulations. In *International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'03)*, pages 20–27. IEEE, 2003.

[324] S.Q. Wang, L. Chen, and G.C Chen. A framework for Java 3D based collaborative virtual environment. In *International Conference on Computer Supported Cooperative Work in Design*, volume 1, pages 34–39. IEEE, May 2004.

[325] Greg Ward. Distributing Python modules. `http://www.python.org/doc/current`.

[326] Richard Waters and David Anderson. Scalable platform for large interactive networked environments. `http://www.merl.com/projects/spline`, October 2002.

[327] David J. Weiss. *BRIDGE Parity Leads in Defence.* Robert Hale.London, 1994.

[328] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *Proceedings on High Performance Distributed Computing (HPDC'03)*, pages 48–57. IEEE, June 2003.

[329] Whatis.com. Web services definitions. `http://whatis.techtarget.com`, April 2006.

[330] Larry D. Wittie. Computer networks and distributed systems. *Computer*, 24(9):67–76, Sept 1991.

[331] Bing Xie, Xiaolin Gui, Yinan Li, and Depei Qian. A new grid security framework with dynamic access control. In H. Jin, Y. Pan, N. Xiao, and J. Sun, editors, *Grid and Cooperative Computing*, volume 3251, pages 863–866. Springer-Verlag Berlin Heidelberg, Oct 2004.

[332] J. Yang and D. Lee. Scalable prediction based concurrency control for distributed virtual environment. In *Virtual Reality*, pages 151–158. IEEE, March 2000.

[333] Kevin Yank. Interview- PHP's creator, Rasmus Lerdorf. `http://www.sitepoint.com/print/phps-creator-rasmus-lerdorf`, May 2002.

[334] Stephen S. Yau and Fariaz Karim. An adaptive middleware for context-sensitive communications for real-time appliations in ubiquitous computing environments. *Real-Time Systems*, 26(1):29–61, Jan 2004.

[335] E. Yoshida and H. Kakugawa. A learning system for the problem of mutual exclusion in multithreaded programming. In *Conference on Advanced Learning Technologies*, pages 2–6. IEEE, Aug–Sept 2004.

[336] A.P. Yu and S.T. Vuong. MOPAR: A mobile peertopeer overlay architecture for interest management of massively multiplayer online games. In *NOSSDAV'05*, pages 99–104. ACM, June 2005.

[337] F. Zaffar, G. Kedem, and A. Gehani. Paranoid: A global secure file access control system. In *ACSAC 2005*. IEEE, 2005.

[338] M. Zellouf, P. Prevot, and R. Aubry. Computer–supported coordination and communication in collaborative development of courseware. In *Conference on Communications, Power, and Computing*, pages 95–100. IEEE, May 1995.

[339] S. Zhang, A. Burns, J. Chen, and E. Stewart Lee. Hard real-time communication with the timed token protocol: Current state and challenging problems. *Real-Time Systems*, 27(3):271–295, September 2004.

[340] X. Zhang, Y. Li, and D. Nalla. An attribute–based access matrix model. In *Symposium on Applied Computing*, pages 359–363. ACM, March 2005.

[341] L. Zou, M.H. Ammar, and C. Diot. An evaluation of grouping techniques for state dissemination in networked multi-user games. In *Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 33–40. IEEE, 2001.

List of Figures

1.1 Internet Collaborative Activities. 4
1.2 JACIE Sample Output 5

2.1 Client/Server Connections that Allow Users Collaboration. 16
2.2 Pogo Game Collaboration Chess. 29
2.3 Entities, Cells and Vision Domains [341]. 30
2.4 Consistency Mechanisms in MASSIVE-3 [135]. 37

3.1 Possible Configuration of RPCs [293]. 46
3.2 Session Manager in DistView [241]. 47
3.3 Shared Memory as an API [300]. 59

4.1 JACIE Software Architecture. 64
4.2 JACIE Compiler. 66
4.3 Standard JACIE Component. 67
4.4 Layout Diagram of JACIE User Interface. 69
4.5 State Diagram for Server and Client. 70
4.6 A JACIE Message and Its Representation in Relation to Other Network Layers. 70
4.7 An Example of a JACIE Message with Values. 71
4.8 Message Exchange. 73
4.9 Outline Structure of JACIE Compiler. 83
4.10 Error on the Lexical Phase. 84
4.11 Error on Determining Grammar. 85
4.12 Syntax Error in a Program. 86
4.13 Error Message on Syntax. 86
4.14 A Local Variable is Treated as Global. 87
4.15 Error Message on Semantic Checking. 88

5.1 Various Versions of the Noughts and Crosses Type Games. 94
5.2 Win Conditions. 97
5.3 Screenshots of Two Noughts and Crosses Games. 98
5.4 Screenshots of Generalised Games. 102
5.5 JACIE Collaborative Management. 112

6.1 Tables for Managing Shared Variable. 129
6.2 Flow Chart of the Server Control on Permission Setting. 131

6.3 Flow Chart of Write Operation. 135
6.4 Flow Chart of Read Operation. 137
6.5 Flow Chart of Filtering Process. 138
6.6 Summary of Message Passing Activities. 150
6.7 Screenshot of the Noughts and Crosses Game with Password. 153

7.1 Example of Bridge Games. 156
7.2 Bridge Game (The Bidding Process). 158
7.3 Bridge Game (The Trick Play). 162
7.4 Overall Network Settings. 167
7.5 Layout of the Individual Room. 168
7.6 Room Selection. 169
7.7 Read Operation on Devices in Room 1. 173
7.8 Ready To Do a Write Operation on a Device in Room 3. 174
7.9 Checking on Network Components. 175
7.10 Detecting the Problem and Solve. 176
7.11 Output of JACIE Hello Program. 180
7.12 Output of Program with Contention Protocol. 181
7.13 Output of Java Hello Program 184

List of Tables

2.1 CSCW Environment. . . . 13

3.1 Comparison on Scripting Languages. . . . 53
3.2 General Form of Access Control Matrix Model. . . . 61

5.1 Variations of the Noughts and Crosses Game, and Their Main Features. . . 99
5.2 New Java Classes for Interaction Management. . . . 115
5.3 Changes on FloorManagerTemplate.java File. . . . 115
5.4 Changes on JACIECGroupManagerTemplate.java File. . . . 116
5.5 Timer Based Comparison. . . . 121

6.1 Environment and Applications for Distributed Systems. . . . 126
6.2 Shared Variable Attributes on the Server. . . . 130
6.3 Shared Variable Attributes on the Client. . . . 132
6.4 Example Applications Involving Read and Write Access Conditions. . . . 134
6.5 New Java Classes for Interest Management. . . . 145
6.6 Comparison on the Secret Switch Implementations. . . . 152

7.1 Codes Length Comparison on JACIE and Its Java Translated Programs. . . 178
7.2 Processing Comparison on JACIE and Its Java Translated Programs. . . . 179
7.3 Participants Background Data. . . . 181
7.4 Learning and Testing Sessions. . . . 182
7.5 Transmission Delay Between Server and Client 185
7.6 Interaction Delay on a User 186

B.1 Technical Terms. . . . 201
B.2 Mathematical Symbols. . . . 201

C.1 Primitive Data Type. . . . 202
C.2 Arithmetic Operators. . . . 202
C.3 Relational and Conditional Operators. . . . 203