

*Towards a Model for Teaching Distributed
Computing in a Distance-based Educational
Environment*

by

Petra le Roux

Submitted in fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROF A J VAN DER MERWE

FEBRUARY 2009

Abstract

Several technologies and languages exist for the development and implementation of distributed systems. Furthermore, several models for teaching computer programming and teaching programming in a *distance-based educational environment* exist. Limited literature, however, is available on models for teaching *distributed computing* in a distance-based educational environment. The focus of this study is to examine how distributed computing should be taught in a distance-based educational environment so as to ensure effective and quality learning for students. The required effectiveness and quality should be comparable to those for students exposed to laboratories, as commonly found in residential universities. This leads to an investigation of the factors that contribute to the success of teaching distributed computing and how these factors can be integrated into a distance-based teaching model.

The study consisted of a literature study, followed by a comparative study of available tools to aid in the learning and teaching of distributed computing in a distance-based educational environment. A model to accomplish this teaching and learning is then proposed and implemented. The findings of the study highlight the requirements and challenges that a student of distributed computing in a distance-based educational environment faces and emphasises how the proposed model can address these challenges.

This study employed *qualitative research*, as opposed to quantitative research, as qualitative research methods are designed to help researchers to understand people and the social and cultural contexts within which they live. The research methods employed are *design research*, since an artefact is created, and a *case study*, since “how” and “why” questions need to be answered. Data collection was done through a *survey*. Each method was evaluated via its own well-established evaluation methods, since evaluation is a crucial component of the research process.

Keywords

Distributed Computing, Distributed Systems, Distance-based Educational Environment, Qualitative Research, Case Study, Survey.

Acknowledgements

I wish to express my thanks to the following people, whose assistance and cooperation made this study possible:

My Heavenly Father: for His mercy and grace, this enabled me complete this study.

My supervisor, Prof Alta van der Merwe: for her positive attitude and enthusiastic supervision.

My husband, Evert: for his unwavering support, understanding, patience and love.

My children, Evert (Jnr), Karli and Mieke: for allowing me to use the computer all the time.

My mother: for always believing in me.

Lizzie: for all the coffee, which made it possible.

My sister, Karin in London: for having a thesaurus at hand via Skype.

Table of Contents

1.	INTRODUCTION	12
1.1	Chapter Overview	12
1.2	Introduction	13
1.3	Background	14
1.4	Problem Statement	15
1.5	Research Goal.....	16
1.6	Research Question	17
1.7	Solution Strategy	17
1.8	Scope.....	18
1.9	Relevance and Significance of this Research.....	19
1.10	Synopsis	19
2.	DISTRIBUTED COMPUTING THEORY.....	22
2.1	Chapter Overview	22
2.2	Introduction	23
2.3	History of Distributed Systems.....	24
2.3.1	Monolithic Applications	24
2.3.2	Client/Server	24
2.3.3	Three-tier Client/Server	25
2.3.4	Multiple-tier Client/Server.....	25
2.3.5	Distributed Components	25
2.3.6	Conclusion	26
2.4	Distributed Computing.....	26
2.4.1	Why a Distributed System?	26
2.4.1.1	Scalability.....	27
2.4.1.2	Extensibility and Maintenance	27
2.4.1.3	Heterogeneity	27
2.4.1.4	Resource Sharing	28
2.4.1.5	Fault Tolerance	28
2.4.1.6	Conclusion.....	28
2.4.2	Distributed Computing Technologies.....	28
2.4.2.1	Socket Programming.....	28
2.4.2.2	Remote Procedure Call (RPC).....	29
2.4.2.3	Distributed Computing Environment (DCE).....	29
2.4.2.4	Object Request Brokers (ORBs).....	30
2.4.2.4.1	CORBA.....	30

2.4.2.4.2	DCOM	30
2.4.2.4.3	RMI.....	30
2.4.2.5	MOM.....	31
2.4.2.6	Conclusion.....	31
2.4.3	Elements of a Distributed System	31
2.4.4	Middleware	33
2.4.4.1	Middleware Defined	33
2.4.4.2	Middleware Objective.....	35
2.4.5	Transparency in Distributed Systems	36
2.5	Design of Distributed Systems	38
2.5.1	Introduction.....	38
2.5.2	Elements in the Design of Distributed Systems	39
2.5.2.1	Interfaces	39
2.5.2.2	Granularity.....	39
2.5.2.3	System Partitioning	40
2.5.3	Design of Individual Distributed Objects	40
2.5.3.1	Object-oriented Middleware.....	40
2.5.3.2	Developing with Object-oriented Middleware	41
2.5.3.2.1	Design	41
2.5.3.2.2	Interface Definition Language.....	42
2.5.3.2.3	Stub Generation.....	42
2.5.3.2.4	Implementation of Client Objects.....	43
2.5.3.2.5	Implementation of Server Objects	44
2.5.3.2.6	Server Registration	45
2.5.3.2.7	Use of Server Object.....	45
2.6	Summary	46
3.	TEACHING MODELS.....	47
3.1	Chapter Overview.....	47
3.2	Introduction	48
3.3	What do we know about Teaching Programming?.....	48
3.3.1	Programming Defined	48
3.3.2	Why do we teach Programming?	49
3.3.3	Why is it Difficult?.....	49
3.3.4	Examples in Use.....	50
3.3.4.1	Microworlds	50
3.3.4.2	Compilers with Improved Diagnostic Capabilities	51
3.3.4.3	Iconic Programming Languages	52
3.3.4.4	Program Animation.....	53
3.3.5	Conclusion	54
3.4	What do we know about Teaching in a Distance-based Educational Environment?...55	
3.4.1	Distance-based Educational Environment Defined.....	55

3.4.2	Problems Identified	56
3.4.3	Examples in Use.....	57
3.4.4	Conclusion	66
3.5	What do we know about Teaching Distributed Computing?.....	68
3.5.1	Distributed Computing Defined.....	68
3.5.2	Problems Identified	68
3.5.3	Examples in Use.....	69
3.5.4	Conclusion	76
3.6	What do we know about Teaching Distributed Systems in a Distance-based Educational Environment?.....	77
3.6.1	Background: COS3114.....	78
3.6.2	Changes in the Computer Science Discipline	78
3.6.2.1	Technical Changes	78
3.6.2.2	Cultural Changes.....	79
3.6.2.3	COS3114.....	79
3.6.3	Conclusion	79
3.7	Summary	80
4.	RESEARCH METHODOLOGY AND DESIGN	81
4.1	Chapter Overview	81
4.2	Introduction	82
4.3	Research.....	82
4.4	Research Approach.....	83
4.4.1	Quantitative Research.....	83
4.4.2	Qualitative Research.....	83
4.4.3	Qualitative Research in Computing Education Research	84
4.5	Research Method	84
4.5.1	Introduction.....	84
4.5.2	Design-research Approach.....	84
4.5.2.1	Design	84
4.5.2.2	Design as Research	85
4.5.2.3	Design Research	87
4.5.3	Case-study Research Approach.....	90
4.5.3.1	Background	90
4.5.3.2	Designing a Case Study.....	90
4.5.4	Survey Research.....	93
4.5.4.1	Background	93
4.5.4.2	Designing a Survey	93

4.6	Evaluation Methods	95
4.6.1	Introduction.....	95
4.6.2	Design-research Evaluation Methods.....	95
4.6.3	Case-study Research Evaluation Methods.....	96
4.6.4	Survey-research Evaluation Methods.....	97
4.7	An Appropriate Research Approach for this Study	98
4.7.1	Research Question.....	98
4.7.2	Qualitative Research.....	98
4.7.3	Design Research.....	99
4.7.4	Case-study Research.....	100
4.7.5	Survey Research.....	100
4.8	Summary	101
5.	THE MODEL	102
5.1	Chapter Overview.....	102
5.2	Introduction	103
5.3	Independent Distributed Learning Model	103
5.4	Resource Space.....	104
5.5	Learning Space	105
5.5.1	Student Resource and Workspace	106
5.5.2	Teacher Resource and Workspace	106
5.6	Model Specification.....	107
5.7	Use-case Diagrams: Resource Space	108
5.7.1	Introduction.....	108
5.7.2	Description of Users.....	108
5.7.2.1	Teacher.....	108
5.7.2.2	Student	109
5.7.3	Description of Use Cases.....	109
5.7.3.1	Use Case: Tutoring Resources.....	109
5.7.3.2	Use Case: Assignment Management.....	110
5.7.3.3	Use Case: Communication	111
5.7.3.3.1	Module Webpage.....	111
5.7.3.3.2	Email.....	111
5.7.3.3.3	Discussion Forum.....	111
5.7.3.3.4	Newsgroups.....	111
5.8	Use-case Diagrams: Teacher Resource and Workspace	112
5.8.1	Introduction.....	112
5.8.2	Description of Users.....	112
5.8.3	Description of Use Cases.....	113

5.8.3.1	Use Case: Expertise Space.....	113
5.8.3.2	Use case: Semantic Rules.....	113
5.8.3.3	Use Case: Tutoring Text	114
5.8.3.4	Use Case: Assignments	114
5.9	Use-case Diagrams: Student Resource and Workspace	115
5.9.1	Introduction.....	115
5.9.2	Description of Users.....	116
5.9.3	Description of Use Cases.....	116
5.9.3.1	Use Case: User Interface.....	116
5.9.3.2	Use Case: Tutoring Material.....	117
5.9.3.3	Use Case: Student File Management Systems.....	118
5.9.3.4	Use Case: Assignments	118
5.10	Summary	119
6.	CASE STUDY: UNISA.....	120
6.1	Chapter Overview	120
6.2	Introduction	121
6.3	Application of the Independent Distributed Learning Model.....	121
6.3.1	Case Study: Determine and Define the Research Question	121
6.3.2	Case Study: Participant Selection	121
6.3.3	Case Study: Select the Case(s).....	122
6.3.3.1	Teacher Resource and Workspace: Create Expertise Space.....	123
6.3.3.2	Teacher Resource and Workspace: Test Semantic Rules.....	125
6.3.3.3	Teacher Resource and Workspace: Tutoring Text.....	126
6.3.3.4	Teacher Resource and Workspace: Assignments	129
6.3.3.5	Student Resource and Workspace: User Interface.....	130
6.3.3.5.1	Exercise 01	132
6.3.3.6	Student Resource and Workspace: Tutoring Environment	132
6.3.3.7	Student Resource and Workspace: File Management System.....	133
6.3.3.8	Student Resource and Workspace: Assignments	134
6.3.3.8.1	Exercise 02.....	136
6.3.3.8.2	Exercise 03.....	137
6.3.3.8.3	Exercise 04.....	137
6.3.3.8.4	Exercise 05.....	138
6.3.3.8.5	Exercise 06.....	139
6.3.3.8.6	Project	139
6.3.4	Case Study: Prepare to Collect Data	141
6.3.4.1	Introduction	141
6.3.4.2	Survey: Constructing a Survey	141
6.3.4.3	Survey: Conducting a Survey	142
6.3.5	Case Study: Evaluate and Analyse Data.....	143

6.3.5.1	Survey: Analysing Survey Results.....	143
6.3.6	Case Study: Composing the Report.....	146
6.3.6.1	Survey: Reporting Survey Results	146
6.4	Summary	147
7.	DISCUSSION OF FINDINGS	148
7.1	Chapter Overview	148
7.2	Introduction	149
7.3	Reflection and Contribution	150
7.3.1	Reflection and Contribution: SQ 01	150
7.3.2	Reflection and Contribution: SQ 02.....	151
7.3.3	Reflection and Contribution: SQ 03.....	151
7.3.4	Reflection and Contribution: SQ 04.....	152
7.3.5	Conclusion	153
7.4	Scientific Contribution.....	153
7.5	Practical Contribution	154
7.6	Further Research.....	154
7.7	Conclusions	154
8.	WORKS CITED.....	156
	APPENDIX A – SOFTWARE ENVIRONMENT SELECTION.....	161
	APPENDIX B – ADDITIONAL RESOURCES	164
	APPENDIX C - SOFTWARE INSTALLATION.....	165
	APPENDIX D – SOFTWARE MECHANICS	169
	APPENDIX E - RUNNING A CORBA CLIENT AND SERVER	179
	APPENDIX F - CORBA WITH MICO.....	183
	APPENDIX G - MORE CORBA WITH MICO.....	192
	APPENDIX H - THE CORBA NAMESERVICE.....	200
	APPENDIX I - A SIMPLE APPLICATION	205

List of Figures

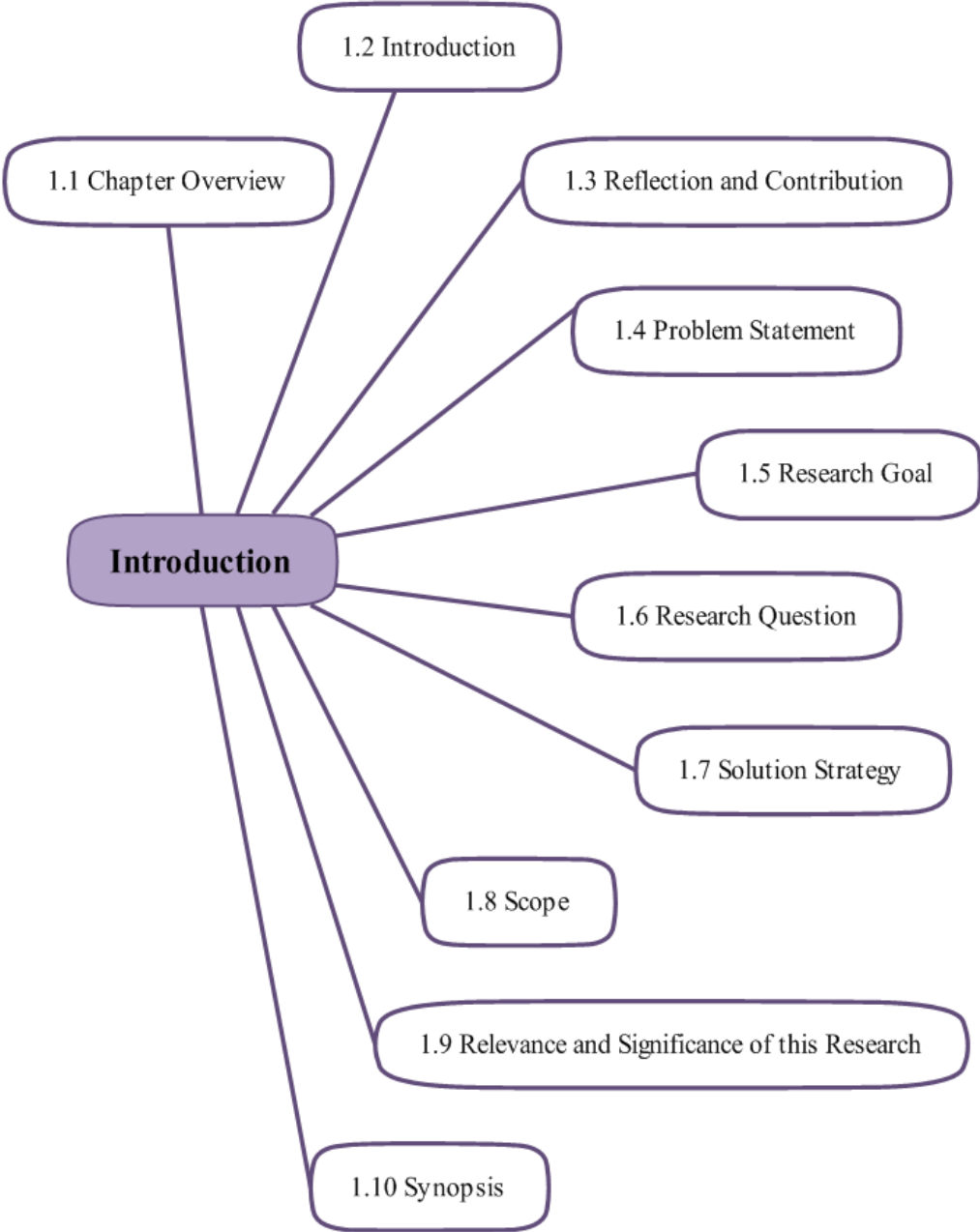
Figure 2-1. Evolution of Distributed Systems (Balen, 2000)	24
Figure 2-2. Evolution of Distributed Systems	26
Figure 2-3. Hierarchy of distributed computing approaches	29
Figure 2-4. Hosts in a Distributed System (Emmerich, 2000)	31
Figure 2-5. Middleware in a Distributed System (Emmerich, 2000).....	32
Figure 2-6. Working Model of a Distributed System	33
Figure 2-7. Layering of Middleware and Related Software (Benns, 1999)	35
Figure 2-8. Dimensions of Transparency in Distributed Systems (Emmerich, 2000)	37
Figure 2-9. Design and Implementation Process for Distributed Systems	41
Figure 2-10. Object Requests (Emmerich, 2000)	43
Figure 3-1. Generic Model for Teaching Programming Languages	55
Figure 3-2. Teaching Programming in a Distance-based Educational Environment	67
Figure 3-3. Teaching Distributed Computing.....	77
Figure 4-1. The Design Activity according to Simon (Vaishnavi, 2004/5).....	85
Figure 4-2. The Design Activity according to Simon (expanded).....	85
Figure 4-3. A General Model for Generating and Accumulating Knowledge (Owen, 1997)	86
Figure 4-4. The Design Cycle	86
Figure 4-5. New Circumscription-knowledge Production	87
Figure 4-6. New Operational and Goal-knowledge Production	87
Figure 4-7. Basic Types of Design in Case Studies	92
Figure 5-1. Distributed Learning Model	103
Figure 5-2. Resource Space.....	105
Figure 5-3. Learning Space	105
Figure 5-4. Student Resource and Workspace.....	106
Figure 5-5. Teacher Resource and Workspace	106
Figure 5-6. Use-case Diagram - Resource Space	108
Figure 5-7. Use-case Diagram - Resource Space: Tutoring Resources	110
Figure 5-8. Use-case Diagram - Resource Space: Assignment Management	110
Figure 5-9. Use-case Diagram - Resource Space: Communication.....	111
Figure 5-10. Use-case Diagram - Teacher Resource and Workspace.....	112
Figure 5-11. Use-case Diagram - Teacher Resource and Workspace: Expertise Space	113
Figure 5-12. Use-case Diagram - Teacher Resource and Workspace: Semantic Rules	113
Figure 5-13. Use-case Diagram - Teacher Resource and Workspace: Tutoring Text	114
Figure 5-14. Use-case Diagram - Teacher Resource and Workspace: Assignments	115
Figure 5-15. Use-case Diagram - Student Resource and Workspace	115
Figure 5-16. Use-case Diagram - Student Resource and Workspace: User Interface.....	117
Figure 5-17. Use-case Diagram - Student Resource and Workspace: Tutoring Material	117
Figure 5-18. Use-case Diagram - Student Resource and Workspace: File-management System	118
Figure 5-19. Use-case Diagram - Student Resource and Workspace: Assignments.....	118
Figure 6-1. Teacher and Student Resource and Workspace Case Diagrams	122
Figure 6-2. Activity Diagram - Teacher Resource and Workspace: Create Expertise Space.....	125
Figure 6-3. Activity Diagram - Teacher Resource and Workspace: Test Semantic Rules.....	126
Figure 6-4. Activity Diagram - Teacher Resource and Workspace: Tutoring Text.....	128
Figure 6-5. Activity Diagram - Teacher Resource and Workspace: Assignments	130
Figure 6-6. Activity Diagram - Student Resource and Workspace: User Interface.....	131
Figure 6-7. Activity Diagram - Student Resource and Workspace: Tutoring Environment	133
Figure 6-8. Activity Diagram - Student Resource and Workspace: File-management System.....	134
Figure 6-9. Activity Diagram - Student Resource and Workspace: Assignments	135
Figure F-1. C++ files used to compile the server	185
Figure G-1. Architecture of a simple application	192
Figure G-2. Mapping of object reference to servant	198

List of Tables

Table 1-1. Outline of Chapters	20
Table 3-1. Teaching Programming: Microworlds	50
Table 3-2. Teaching Programming: Compilers with Improved Diagnostic Capabilities	52
Table 3-3. Teaching Programming: Iconic Programming Languages	53
Table 3-4. Teaching Programming: Program Animation.....	54
Table 3-5. Teaching in a Distance-based Educational Environment: Summary of Examples	66
Table 3-6. Teaching Distributed Computing: Summary of Systems Evaluated	76
Table 4-1. Seven Guidelines for Effective Design Research	88
Table 4-2. Four Design Outputs (Artefacts) to be produced by Design Research.....	89
Table 4-3. Vaishnavi's Addition to the Outputs of Design Research	89
Table 4-4. Components of the Case-study Design Process.....	91
Table 4-5. Components of the Survey-design Process.....	93
Table 4-6. Design-research Evaluation Methods.....	95
Table 4-7. Case-study Evaluation Methods.....	96
Table 6-1. All Possible Cases that are identified	123
Table 6-2. Tutorial Letter Notation.....	127
Table 6-3. Activities of the Student Resource and Workspace: User Interface.....	132
Table 6-4. Exercises in Module COS3114.....	136
Table 6-5. Student Numbers in Survey	142
Table 6-6. Survey Question #1.....	143
Table 6-7. Graphical Representation of the Response to Survey Question #1	144
Table 6-8. Survey Question #2.....	144
Table 6-9. Graphical Representation of the Response to Survey Question #2.....	145
Table C-1. Supported Environments.....	166

1. Introduction

1.1 Chapter Overview



The focus of this study is on the presentation of distributed computing within the distance-based educational domain. In this chapter, some background to this study is provided. The goal is to give an overview of the problem and to set out the research questions that guide this study.

1.2 Introduction

The design and implementation of software is a difficult and an expensive process (Condi, 1999). This process is also complex, even in a homogeneous environment, that is, an environment consisting of a single, stable platform, using a single operating system and a single programming language, usually from a single vendor. The complexity increases immensely if this process takes place in a heterogeneous environment, in which the vendors of the hardware and system software may differ. A heterogeneous environment adds a number of additional complexities to new software development (Baker, 1997; Balen, 2000; Condi, 1999; Rock-Evans, 1999), in that the new software must run on a *network* of hosts, and these *hosts* may run different *operating systems*. Furthermore, components of the new software will in all probability have to be integrated into *legacy* systems. Adding to the complexity is the probable use of different *programming languages* for different components of the new software system.

The complexities referred to above can cause an increase in the cost of developing and maintaining software. To address some of these complexities, Condi (1999) proposes that new software needs to be designed and developed as a set of integrating components that can *communicate* across the boundaries of a network, different operating systems and different programming languages. Furthermore, legacy code has to be *wrapped* so as to resemble components to ensure that they can be integrated into new systems. These components also have to be *transferable to new developments* to ensure that they can be integrated into various applications.

Distributed programming and distributed computing frameworks, such as CORBA or JAVA RMI, assist in and simplify the design and development of systems that can communicate across the boundaries of a network, which run on different operating systems and which were written in different programming languages. The distributed computing frameworks commonly referred to as *middleware* offer flexibility and new ways of integrating existing and new technology, as well as new ways of facilitating communication between systems.

Distributed computing frameworks have raised expectations of highly functional systems that can solve the majority of computing problems. Experience, however, has shown that it is challenging to build such distributed applications (Benns, 1999). Neither the CORBA standard nor conventional

implementations of CORBA directly address some complex problems that are related to distributed computing. Examples thereof are real-time quality of service (QoS) or high-speed performance, group communication, partial failures and casual ordering of events. Apart from the difficulties and challenges involved, distributed computing is increasingly being used as a basis for the World Wide Web and distributed network-related software developments.

To simplify the description of the complexities involved in distributed computing, two issues in this environment are distinguished: (1) *distributed technologies*, themselves forming a basis for development and interaction between systems, and (2) the *interaction* between the developer/user and distributed technologies. The focus of this study is on human interaction with these technologies. As a basis for this study, the distributed computing frameworks designed and developed by standards bodies, such as CORBA technologies and Java RMI, are acknowledged and accepted.

As mentioned earlier, the use of distributed computing frameworks is not simple. Furthermore the use (plug and play) of distributed computing components presents various challenges. Besides the embedded complexities of this subject, the teaching thereof poses various problems to both student and teacher. In a residential institution, students can be exposed to the various elements of a distributed computing environment under laboratory conditions. This might simplify some of the complexity associated with the learning of distributed computing. In a distance-based learning environment, however, a different approach is needed to ensure that students receive the same quality of teaching and are able to experience the same degree of learning as under laboratory conditions. Teaching distributed computing without laboratory sessions is much more challenging than in a laboratory scenario and might fail completely if a set of clear guidelines is not available to direct the teaching and learning process.

1.3 Background

The final report of the Computing Curricula 2001(CC2001) project¹ (2001) stated that computer science was an enormously vibrant field. Computers are integral to modern culture and are the primary engine behind much of the world's economic growth. The field is evolving at an astonishing pace as new technologies are introduced and existing ones become obsolete.

¹ CC2001 is a joint undertaking of the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) and the Association for Computing Machinery (ACM) to develop curricular guidelines for undergraduate programmes in computing.

The rapid evolution of the discipline has a profound effect on computer science education, affecting both content and pedagogy. The CC2001 Task Force identified aspects of computer science that had changed over the past decade and categorised them as either technical or cultural. Each of these categories has a significant effect on computer science education. The major changes in each of these categories are described below.

The CC2001 identified that the technical changes in computer science are both evolutionary (exponential increase in available computing power) and revolutionary (the rapid growth of networking after the appearance of the World Wide Web). These changes are affecting the body of knowledge required for computer science and the educational process. Thus, technical advances over the past decade have increased the importance of many curricular topics, such as the World Wide Web and its applications, networking technologies - particularly those based on TCP/IP, embedded systems, interoperability, object-oriented programming, use of sophisticated application programmer interfaces (APIs) and security and cryptography.

Computing education is also affected by changes in the cultural and sociological context in which it occurs, according to CC2110. The following changes, for example, have all had an influence on the nature of the educational process: changes in pedagogy enabled by new technologies, the dramatic growth of computing throughout the world, the growing economic influence of computing technology, greater acceptance of computer science as an academic discipline and broadening of the discipline.

According to CC2001, advanced courses serve three purposes: (1) exposing the student to advanced material beyond the core, (2) demonstrating applications of fundamental concepts presented in the core courses and (3) providing students with a depth of knowledge in at least one subarea of computer science.

Many universities offer distributed systems as a course as part of third-year level undergraduate studies. In this study, offering a complex course of this nature is examined.

1.4 Problem Statement

Several technologies and languages exist for the development and implementation of distributed systems, for example, CORBA, Java RMI and Microsoft's .NET. These technologies address the complexities of designing and implementing distributed systems; systems in which all components are objects that can communicate with each other across boundaries such as networks, different operating systems and different programming languages.

Several models for teaching computer programming, as well as teaching programming in a *distance-based educational environment*, exist (Benns, 1999; Budd, 1997; Condi, 1999; Welsh, 1999). The models were developed to aid in the teaching and learning of programming, and they aim to ease learning of the programming language by eliminating complexities that do not directly contribute to achievement of the learning objective. The examples researched for effective teaching in a distance-based educational environment are built on the client-server model. In its basic form, the client side consists of a lecturing agent (or lecturing agents) and student agents, and the server provides the content and infrastructure needed to present a course. Furthermore, several models for the teaching and learning of distributed computing exist, and these systems came into being to address challenges that had been identified.

Limited literature, however, is available on models for teaching *distributed computing* in a distance-based educational environment.

For the purpose of this study, the question of concern is how distributed computing should be taught in a distance-based educational environment to ensure effective and quality learning for students. The required effectiveness and quality should be comparable to those for students exposed to laboratories, as commonly found at residential universities. This give rise to the following question: What are the factors contributing to the success of teaching distributed computing, and how can these factors be integrated into the distance-based teaching model?

1.5 Research Goal

The goal of this research is:

- To identify the elements of distributed computing in order to compile a body of knowledge needed to be present in a model to teach and learn distributed computing effectively.
- To investigate available models for the teaching of computer programming, programming in a distance-based educational environment and distributed computing in order to identify success factors for teaching distributed computing in a distance-based educational environment.
- To establish and verify a model for the effective teaching and learning of distributed computing in a distance-based educational environment.

1.6 Research Question

Numerous theoretical and technological issues need to be addressed during the teaching and learning of distributed computing. The purpose of this study is to identify and investigate the issues that could contribute towards a model for the teaching of distributed computing in a distance-based institution of higher education.

The primary research question addressed by this research study is:

How should distributed computing be taught in a distance-based educational environment?

The subsidiary questions that relate to the research objectives that guide this study include:

- **SQ1:** *What is a distributed system, and what are the aspects to be considered in designing a distributed system?*
- **SQ2:** *What are the factors contributing to the success of teaching programming, and how can these factors be integrated into teaching distributed computing in a distance-based teaching model?*
- **SQ3:** *Which guidelines can the teacher involved in teaching distributed computing in a distance-based educational environment use to direct the teaching and learning process?*
- **SQ4:** *How can the proposed model for effective teaching and learning of distributed computing be implemented at a distance-based institution of higher education?*

1.7 Solution Strategy

The purpose of this study is to provide a teaching model to direct the effective teaching and learning process of distributed computing in a distance-based educational environment.

A theoretical survey and a literature review were undertaken to identify the elements involved in teaching distributed computing in a distance-based educational environment. Firstly, as the content to be presented is *distributed computing* and is not generic by nature, the purpose of this literature study includes the investigation and identification of the elements involved in distributed computing. This

includes, *inter alia*, the study of distributed computing technologies, identification of a design strategy for distributed systems and identification of a model to define distributed objects as an object-oriented view of distributed systems. This investigation is necessary, since technology and design are important aspects of a system, provide a framework for construction, embody principles for developers to follow and support future enhancements. Furthermore, a model to define distributed objects is important, since different objects on different servers may participate in an application at different times, and different applications and/or users may actively share objects (Emmerich, 2000).

Secondly, an investigation of available models for the teaching of computer programming, the teaching of programming in a distance-based educational environment and the teaching of distributed computing was conducted. As a result of this background literature study, the features, characteristics and elements necessary for the teaching and learning of distributed computing in a distance-based educational environment could be extracted. This leads to the argument that the available models do not adequately address the teaching of distributed computing in a distance-based educational environment.

After identification of the shortcomings that impede the quality and effectiveness of the teaching process, a teaching model is proposed. The proposed model comprises both known elements and newly identified elements needed for effective learning and teaching of distributed computing in a distance-based educational environment.

Specifically aimed at teaching distributed computing in a distance-based educational environment, the model was subsequently implemented. The model uses standardised technologies (such as the Internet and CORBA) available for the development and implementation of distributed systems. The model was verified and evaluated by students. Finally, the results were interpreted and written up.

1.8 Scope

Because of the vast scope of distributed computing, the theoretical study is limited to the notion of a distributed system, the key elements and their properties and also the underlying models forming a basis for the development and deployment of distributed systems. Although the existence and value of middleware platforms, such as Java RMI and Microsoft's .NET (previously DOM), are recognised, the model of this research is based on the CORBA specification. This is due to the openness of the aforementioned specification and the existence of open source implementations, which make it an ideal vehicle for the study of middleware protocols and distributed component infrastructure issues (Baker, 1997; Bennis, 1999; Condi, 1999; Mowbray, 1997; Orfali, 1998).

The focus of this study is to propose a model to direct the effective teaching and learning process of distributed computing in a distance-based educational environment by making use of the *Internet* as the vehicle to present the practical sessions. The proposed model is not a generic e-learning model, and it is not the intention of this study to propose such a model. There is also no focus on specific e-learning strategies or learning theories. The intention of this study is not to become involved in pedagogical theories of learning. The underlying importance of these theories is recognised, and they are used where applicable.

1.9 Relevance and Significance of this Research

The presentation of a distinctive approach when teaching distributed computing in a distance-based educational environment is relevant and significant, since it identifies factors that facilitate a teaching and learning environment in which it is possible to achieve quality of learning. The significance of the model is that it also addresses a gap in the literature. Although several models exist for teaching computer programming, as well as teaching in a distance-based educational environment, explicit literature for teaching distributed computing in a distance-based educational environment, which requires not only a practical programming component, but also a heterogeneous implementation environment, could not be found. This study contributes to filling the literature gap for other researchers and practitioners, who may consequently be able to provide implementation strategies. The target audience is thus researchers, developers and practitioners studying and implementing distributed systems.

The benefits are significant in that this study provides a clear understanding of the desired elements involved in the effective teaching and learning of distributed computing in distance-based educational environment for new and inexperienced practitioners. All in the target group, being researchers, developers and practitioners, will benefit, since the research identifies factors contributing to the success of teaching distributed computing in a distance-based educational environment. For researchers and developers, this study provides implementations in different technologies and languages that can serve as a basis for further development.

1.10 Synopsis

The proposal of a new model to direct the effective teaching and learning process of distributed computing in a distance-based educational environment consists of seven chapters. Table 1-1 outlines the content of the chapters of this study.

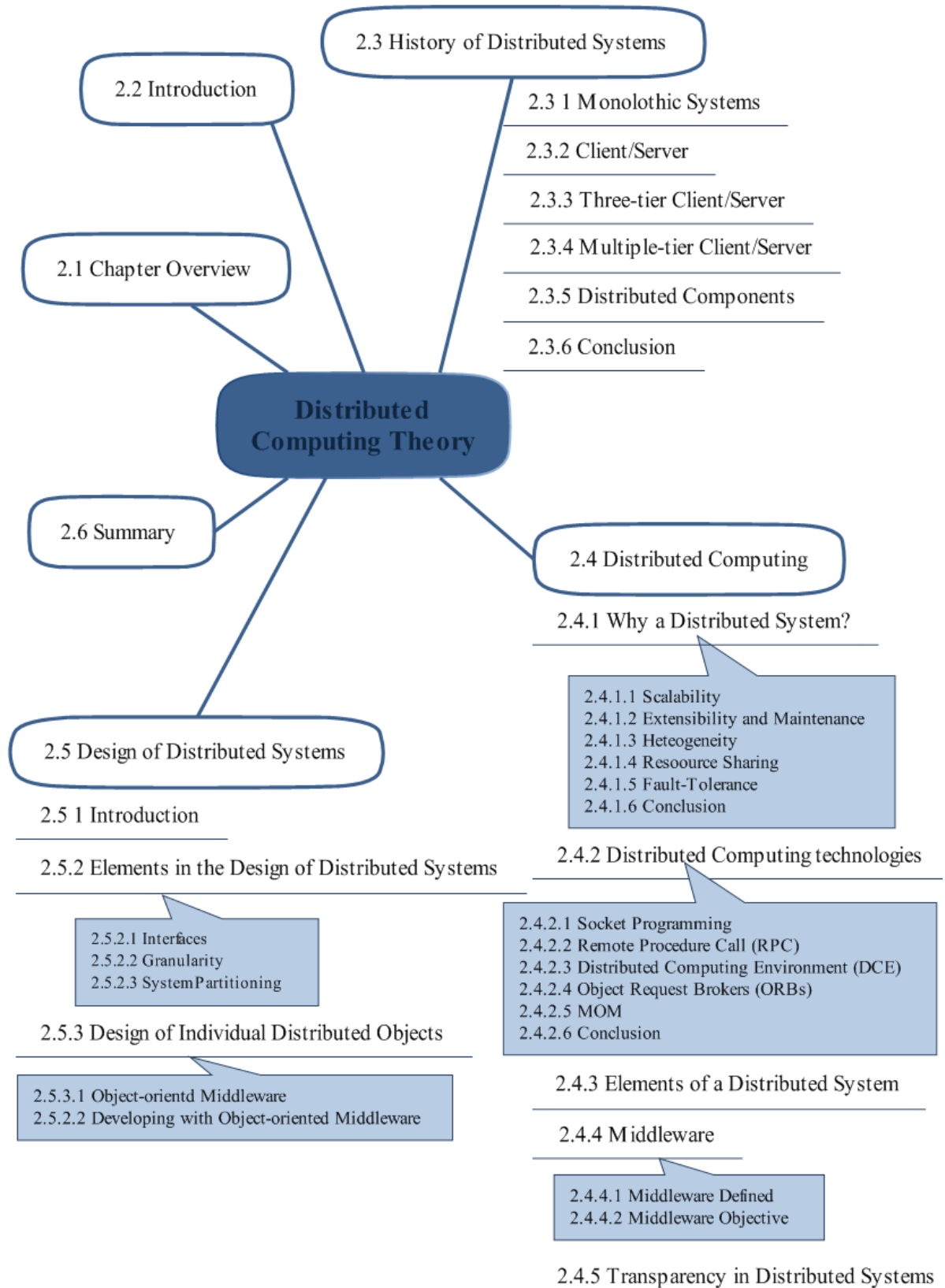
Table 1-1. Outline of Chapters

Chapter	Chapter Overview
1	<p style="text-align: center;">Introduction</p> <p>Chapter 1 introduces the study, provides background to the study and gives an overview of the strategies followed in this study.</p>
2	<p style="text-align: center;">Distributed Computing Theory</p> <p>Chapter 2 establishes a theoretical framework of the elements involved in distributed computing. The chapter provides brief coverage of the history of distributed systems and distributed system requirements; it also provides an overview of middleware and concludes with the design and implementation process for a distributed object.</p>
3	<p style="text-align: center;">Teaching Models</p> <p>Chapter 3 establishes a theoretical framework of the features, characteristics and elements necessary for the teaching and learning of distributed computing in a distance-based educational environment. The chapter starts with an investigation of available models for the teaching of computer programming; this is followed by an investigation of available models for the teaching of programming in a distance-based educational environment. An investigation of the teaching of distributed computing concludes this literature study.</p>
4	<p style="text-align: center;">Research Methodology and Design</p> <p>Chapter 4 motivates and describes the research design used for this study. Before describing and motivating the research design, the chapter provides relevant background information to contextualise the methods that influenced and contributed to the research design.</p>

Chapter	Chapter Overview
5	<p data-bbox="836 277 1011 313" style="text-align: center;">The Model</p> <p data-bbox="480 367 1369 667">Chapter 5 proposes a teaching model, based on both the known elements and newly identified elements needed for effective learning and teaching of distributed computing in a distance-based educational environment. First, the <i>independent distributed learning model</i> is presented as the Resource Space and Learning Space, consisting of a Student Resource and Workspace and a Teacher Resource and Workspace. The model components are then described by employing use-case diagrams to illustrate the users and their interactions with the system at a very high level of abstraction.</p>
6	<p data-bbox="764 741 1083 777" style="text-align: center;">Case Study: UNISA</p> <p data-bbox="480 831 1369 996">Chapter 6 implements the <i>independent distributed learning model</i> at a distance-based institution of higher education. The model is applied through a case study. After participant selection, several cases are discussed. A survey is then used to collect, evaluate and analyse the data. The chapter concludes with the reporting of the survey results.</p>
7	<p data-bbox="743 1072 1102 1108" style="text-align: center;">Discussion of Findings</p> <p data-bbox="480 1162 1369 1229">Chapter 7 summarises the study and its findings, discusses the findings and provides a conclusion.</p>

2. Distributed Computing Theory

2.1 Chapter Overview



2.2 Introduction

In order to address the primary research question, namely *How should distributed computing be taught in a distance-based educational environment?*, the first subsidiary question was defined as *What is a distributed system, and what are the aspects to be considered in designing a distributed system?* The focus of this chapter is to address this question and provide some background on the distributed computing discipline.

Computing, as a discipline, is defined by Computing Curricula (2005) as follows:

In a general way, we can define computing to mean any goal-oriented activity requiring, benefiting from, or creating computers. Thus, computing includes designing and building hardware and software systems for a wide range of purposes; processing, structuring, and managing various kinds of information; doing scientific studies using computers; making computer systems behave intelligently; creating and using communications and entertainment media; finding and gathering information relevant to any particular purpose, and so on. The list is virtually endless, and the possibilities are vast.

A *distributed system*, as defined by Emmerich (2000) is:

A Distributed System is a collection of autonomous hosts that are connected through a computer network. Each host executes components and operates a distribution middleware, which enables the components to coordinate their activities in such a way that users perceive the system as a single, integrated computing facility.

For the purpose of this dissertation, the above two definitions are used.

To design and develop a distributed system, it is important to understand the aim of what is to be constructed. In order to understand and define ‘what is to be constructed’, it is necessary to identify the characteristics of distributed systems. In this chapter, the focus is, first, on the history (evolution) of distributed systems in section 2.2. An overview of distributed computing is provided by investigating the requirements of a distributed system and the technologies that make distributed computing possible. The elements of a distributed system are then identified, followed by a discussion of middleware and transparency. This section concludes with an in-depth look at the design process of a distributed system.

2.3 History of Distributed Systems

When a distributed system is constructed, the various components that interact increase the complexity of the final system, and this complexity is addressed by good software systems. The term software systems refers to the types of application created, for example, a monolithic or client/server application. The history of distributed systems shows an evolution of software systems to accommodate the development of distributed systems. This evolution ranges from monolithic applications to distributed components. Bennis (1999) identified these software systems as the five generations of distributed systems, as illustrated in Figure 2-1. The sections that follow discuss each of the five generations.

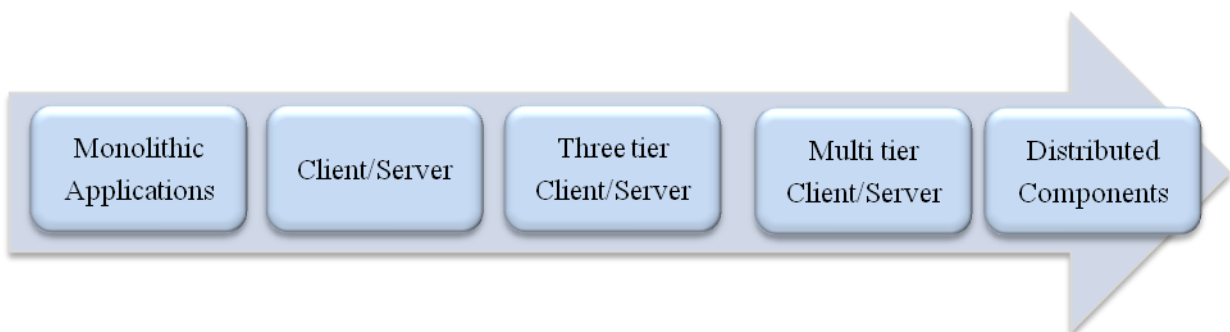


Figure 2-1. Evolution of Distributed Systems (Balen, 2000)

2.3.1 Monolithic Applications

The computing era started with the introduction of the mainframe, being a large data-processing system. The costs involved in mainframes were high, but mainframes were capable of serving large numbers of users and had the advantage, or disadvantage, of being centrally managed. Earlier software systems were often written to be monolithic. In other words, the user interface, logic and database access were all contained in one large application.

2.3.2 Client/Server

The mainframe was followed by the personal computer (PC). The PC is a small digital computer, based on a microprocessor and designed to be used by one person at a time, also referred to as a microcomputer. This brought about a dramatic paradigm shift. The capabilities of the PC changed in that generation from microcomputers to workstations. The networking capability of a workstation and fact that a workstation could run under a multitasking operating system led to the client/server architecture. Client/server describes the relationship between two computer programs whereby one

program, the client, makes a service request from another program, the server, which fulfils the request.

As a result of this reduction of processing required by a ‘central’ machine – the mainframe - the use of cheaper servers allowed not only more widespread deployment, but also the use of servers within smaller teams, instead of a low number of centrally managed mainframes for an entire company. The classic approach of client/server technology was to separate some application back-end functions, specifically those dealing with information management (for example, the database), from the other functions in the application.

2.3.3 Three-tier Client/Server

A typical client/server application splits the application’s components such that the database resides on the server, the user interface resides on the client, and the logic resides on either or both of these. This ‘two-tier’ split means that any change to the client components will result in new copies of the software application being distributed to all users of the system. This led to a three-tier client/server architecture whereby the system is partitioned into three logical groupings: the user-interface layer, the logic layer and the database-access layer.

2.3.4 Multiple-tier Client/Server

Through the insulation and separation of the layers, the two-tier and three-tier architecture enhances the multi-tier client/server architecture. The result is that the user-interface layer communicates only with the logic layer, never directly with the database-access layer. For that reason, many logic or database-related changes could be carried out without impacting on the clients, as long as the interface between the client and business logic layer remained consistent.

2.3.5 Distributed Components

Objects² and distributed objects take the multi-tier architecture to the next evolutionary stage. A multi-tier architecture differentiates between logic and data access. The distributed object model exposes the functionality of the whole application as objects, each of which, in turn, can use any of the functionality of the other objects in the system, or even objects in other systems. The concept of ‘clients’ and ‘server’ also changes. A client component can create objects that behave in server-like

² Objects include data and the procedures necessary to operate on that data.

roles. Therefore, the application of the object-oriented methodology to an n -tier environment results in distributed component-based systems.

2.3.6 Conclusion

The construction of each of these systems would not have been possible without the support of the underlying technologies. As the level of abstraction embodied within those technologies increases, so does the level of sophistication of the systems that is produced. This is illustrated in Figure 2-2.

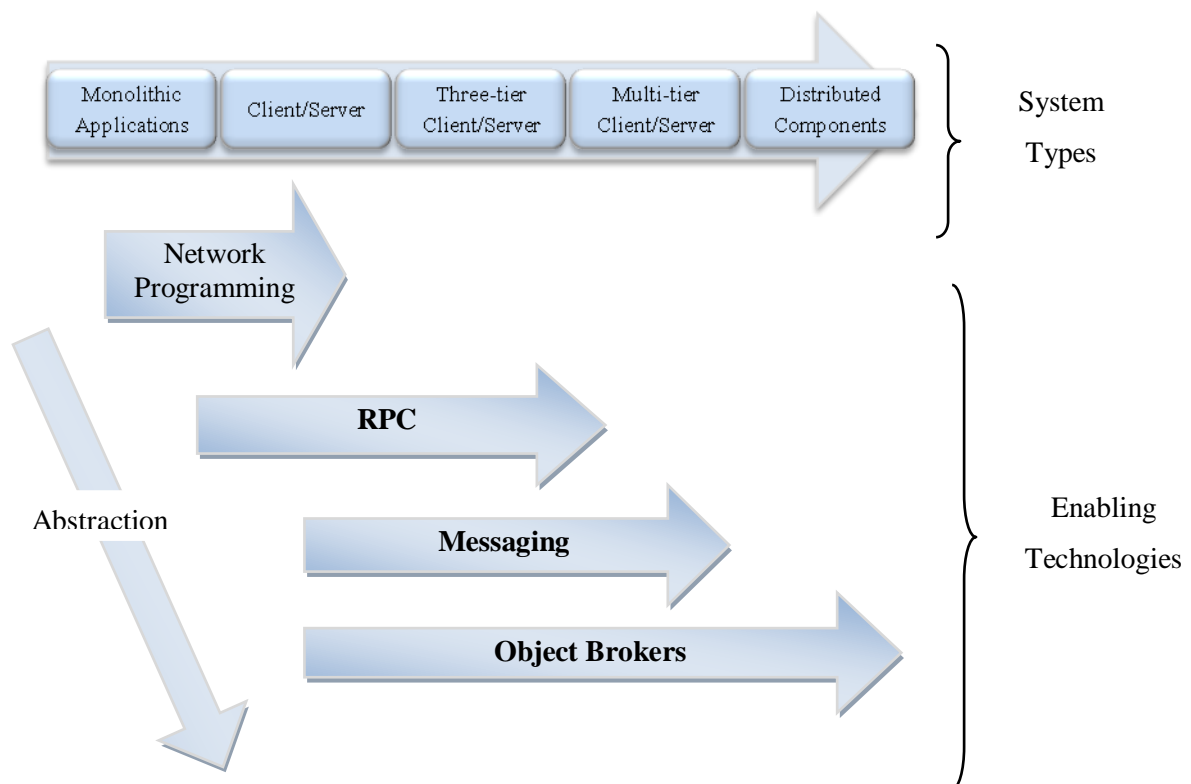


Figure 2-2. Evolution of Distributed Systems

2.4 Distributed Computing

2.4.1 Why a Distributed System?

In a centralised system, all communication is done through a single central computer, and the system relies on a few central components. A centralised system is thus considerably simpler and, therefore, cheaper to build than a distributed system. It would therefore be beneficial not to build a distributed system if it can be avoided. Some non-functional requirements, however, cannot be achieved by a centralised system. According to Emmerich (2000), the non-functional requirements that typically lead

to the adoption of distributed system architectures include scalability, extensibility and maintenance, heterogeneity, resource sharing and fault tolerance.

2.4.1.1 Scalability

Software architectures should be designed in such a way that they are stable for the lifetime of the system. The architecture, therefore, not only has to be able to bear the load when the system goes into production, but a prediction has to be made about how the load will develop during the lifetime of the system. If architecture can grow as the load increases, the architecture is referred to as scalable, since it can accommodate the growth, be it expected or not. Scalability requirements often lead to the adoption of distributed system architectures. Distributed systems can be scalable, because additional computers can be added in order to host additional components. (Balen, 2000) (Eaglesfield, 1999) (Emmerich, 2000)

2.4.1.2 Extensibility and Maintenance

A non-functional property that is often demanded from software and system architecture is that it be open. Openness means that the system can easily be extended and modified. To facilitate openness, the system components need to have well-defined and well-documented interfaces. The system construction needs to adhere to recognised standards. This allows for system components to be exchanged and for the system not to become dependent on a particular vendor. Openness and distribution are related. Components in a distributed system have to declare the services that they offer so that other components can use these services. They can be made for the same or a different machine. The services can be executed synchronously or asynchronously, and the requester and the server can be constructed in a heterogeneous way. (Benns, 1999) (Emmerich, 2000)

2.4.1.3 Heterogeneity

A frequent requirement for new systems is that they have to integrate heterogeneous components. Heterogeneity arises for a number of reasons. Often, components are bought off the shelf. Components that are constructed anew might have to interface with legacy components that have been used in an enterprise for a long time. Components might be built by different contractors, using different technology for the implementation of services, for the management of data and for the execution of components on hardware platforms. Integration of heterogeneous components, however, implies the construction of distributed systems. In order to appear as an integrated whole to users, components have to communicate via a network, and heterogeneity has to be resolved during that communication. (Benns, 1999) (Emmerich, 2000) (Sadiq, 1998)

2.4.1.4 Resource Sharing

Resources denote hardware, software and data. Resource sharing is often required in order to render an expensive resource more cost-effective and as a consequence of communication and co-operation between users. Access to shared data is handled by introducing users, and systems have to validate that the users actually are who they claim to be. Resource access is controlled by implementing resource managers. A resource manager is a component that grants access to a shared resource. Resource managers and their users can be deployed in various ways in distributed system architecture. (Bennis, 1999) (Emmerich, 2000)

2.4.1.5 Fault Tolerance

Hardware, software and networks are not free of failures. They fail because of software errors, failures in the supporting infrastructure (power supply or air conditioning), abuse by their users, or, simply, ageing hardware. An important non-functional requirement that is often demanded from a system is fault tolerance. Fault tolerance demands that a system continue to operate, even in the presence of faults. Ideally, fault tolerance should be achieved with limited involvement of users or system administrators, people being an inherent source of failures themselves. Fault tolerance is a non-functional requirement, which often leads to the adoption of distributed system architecture. (Balen, 2000) (Emmerich, 2000) (Siegel, 1996)

2.4.1.6 Conclusion

The requirements discussed above can be achieved only if technology is available to make it possible. As seen in the previous sections, technologies have evolved to enable the creation of distributed systems. The next section looks into the distributed computing technologies that are available.

2.4.2 Distributed Computing Technologies

When designing and implementing distributed applications, a developer has to consider a number of alternative approaches based on technologies. The choice of how remote interaction between software components is to be supported will depend on the nature of the application, ranging from its complexity to the platform(s) on which it runs and the language(s) used to implement it. The hierarchy of distributed computing approaches is summarised in Figure 2-3 below.

2.4.2.1 Socket Programming

In most modern software systems, communication between machines, and sometimes between processes in the same machine, occurs through the use of ‘sockets’. A socket is a channel by means of which applications can connect to each other and can communicate. The process of using sockets to

communicate between applications, by writing data to and/or reading data from a socket, is called socket programming. The application programming interface (API) for socket programming is very low level, resulting in the advantage that the overhead associated with an application that communicates in this fashion is very low. There are, however, also disadvantages. The lack of type safety and tool support for socket programming leads to longer development times and a greater testing overhead. Also, socket programming does not easily handle complex data types, or data transfer across different machines or languages. It is for these reasons that, even though socket programming can result in efficient applications, this approach is not suitable for complex applications.

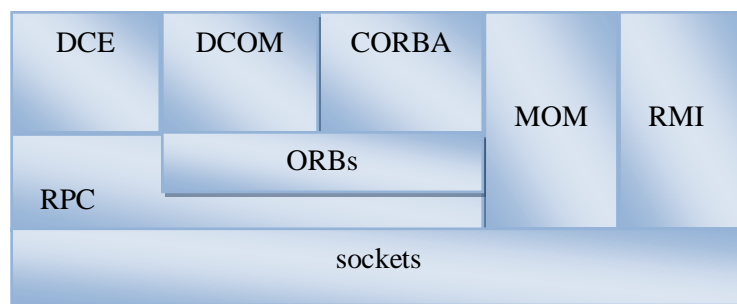


Figure 2-3. Hierarchy of distributed computing approaches

2.4.2.2 Remote Procedure Call (RPC)

Rather than directly manipulating the data flowing to and from a socket, making use of RPC, the developer defines a function and, using a code generator, generates code that makes that function look like a local procedure to a remote caller. In reality, the function uses sockets to communicate with the remote server, which, upon completion, uses sockets to return the result. Because of this procedural interface, RPC is easier to use than raw socket programming. For enterprise-spanning applications, in which both high performance and high reliability are required, RPCs are not a good choice because of their synchronous nature. RPC programming is the base upon which a number of higher level programming models, such as DCE, CORBA, DCOM and RMI, have been built.

2.4.2.3 Distributed Computing Environment (DCE)

The Open Group (formerly known as the Open Software Foundation) pioneered the distributed computing environment (DCE), an industry-standard and vendor-neutral set of distributed computing technologies. The Open Group provides source code for vendors to provide licensed implementation. Even though the DCE makes extensive use of RPC mechanisms, it is much richer in functionality and provides, for example, security, naming, thread, time and directory services. DCE is a highly scalable model based on a standard RPC for organising widely scattered users, services, and data. The DCE

run-time environment consists of hosts that run DCE client software or DCE. DCE provides an excellent technical architecture for distributed computing, but there are drawbacks: it is expensive and requires a high degree of technical competence to implement even simple applications.

2.4.2.4 Object Request Brokers (ORBs)

ORBs can be considered to be language-independent mechanisms. ORBs enable the objects that comprise an application to be distributed, shared and invoked across heterogeneous networks. Because of the way ORBs in which are designed, they are used for applications using a strict object-oriented approach. As with RPCs, in which an interface definition language (IDL) is used to define interfaces to procedures, an IDL is used to define the interfaces between objects. ORBS work best when the interfaces between objects do not change frequently, since changes require code to be recompiled and relinked. As in the case of RPCs, ORBs are generally synchronous, operating in a point-to-point fashion. Examples of ORB's include CORBA, DCOM and RMI, which are discussed in more detail in the sections that follow.

2.4.2.4.1 CORBA

The Common Object Request Broker Architecture (CORBA) is a vendor-neutral standard for middleware. CORBA standardises technology that already exists, instead of trying to invent new technology from scratch. It is very important to distinguish between CORBA as a standard, and middleware products that conform to CORBA (object request brokers – ORBs). Because CORBA is a standard, it specifies only certain aspects of middleware, and many characteristics are therefore product specific.

2.4.2.4.2 DCOM

The capabilities offered by Microsoft's entry into the distributed computing market, the Distributed Component Object Model (DCOM), are very similar to those of CORBA. Since DCOM is integrated into Windows95 and WindowsNT, it enjoys particularly good support on Microsoft operating systems. DCOM is also a relatively robust object model. But because DCOM is not a standard, it is not aimed at heterogeneity or scalability, and its availability outside the realm of the Windows operating system is low.

2.4.2.4.3 RMI

The Java Remote Method Invocation (RMI) is a very CORBA-like architecture. An advantage of RMI is that it supports the passing of objects by value. A major disadvantage, however, is that the programming language is restricted to Java. Consequently, RMI is a good choice for an all-Java application. Any future modification to the system that may require interoperation with applications written in languages other than Java, however, is severely restricted.

2.4.2.5 MOM

Message-oriented middleware (MOM) provides a very simple mechanism for asynchronous, high-performance communication between applications. The basic service offered is the ability to send a parcel of data to a known destination and the ability to receive incoming data. MOM can be viewed as ‘process centric’, since it provides process-to-process data exchange, enabling the creation of distributed applications. MOM is analogous to e-mail in the sense that it is asynchronous, requiring the recipients of messages to interpret their meaning and to take appropriate action. A message usually consists of control information and application data, the control data being used by the messaging software. The single most important difference between MOM and the other approaches, such as ORBs and RPCs, is that MOM does not assume reliable transport layers. As a result, products must provide support for integrity, guaranteed delivery, data-format translation and protection of message queues.

2.4.2.6 Conclusion

Interaction between the individual software components of a distributed system will be determined by the nature and complexity of the application, the platform(s) on which the software components run and the language(s) used to implement them. These communicating software components constitute the elements of a distributed system, which are addressed in the next section.

2.4.3 Elements of a Distributed System

Intuitively, a distributed system has components that are distributed over various computers. A computer that hosts some components of a distributed system is referred to as a host. A host can be seen as the device that includes all operational components of a computer, including hardware and its network operating system software. This is represented in Figure 2-4.

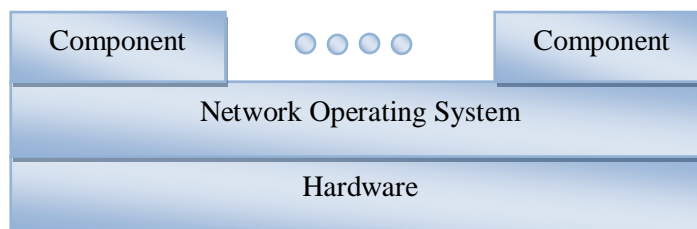


Figure 2-4. Hosts in a Distributed System (Emmerich, 2000)

Since a distributed system has more than one component on more than one host, these components need to interact with one another. Access to each one's services must be provided, and they must be able to request services from one another. In theory, the components could perform this task directly by using primitives, which network operating systems provide. In practice, however, this would be too complex for many applications.

Figure 2-5 shows that distributed systems usually employ some form of *middleware*, which is layered between distributed system components and network operating components, to resolve heterogeneity and distribution (Condi, 1999) (Rock-Evans, 1999).

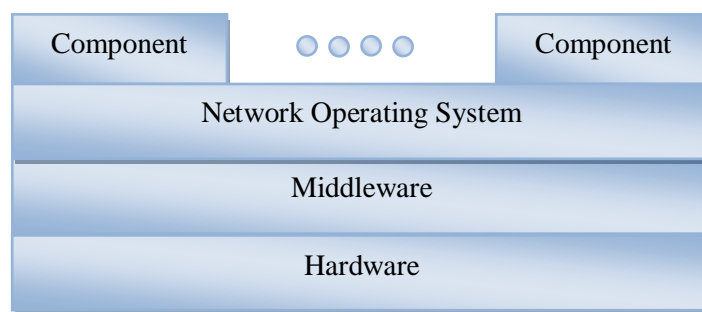


Figure 2-5. Middleware in a Distributed System (Emmerich, 2000)

Middleware provides the application designer with a higher-level of abstraction (Emmerich, 2000) (Rock-Evans, 1999) (Condi, 1999). Middleware implements this higher level of abstraction based on primitives that are provided by network operating systems. While doing so, middleware hides the complexity of using a network operating system from application designers. The Figure below illustrates a working model of a distributed system, as discussed by Emmerich (2000).

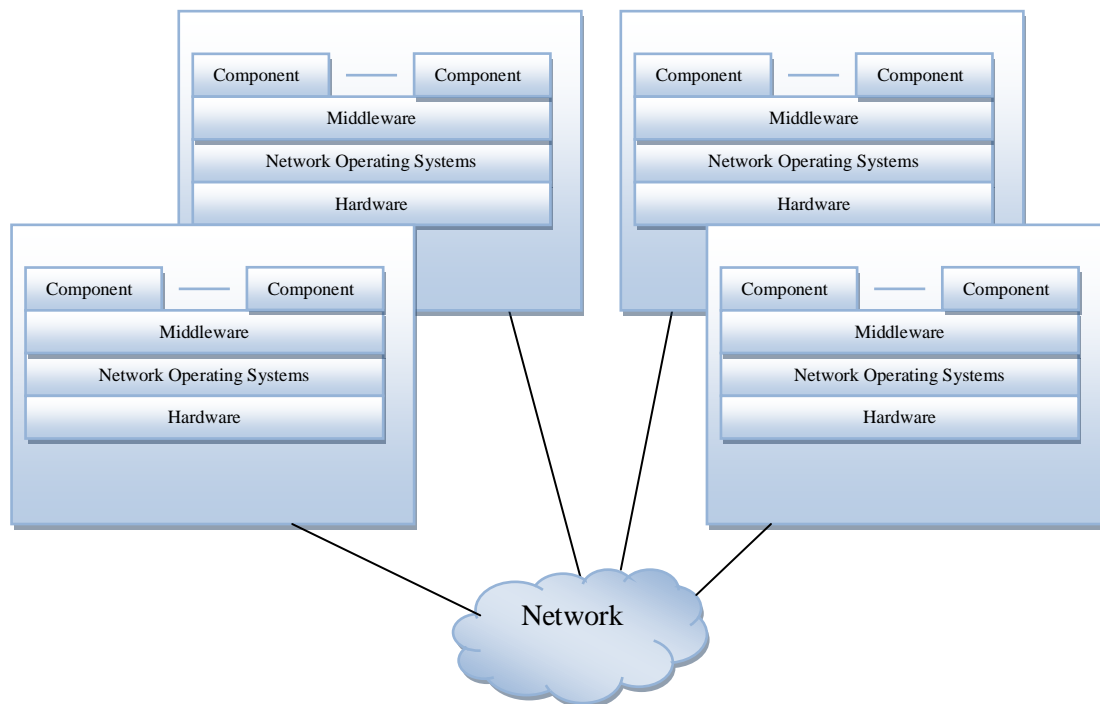


Figure 2-6. Working Model of a Distributed System

A very important property of a distributed system is that distribution must be hidden from users – the user must perceive the system as a single computing facility (Balen, 2000). It is also desirable to hide the complexity of distribution as much as possible from application programmers. This hiding is referred to as *transparency*. If this is achieved, application programmers can develop applications for distributed systems in very much the same way as they developed applications for centralised systems.

The two important concepts identified in this section are transparency and middleware, which need to be elaborated. The next two sections are dedicated to a discussion of these concepts.

2.4.4 Middleware

2.4.4.1 Middleware Defined

Open distributed systems differ from distributed operating systems in a fundamental way, in that they provide support for distributed applications in each host in an open manner. Condi (1999) defines distributed systems *as combinations of hardware, software and network components in which software components execute on two or more processors, communication via a network, providing the necessary application layer support.*

In order to achieve this, a software layer is introduced, which is conveniently termed ‘middleware’, since this layer is *between* the communications facilities provided on each host and the application. Middleware ensures a high degree of independence between the underlying systems and the applications. In order adequately to support the set of distributed applications required for the host to participate in the distributed system, each host’s operating system must be provided with a middleware *software toolset*.

The term middleware is open to a great deal of interpretation. Some consider middleware to be any software that does not implement business logic, whereas others associate middleware more closely with support for distributed computing. Bennis, Crawford and Touman (Bennis, 1999) consider middleware to include:

- Support for production and use of physically distributed systems with appropriate non-functional capabilities (that is distributed computing).
- Support for integration of systems produced using different platforms, languages, middleware products and technologies, and run by different organisations.
- Support for efficient development of business applications, whereby the developer effort required for non-business related aspects is minimised.

In addition, middleware is considered not to include:

- Support for data management and data distribution of the kind provided by DBMS products.

Condi (1999) defines middleware as software that is used to move information from one program to one or more other programs, shielding the application developer from dependencies inherent in communication protocols, operating systems, and hardware platforms. More formally, middleware provides a mediation service between application programs and networks, managing the interactions between disparate applications across heterogeneous computing platforms. Generally, middleware provides a set of services specifically geared towards supporting distributed computing. These services ensure that any distributed application is scalable, reliable, secure and available for use and gives high performance.

Brown (1999) sees middleware as a system component that provides data transfer or real-time access across non-native heterogeneous systems. ... *Middleware is a sort of universal translator and data locator that allows any user, working on any platform, to locate, access, manipulate and move data around in an enterprise without having to understand or be aware of the underlying complexities of the organizations information systems or networks.*

2.4.4.2 *Middleware Objective*

The objectives of middleware can be achieved by the use of a number of different technology approaches, as discussed in Section 2.4.2. Vendors have chosen to implement their particular middleware solutions using one or more particular approaches.

As the name implies, middleware is the layer between the operating system and network, on the one hand, and the distributed application, on the other. The services provided by middleware allow a distributed application to interact with a collection of possibly heterogeneous computers, operating systems and networks, as if they were a single system. The figure below depicts middleware in relation to operating systems, network communications software and application software.

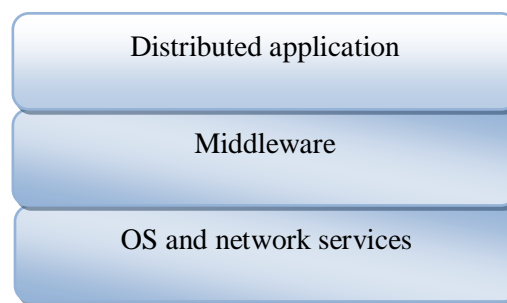


Figure 2-7. Layering of Middleware and Related Software (Benms, 1999)

In order to implement the middleware layer, several technology components must work together. These components provide in a distributed environment what an operating system provides in a centralised environment. In general, a middleware layer's remote procedure call facility consists of both a development tool and a run-time service. The development tool consists of a language and its compiler, which supports the development of distributed applications following the client/server model. Code is automatically generated to transform procedure calls into network messages. The run-time service implements the network protocols by which the client and server sides of an application communicate.

To identify service interfaces and other resources, middleware also includes software for generating unique identifiers. The central repository for information is a directory service, which contains information about all resources in the distributed system. Examples of typical resources are users and machines. The information consists of, *inter alia*, the name of the resource and its associated attributes. Typical attributes might include a user's home directory, or the location of a server.

In middleware, the process of marshalling and demarshalling the input/output parameters to/from a format that can be transmitted across a network takes place without any programmer intervention.

A pillar upon which middleware technologies are built is the concept of the interface definition language (IDL). The IDL specification is a standard language used to define the interface used by objects and is responsible for ensuring that data are properly exchanged between different languages.

Another important concept in middleware is the relationship between the middleware and the network and system services. As shown in Figure 2-7, the middleware is layered on top of the local operating system and networking software. The middleware deployed makes certain assumptions about the services provided by the underlying network and operating systems.

Middleware is layered over one or more transport level services, which are accessed through a transport interface, such as sockets. Middleware assumes that a highly available network physically connects all nodes participating in the distributed environment. The network can be a local area network (LAN), a wide area network (WAN), or a combination of both.

The distributed architecture supports different types of network protocol family. In order, however, for distributed systems to communicate with one another, they must have at least one set of network protocols in common. Middleware also assumes the ability to identify a node with a unique network address, the ability to identify a process with a network end-point address, such as a port. Middleware presumes that certain services are available through the underlying operating system. These services include multitasking, timers, local interprocess communications, basic file system operations, memory management, local security mechanisms, threads and general system utility functions.

This concludes the discussion of middleware. The other important concept discussed in elements of a distributed system is transparency.

2.4.5 Transparency in Distributed Systems

The design of a distributed system architecture that meets all or some of the requirements discussed earlier is rather complicated. The definition used for a distributed system demands that it appear as a single integrated computing facility to users. The fact that a system consists of distributed components should be hidden from users – the system has to be transparent (Baker, 1997) (Emmerich, 2000).

As discussed before, it is also highly beneficial to hide the complexity of distributed system construction from the average application engineer as much as possible. Applications can be

constructed and maintained much more efficiently and cost effectively if they are not slowed down by the complexity introduced by distribution.

There are many dimensions of transparency. Because these dimensions are so fundamentally important, they formed an important part of the International Standard on Open Distributed Processing (ODP) [ISO/IEC, 1996].

As shown in Figure 2-8. Dimensions of Transparency in Distributed Systems (Emmerich, 2000) there are different levels of transparency. The criteria at lower levels support achievement of the transparency criteria at higher levels.

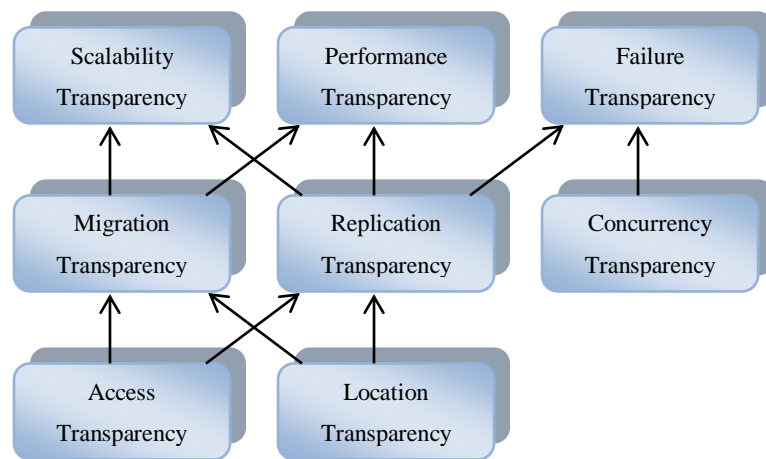


Figure 2-8. Dimensions of Transparency in Distributed Systems (Emmerich, 2000)

A definition of each type of transparency is summarised below:

- **Access transparency** means that the interfaces for local and remote communication are the same.
- **Location transparency** means that service requesters do not need to know physical component locations.
- **Migration transparency** means that a component can be relocated without users or clients noticing it.
- **Replication transparency** ensures that a replica, which is a component copy, remains synchronised with its original.
- **Concurrency transparency** means that users and programmers are unaware of components requesting services concurrently.

- **Scalability transparency** means that users and programmers do not know how scalability of a distributed system is achieved.
- **Performance transparency** means that users and programmers are unaware how good system performance is maintained.
- **Failure transparency** means that users and programmers are unaware of how distributed systems conceal failures.

Transparency provides criteria to be used as a yardstick against which middleware components can be measured. Transparency also helps to review the extent to which middleware systems actually support the application engineer in achieving overall distribution transparency. Finally, transparency helps to review the design of distributed components – it allows a distinction to be made between well-designed components and components that cause problems during operation, administration or maintenance.

2.5 Design of Distributed Systems

2.5.1 Introduction

Building a system based on distributed objects is not the same as constructing a single object-oriented program. The effects of the network, the fact that tasks interact while running in parallel and the possibility of partial failure must all be taken into account. Implementing mechanisms to handle failure and coordination of concurrent processes may be extremely important. The choices made when designing distributed object, however, profoundly impact on the performance and flexibility of the system.

Designing a system of distributed objects and their interfaces is a heuristic process, but there are guiding principles. There is no single best solution; the solution will be determined by the requirements that have been set for the systems. A common desire, however, is to produce a distributed system that is maintainable and flexible.

The purpose of this section is to consider the design process of a system of distributed objects. Firstly, the design elements that have to be considered when such a system is designed is described; then, the design of individual distributed objects is discussed.

2.5.2 Elements in the Design of Distributed Systems

The production of distributed object systems introduces new factors into the design process. Initially, when distributed objects in a system are designed, the primary concerns will be each object's *interface*, the object model's *granularity* and how the complete system is *partitioned*.

Each of these concerns has an effect on the others. It is difficult to deal with issues of interface design, granularity and partitioning in isolation. A well-designed system will balance the forces emanating from each of these areas, providing interfaces at a suitable level of abstraction and granularity. Each of these aspects will now be discussed in turn.

2.5.2.1 Interfaces

For each distributed object to be accessible, an interface must be defined. This set of interfaces then becomes the public face of the subsystem, and the functionality defined by the subsystem is supported by the implementation within the subsystem. An interface is thus a collection of attributes and operations (Budd, 1997). These subsystems can contain many distributed objects, implying that multiple interfaces can be supported. Thus, the structure of the interfaces will affect the performance and reusability of components of the system.

As explained before, various subsystems are identified during the process of development of the distributed object model, which may be located in disparate parts of the network. Each of these subsystems supports one or more distributed objects, and each of these distributed objects provides a public interface expressed in IDL. Interfaces may represent both entity objects and process objects. An entity object represents a thing, whereas a process object provides an abstraction of a process.

Several issues have to be dealt with when an interface is designed (Balén, 2000), (Budd, 1997). *Strong cohesion* means that the interface should be cohesive when the responsibility of each element is strongly related and focused. *Low coupling* means that individual elements in the interface should preferably not be connected to, have knowledge of, or rely on other elements in the interface. *Exception definitions* refer to the fact that standard methods of failure handling should be well defined for each interface. Lastly, common definitions of interfaces should be clearly identified so that reuse in different objects can be achieved; this is called *polymorphism*.

2.5.2.2 Granularity

Granularity refers to the fineness, or coarseness, of the object model. Fine-grained models are composed of many small objects, each cooperating to provide an implementation that solves the

problem. Coarse-grained modes, on the other hand, provide large controller-like objects, which represent large concepts (Balen, 2000), (Budd, 1997).

If an approach of distributing the full domain object model is taken, a very fine-grained system will be produced. Conversely, if a single interface to each remote process is provided, a course-grained system will result. There is a cost associated with each of these approaches. A fine-grained system will affect performance owing to a high level of interaction: more interactions between objects are required to perform a single task. A course-grained system will introduce problems with cohesion, usually because more than one concept from the domain model within a single interface is embodied. This reduces flexibility, since the interface becomes more restrictive in the ways that it can be used.

2.5.2.3 System Partitioning

A distributed system will probably be decomposed into a set of subsystems. Each subsystem consists of a set of distributed objects, supporting the functionality of that subsystem. Partitioning consists of the choices that are made when functionality is divided into subsystems and the subsequent placement of distributed objects. If these objects are haphazardly scattered across the network, the result will be to an inefficient system, because of the unnecessarily high level of interaction among the objects across the network.

One of the goals when a system is “partitioned” is to create an architecture that will support evolution. New requirements can be incorporated when they are discovered. The system should also be sufficiently flexible to support reuse of components so as to provide new functionality (Emmerich, 2000).

It is useful to regard each subsystem as providing a service. If this approach is taken, the distributed system uses an architecture based on a set of cooperating services. This partitioning also affects the interface choices for distributed objects that comprise the subsystem.

2.5.3 Design of Individual Distributed Objects

The design and development of distributed objects is facilitated by object-oriented middleware.

2.5.3.1 Object-oriented Middleware

Object-oriented middleware evolved more or less directly from the idea of remote procedure calls. The first of these systems was the OMG’s Common Request Broker Architecture (CORBA). Then, Microsoft added distribution capabilities to its Component Object Model (COM) and Sun provided a mechanism for Remote Method Invocation (RMI) in Java. The development of object-oriented

middleware mirrored similar evolutions in programming languages - object-oriented programming languages, such as C++, evolved from procedural programming languages as C. The idea is to make object-oriented principles, such as object identification through references and inheritance, available for the development of distributed systems.

2.5.3.2 Developing with Object-oriented Middleware

A development process for distributed objects drives the organisation of this section and is represented in Figure 2-9. Design and Implementation Process for Distributed Systems

Appendix I includes a practical example, implementing all steps.

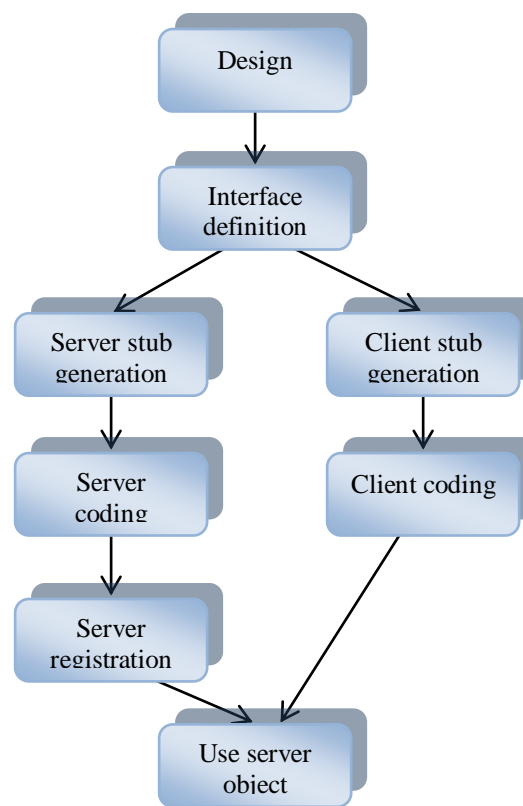


Figure 2-9. Design and Implementation Process for Distributed Systems

2.5.3.2.1 Design

The process starts by the design of the server object based on requirements that existing and future client objects may have. The design could be done using a computer-aided software engineering (CASE) tool that supports an object-oriented design notation, such as the Unified Modeling Language (UML). Hence, class diagrams are designed that show the relationship between server objects and client objects and that indicate the static structure of object types in terms of operations and attributes.

Other tools can be used to model scenarios of how client objects interact with server objects and to indicate constraints on the order in which server object operations can be requested.

The next step of the process transforms the information acquired in the design phase into interface definitions.

2.5.3.2.2 Interface Definition Language

Every object-oriented middleware has an object or meta model. It has an interface definition language (IDL), which has language constructs for all concepts of the respective model. An IDL interface provides a description of the functionality that is provided by an object. An IDL specifies an object's boundaries and its contractual interfaces with potential clients. An interface definition provides all information needed to develop clients that use the interface. Typically, an interface definition specifies the attributes and operations belonging to that interface, as well as the parameters of each operation. An IDL is purely declarative; it does not provide implementation details.

The declarations provided as part of an interface definition are also used by the middleware itself. In particular, for dynamic invocations when requests are defined only at run-time, middleware systems need to safeguard that both clients and the server actually obey the contract. In certain situations, the middleware needs to translate requests between formats used by different middleware systems and usually needs to have type information to do so. Middleware systems store interface definitions for that purpose and provide a programming interface so that interface definitions can be accessed.

Definition of the interfaces between objects is the most important aspect of distributed system design. Therefore, interfaces are the single most important feature of an IDL.

Although all object-oriented programming languages have types or interfaces, interfaces definition have a more specific role for distributed objects; they are the principal source from which middleware systems generate client and server stubs.

2.5.3.2.3 Stub Generation

Client and server object programmers use the interface definition designed in the previous step in order to generate stubs. For a client object, client stub serves as a local proxy for a server object. This means that the client object can call the client stub with local method or procedure calls. For a server objects, a server stub serves as a local proxy for client objects.

Therefore, for a client developer, a stub represents the server objects, whereas for a server object, the server stub represents the client object. Stubs are, in fact, incarnations of the proxy design pattern that

was identified. A proxy is a placeholder, which hides a certain degree of complication. Client and server stubs hide the fact that an operation execution is being requested in a distributed fashion. The client stub resides on the same host as the client object, whereas the server stub resides on the same host as the server object. The implementer of the client object may therefore call the client stub by using a local method call. Likewise, the server stub calls the server object by using a local method call, and the server and client stub implementations implement the distributed communication transparently for client and server objects developers.

Figure 2-10 shows an object request that is implemented using client and server stubs. Since object requests are executed remotely, stubs are utilised to perform marshalling and demarshalling, resolution of heterogeneity and the synchronisation of client and server objects.

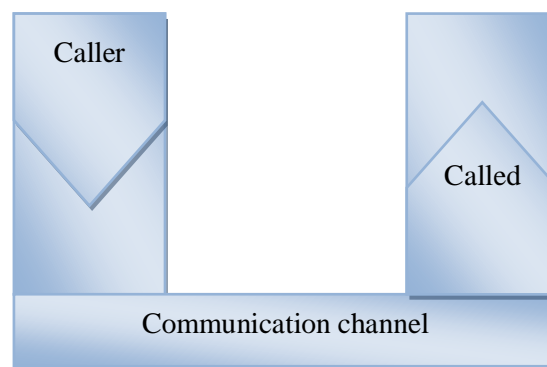


Figure 2-10. Object Requests (Emmerich, 2000)

Stubs are always generated by a compiler, which the middleware provides for the respective interface definition language. Part of every compliant CORBA implementation is a CORBA/IDL compiler, which generates stubs.

2.5.3.2.4 Implementation of Client Objects

Client objects make an object request by invoking a client stub. Hence, a local method to define the request is used. This implies that the request is fully defined at the time that the source code of the client is compiled. Consequently, the request definition is referred to as a static request definition.

The invocation of a stub implies a dependency of the client code on the stub. Hence, the stub has to be generated before the client code can be compiled. Moreover, the client code needs to be recompiled whenever the stub has been changed, for example owing to a change in the interface definition of the server object.

An object request must be type safe. Type safety means that, at run-time, the server object will be able to meet the requests that have been accepted at compile time. Without type safety, there might be run-time errors owing to requests for unavailable operations, or the actual parameters might be wrong. When a client object source is compiled in a type-safe setting, the compiler is expected to detect requests that are not supported by the server object and actual parameter lists that are wrong. This is generally achieved by using statically typed languages such as C++ or Java. Using these languages, the stub class declarations will be imported or included, and the compiler will check any static request definition against the class declaration. Because the client stub class declaration directly evolved from the interface definition, programmers can be certain that they make only requests that are actually supported by the interface of the server object.

For any operation included in the interface, the operation in the client stub generally has the same signature as the operation in the server object. This supports linking the server object directly with the client when it is known at linking time that remote access will not be needed. Therefore, the identity of operation signatures means that static object requests achieve access transparency. From an engineering point of view, it is prudent to define interfaces between subsystems using an interface definition language, even though the interfaces might not be used in a distributed setting. The availability of interfaces, however, simplifies distribution of these components considerably.

The identity of the signatures of client stubs and server objects also enables optimisation by the middleware based on the location of objects. If client and server objects reside on the same host, there is no need to use client and server stubs and the network for the communication of object requests. That would be inefficient. Sophisticated object-oriented middleware implementations, therefore, shortcut the network and invoke the server object directly if it is located on the same host, which is considerably simplified by access transparency.

2.5.3.2.5 Implementation of Server Objects

The implementation of interfaces has to solve the problem of a generated server stub having to call the server implementation in a type-safe way. The compiler that is used must detect typing problems, such as missing parameters or parameters of the wrong type, for the invocation of server objects in the same way as for the invocation of client stubs.

Two approaches are available to implement server objects in a type-safe way. These are interfaces and inheritance. Both approaches achieve calling of the server object by the server stub.

Not all object-oriented languages support the concept of interfaces; these languages may use inheritance. For implementation by inheritance, the middleware generates an abstract class as part of

the server stub. The abstract class has abstract operations that declare the operation signature, but which are not implemented by the class. The class implementing the server object is then defined as a subtype of this abstract class. The server stub will have a reference that is statically of the abstract `**class`, but which refers dynamically to an instance of the sever object.

The programming language compiler used for the server implementation will check that the redefinition of an abstract operation meets the declaration of the respective operations in the abstract superclass. Thus, the inheritance approach is another way of achieving type-safe implementations of server objects.

Interfaces for distributed objects are written using an IDL in a way that is independent of programming languages. Following this approach, the IDL compiler generates a programming language interface that includes all operations exported by the IDL interface. The server object implementation then declares that it is implementing the programming language interface. This approach achieves the required type safety, because the compiler used for translating the implementation of the server object checks that it correctly implements the interface that was derived by the IDL compiler.

2.5.3.2.6 Server Registration

Once the server objects have been compiled, they need to be registered with the object-oriented middleware. The purpose of this registration is to make known where to find the executable code for the object and how to start it. How server implementations are registered depends largely on the particular middleware and on the operating system on which it runs.

A system administrator, who maintains an implementation repository, generally does the registration. Thereby, objects, which are usually identified by an object reference, are associated with some instructions on how to start up the servant in which the server object executes. There is an implementation repository on each host. The middleware provides administrative tools to maintain the implementation repository. These tools enable administrators to browse through the list of servants that are registered on a particular host. Moreover, these tools allow administrators to register new server objects and to delete existing server objects.

2.5.3.2.7 Use of Server Object

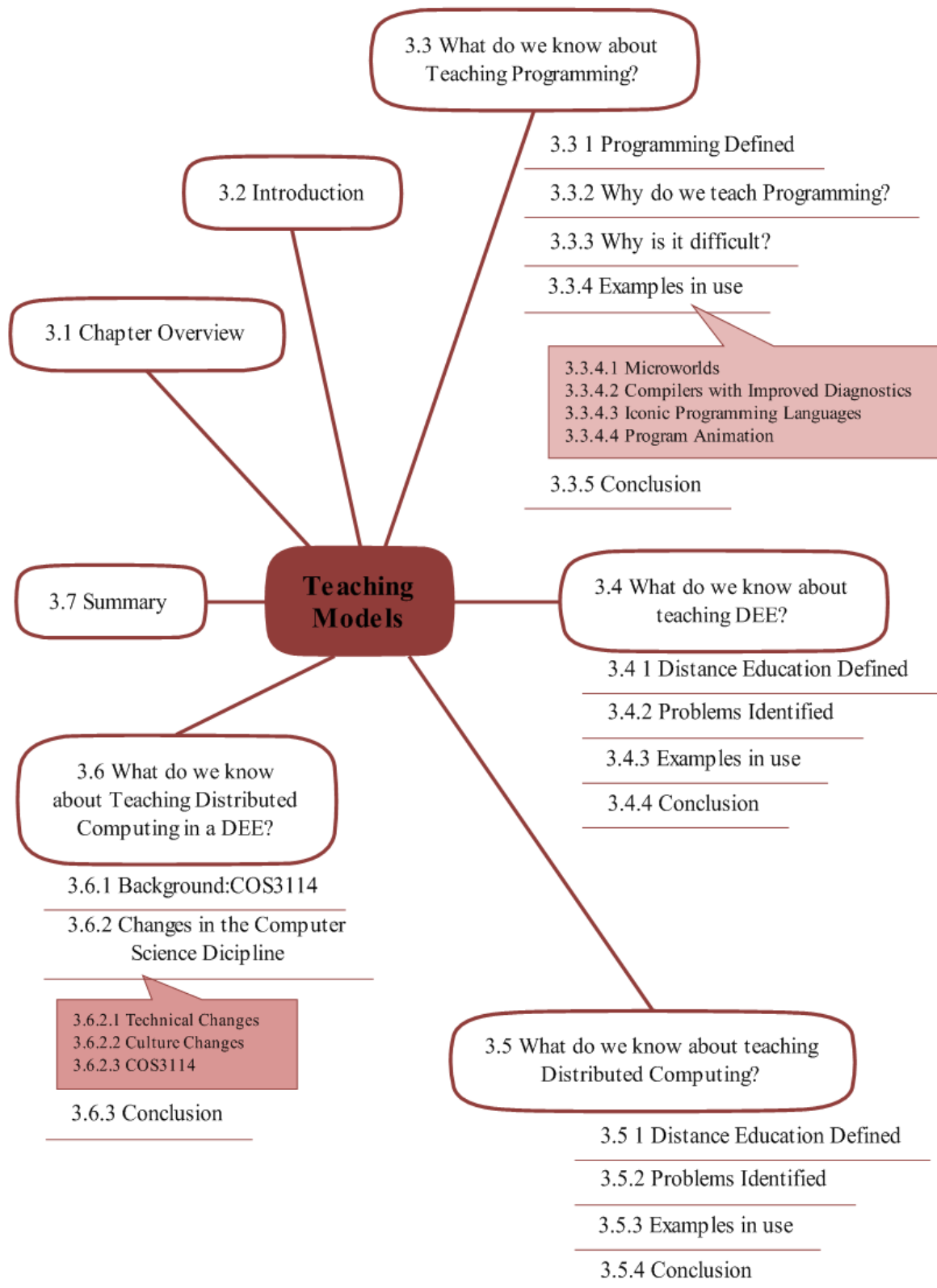
In order to use a server object, a client main function to connect to the server must be created. The client is not concerned with the implementation of the server class, which is the concern of the server programmers. This ensures that the client does not rely on the implementation details within the server.

2.6 Summary

The purpose of this section was to answer the question *What are the factors contributing to the success of teaching programming, and how can these factors be integrated into teaching distributed computing a distance-based teaching model?* The history of distributed systems shows the five generations of software systems and how these evolved to accommodate the design and development of distributed systems. The non-functional requirements of a distributed system were identified as scalability, extensibility and maintenance, heterogeneity, resource sharing and fault tolerance. The various technologies that a developer has to consider when designing and implementing distributed applications were acknowledged. Distributed systems usually employ some form of middleware, which is layered between the distributed system components and the network operating components. Middleware hides the complexity of using a network operating system from application designers. Also, the user must perceive the distributed system as a single computing facility. This hiding is referred to as transparency. This section concluded with a discussion of the design of distributed systems and distributed objects. When distributed objects in a system are designed, the primary concerns are each object's interface, the object model's granularity and how the complete system is partitioned. The development of distributed objects is facilitated by object-oriented middleware and is driven by a development process.

3. Teaching Models

3.1 Chapter Overview



3.2 Introduction

The second subsidiary question, *What are the factors contributing to the success of teaching programming, and how can these factors be integrated into teaching distributed computing in a distance-based teaching model?* is addressed in this chapter. The objective is thus to explore the learning and teaching of programming languages. In section 3.2, typical problems and solutions are discussed with regard to the learning and teaching of programming languages, in environments where there are contact or laboratory sessions, as is the practice in residential universities. This is followed, in section 3.4, by a discussion of learning and teaching programming in a distance-based educational environment (DEE). Similarities to and differences from contact session are explored. The teaching and learning of distributed computing in contact sessions poses new challenges. This is addressed in section 3.5 and is followed by an exploration of these challenges in a distance-based educational environment in section 3.6. The cognitive processes that take place in the teaching and learning programming languages are not addressed, but the tools available to accommodate and ease the learning process are explored.

3.3 What do we know about Teaching Programming?

The objective of this section is to investigate the teaching of programming at an undergraduate level. This section starts with the definition of programming. This is followed by two questions. The first question looks into the necessity of teaching programming, and the second attempts to answer why the teaching and learning of programming is perceived to be difficult. Before a conclusion is reached, several examples in use are investigated. These examples are categorised according to different approaches used to teach programming languages.

3.3.1 Programming Defined

Computer science can be defined as the study of computers and algorithmic processes³, including their principles, their hardware and software designs, their applications and their impact on society (Tucker, 2003).

Programming can be defined as the process used to implement algorithms on computers. A definition of programming by Kelleher (2005) is that programming is the process of assembling a set of symbols

³ An algorithm is a precise, step-by-step description of a solution to a problem.

representing computational actions. By using these symbols, users can express their intentions to the computer and, given a set of symbols, a user who understands the symbols can predict the behaviour of the computer. Van Roy and Haridi (2002) broadly define *computer programming* as bridging the gap between specification and running a program. This consists of designing the architecture and abstraction of an application and coding them in a programming language. The discipline of programming has two essential parts: a technology, which consists of tools, techniques, and standards, and allowing one to *do* programming, and a scientific foundation, which consists of a broad and distance-based educational environment theory with predictive power, allowing one to *understand* programming. The science should be practical, that is, able to explain the technology, making it useful for a practising programmer.

To teach programming, both the science and the technology need to be taught. This process, called the design and implementation of a program, is, however, a difficult activity (Lenarcic, 2003).

3.3.2 Why do we teach Programming?

In the report Computing Curricula 2005 published by the Joint Task Force for Computing Curricula (IEEE-CS, 2005), the first of the three categories identified as career opportunities for computer scientists is the design and implementation of software. The design and implementation of a software program is a difficult and expensive activity, since a very large number of languages, environments and technologies exist for the design and development of software. This is why the teaching (as well as learning) of programming is a challenging and difficult task.

3.3.3 Why is it Difficult?

Xinogalos (2002) stated that programming is without doubt one of the most difficult topics in computer science. He researched practices over a period of time (three decades) and identified that one of the most important factors making programming difficult to learn was the classical approach that was used to teach the principles of programming. This approach is based on the programming language that is used being too big and too idiosyncratic, the use of inadequate programming environments and the problems to be solved being far from the student's everyday experiences.

Research proved that use of the classical approach to the teaching of programming does not fulfil the didactic needs and, as a result, many special methodologies, languages and tools were developed. The four approaches to teaching programming proposed by Xinogalos (2002) are microworlds, compilers with improved diagnostic capabilities, iconic programming languages and program animation (all discussed in section 3.2.4).

Several existing methodologies, languages and tools were researched to determine what was available and to help in the process of identifying the factors contributing to the successes and failures in the teaching and learning of a new programming language. These are presented below, in the next section.

3.3.4 Examples in Use

In this section, an overview is given of each of the approaches identified by Xinogalos (2002): microworlds, compilers with improved diagnostic capabilities, iconic programming languages and program animation. This is followed by a discussion of examples in tabular format, under the headings *Description* and *Goals*. A conclusion is reached at the end of the section.

3.3.4.1 Microworlds

A microworld, according to Watt (1988), is a small exploratory learning environment, based on neat and powerful ideas. A well-designed microworld is both easy to use and interesting to the user. A requirement of microworld programming, according to Capretti (2001), is definition of an environment that includes a friendly graphical user interface (GUI). It is important for the graphic commands to be simple so that they do not interfere with the basic goal of the environment, which goal is the learning of a new language and its concepts. Table 3-1 describe the examples researched.

Table 3-1. Teaching Programming: Microworlds

Description	Goals
<p>An integrated programming environment for teaching the object-oriented programming paradigm was developed by Xinogalos and Satratzemi (2002). It is an integrated programming environment⁴ for teaching the object-oriented programming paradigm, based on the microworld approach of teaching programming and the programming language of Karel++.</p>	<ul style="list-style-type: none"> • To address the difficulties that novice programmers experience when learning to program. • To address the difficulties in learning object-oriented programming. • To reuse the efforts of researchers to develop educational tools that can help students.

⁴ An *environment* in computing according to the Oxford dictionary is *the overall structure within which a user, computer or program operates*. For the purpose of this discussion, the term environment is used to encompass the tool used to accommodate the process of learning to program.

Description	Goals
Orespics (Capretti, 2001) is an environment to learn concurrency and is based on the resolution of 'real life problems'. Students program microworlds using a language based on the integration of the Logo turtle's commands, along with a set of message passing constructs. The rise of the parallel programming paradigm makes concurrency part of undergraduate curricula.	The methodologies and tools to teach concurrency were not adequate at the time of development.
An integrated visualisation-based environment for computer science education (Sugita, 2000) is an integrated teaching environment based on visualisation mechanisms.	<ul style="list-style-type: none"> • To exploit teaching based on the visualisation methodology. • To develop computer science knowledge and practical programming experience.

The goals for using microworlds can be summarised as addressing problems pertaining to content and tools. The content is either difficult, or new, or needs practice. Either existing tools must be reused and packaged, or tools are not available and must be created.

3.3.4.2 Compilers with Improved Diagnostic Capabilities

A compiler identifies syntax errors and violations of language rules in the source program. A diagnostic compiler provides assistance and tolerance for syntactic, semantic and execution errors. The compiler attempts a plausible repair of every source error so that execution may be attempted, and the execution phase continues to repair errors until a user-specified cumulative error count is exceeded. The objective of this perseverance is to maximise the useful life of an incorrect program so that more diagnostic information can be obtained from each program submissions and, hence, to reduce the number of submissions required to achieve successful execution (Conway, 1973). **Table 3-2** summarised the examples investigated.

Compilers with improved diagnostic capabilities are being developed to address the difficulty that introductory computer science students experience when using commercially available compilers. These compilers are often difficult to understand at their proficiency level and often the focus is lost. Students struggle with the feedback that compilers supply, and the real learning issues, such as problem solving, are lost.

Table 3-2. Teaching Programming: Compilers with Improved Diagnostic Capabilities

Description	Goals
<p>Thetis is a programming environment designed specifically for student use, since commercially available compilers are not always well suited to students in introductory computer science courses. The system consists of a C interpreter and an associated user interface, which provides students with simple and easily understood editing, debugging and visualisation capabilities. (Freund, 1996)</p>	<ul style="list-style-type: none"> • To develop a programming environment specifically designed to address the needs of introductory computer science students, since student frustration was less a function of the languages than of the programming environment. • To aid in the feedback that students receive when developing a program.
<p>CAP (Code Analyser for Pascal) analyses programs that use a subset of the Pascal language and provides user-friendly feedback on the errors that it finds. (Schorsch, 1995)</p>	<p>To assist students in finding and fixing syntax, logical errors and style errors in the programming process.</p>
<p>HiC is a development environment based on a subset of C++. Introductory computer science students often have to use professional environments to develop programs. Students find these environments difficult to use because the error messages involve terms that they do not understand. Students can focus on learning a new language; they do not have to struggle with tools that are difficult to understand. (Hasker, 2002)</p>	<ul style="list-style-type: none"> • To focus on learning issues, such as formal syntax, algorithmic problem solving and general computer science. • To obviate the problems that students have with professional environments, which are difficult to understand and hinder the learning process.

3.3.4.3 Iconic Programming Languages

Since “one picture is worth a thousand words”, the approach to teaching programming used in learning algorithm development is an iconic environment free of the issues of syntactic detail (Calloni, 1994). The key to making programming more accessible is to simplify it as much as possible without losing its essence. From the perspective of understanding processes, the essence of programming is algorithm development, which can be simplified to the three structures of sequence, branching and looping. Each of these structures can be represented by a flowchart icon. Iconic programming will then be performed by assembly of these flowchart components into a program (Chen, 2005). The examples researched are described in the table below.

Table 3-3. Teaching Programming: Iconic Programming Languages

Description	Goals
BACCII is used to teach beginning procedural programming. All but the most basic operations are accomplished by using icons. When the algorithm is complete, code can be generated that is syntactically correct. (Calloni, 1994)	To solve problems by using a programming language to implement an algorithm that will ultimately assist in program solving versus teaching a programming language.
Keiron is an iconic query language system that enables the interaction of a novice user with a relational database. (Aversano, 2002)	To help a novice user to learn and comprehend a textual query language, such as SQL, by using an iconic system.
The “Iconic Programmer” is an interactive tool that allows programs to be developed in the form of flowcharts through a graphical and menu-based interface. When complete (or at any point during development), the flowchart programs can be executed by stepping through the flowchart components one at a time. (Chen, 2005)	To isolate the essential aspect of programming and allows it to be taught through an intuitive and visually appealing graphic interface.

In all three cases, the choice of method used to present the learning material was driven by the content: the visual presentation of complex learning material to assist in the leaning process.

3.3.4.4 Program Animation

Program visualisation (PV) is a way of providing programmers with more information about a complex domain of program execution through graphical display of the characteristics and behaviour of all aspects of software. Algorithm animation, with its dynamic visualisation of an algorithm or program, is used primarily as a teaching aid and is aimed at improving teaching processes.

Table 5 summarises the examples investigated when program animation is used in teaching programming.

Table 3-4. Teaching Programming: Program Animation

Description	Goals
<p>WinTK (Program Reading Practice) is used as a lecture style for elementary programming education. Students read the source code of training programs based on computer graphics animation programs and are challenged to modify or extend these programs as instructed by Teacher (Program Re-Write Practice). (Matsuda, 2001)</p>	<ul style="list-style-type: none"> • To improve courses using the C language. • To aid in the understanding of the essence of computer programming.
<p>Jeliot is a family of program animation systems. Jeliot3 is a Java PV tool, which automatically visualises the execution of user-written Java programs. (Moreno, 2004)</p>	<ul style="list-style-type: none"> • To show and teach the basics of programming to the student. • To complete and understand the follow-up assignments by the student at own time. • To be used as an interactive tool in laboratory sessions.
<p>Jan is a Java animation system for program understanding. It is object-oriented animation, since it generates object diagrams similar to the collaboration diagrams known from UML and history diagrams similar to the UML sequence diagrams. The emphasis is on program understanding rather than program debugging. (Lohr, 2003)</p>	<p>To aid in program understanding rather than program debugging.</p>

3.3.5 Conclusion

All research efforts and development of environments to aid in the teaching and learning of programming, especially at introductory level, have one intention: to ease the learning of the programming language by eliminating all complexities that do not directly contribute to achieve the learning objective. *Microworlds* provide complete environments in which a program can be developed. *Compilers with improved diagnostic capabilities* make the feedback of the compiler more understandable for the beginner programmer. The intention of *iconic programming languages* and *program animation* is to facilitate difficult learning content.

To conclude, a diagrammatic presentation of how programming is taught is depicted in Figure 3-1.

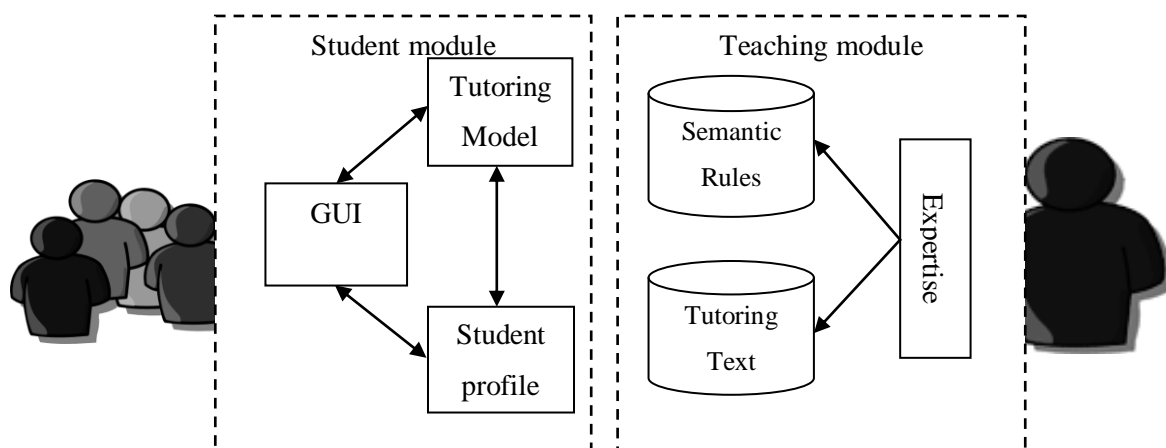


Figure 3-1. Generic Model for Teaching Programming Languages

The model consists of two components: a teaching module and a student module. The teaching module consists of an expertise module, which presents the domain knowledge that the teacher intends to teach to the student. The text that represents the command of the computer programming languages to be taught is maintained in the tutoring text, whereas the semantic rules contain the structure of commands to be taught.

The student module consists of a user interface, which preferably can adapt according to the user's profile. The student profile assesses the student's knowledge state and makes hypotheses about the conceptions and reasoning strategies employed by the student to achieve the current knowledge state, whereas the tutoring module represents the tutoring dialog of the language in question.

3.4 What do we know about Teaching in a Distance-based Educational Environment?

The objective of this section is to investigate the teaching of programming at an undergraduate level in a distance-based educational environment. Before potential and existing problems are identified, distance-based educational environment is defined. Several examples in use are investigated and presented.

3.4.1 Distance-based Educational Environment Defined

Distance-based education, at its most basic level, can be defined as education that takes place when a teacher and student(s) are separated by physical distance, and technology is used to bridge the instructional gap (Gottschalk). Technology can be classified into two types: *traditional tools* (that is,

instructional voice and video tools, data⁵ and print) and *computer-assisted network (Web-based) tools* (that is, World Wide Web applications, whiteboard, chat room and much more) (Hentea, 2003). The printed format still forms the foundational element of distance-based education programs and the basis from which all other delivery systems have evolved.

Although technology plays a key role in the delivery of distance-based education, educators must remain focused on instructional outcomes, not the technology of delivery. The key to effective distance-based education is to focus on the needs of the learners, the requirements of the content and the constraints faced by the teacher, before a delivery system is selected.

3.4.2 Problems Identified

Despite the promise of distance-based learning success, many students who have taken courses at a distance-based institution complain of problems. Hentea (2003) identifies students' evaluations of classes as criticism of assignments or a textbook, frustrations with software and hardware, expressions of feelings of isolation from other students, or the instructor, and concerns about their grades.

Jones (1996) also identifies the above problems under student characteristics, but goes further and categorises the problems. *Student characteristics* are that students encounter problems owing to their desire for flexibility, cultural and time differences and personal characteristics and circumstances. *Institutional characteristics* are the factors that can affect the quality of distance-based education and include the recognition of the importance and support for distance-based teaching, administrative inflexibility and the organisation of distance-based education. Under *education media*, the use of print as the primary medium for teaching in distance-based education contributes a number of problems, including the slowness of distributing subject material, a lack of student/teacher and student/student interaction, and a high level of distance-based transactions. In the category *teacher characteristics*, the commitment of academics to distance-based education plays a vital role. Problems identified in the category *field of study* are that the study of computing consists of three essential paradigms: theory, abstraction and design. In order to experience the abstraction and design paradigms, students must have access to appropriate computing resources. For on-campus students, this access is traditionally provided by computer laboratories. The geographic spread of distance-based students means that this is not an appropriate solution for distance-based education.

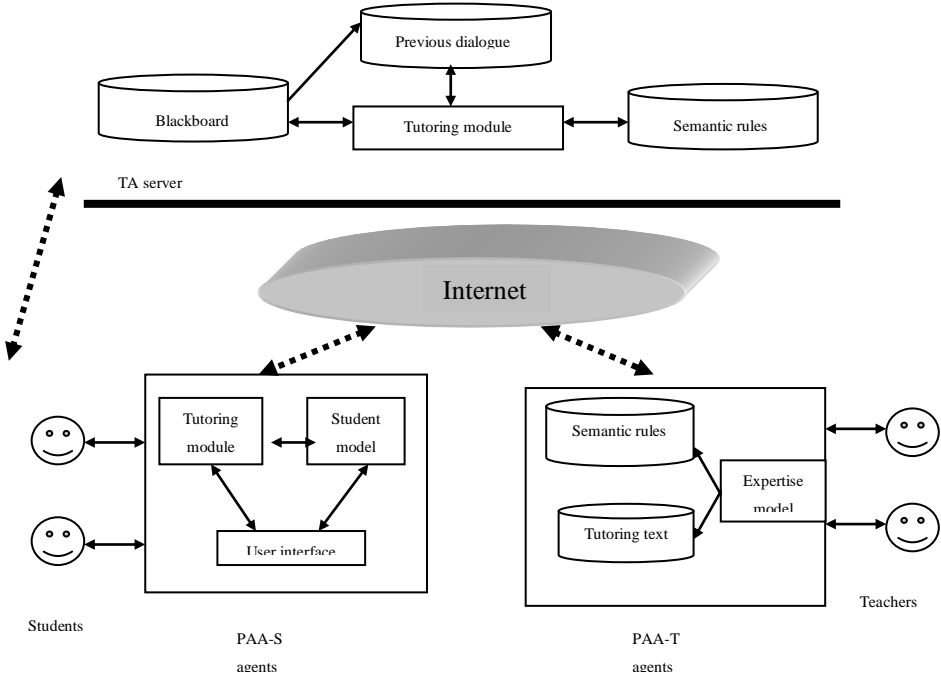
⁵ A wide range of technological options are available and are categorised as follows: Computer-assisted instruction (CAI), computer-managed instruction (CMI), computer-mediated education (CME), electronic mail, fax and real-time computer conferencing.

In the process of gaining an understanding of and seeking solutions to the challenges and problems outlined in the teaching of programming languages in a distance-based educational environment, a number of examples in practice were evaluated.

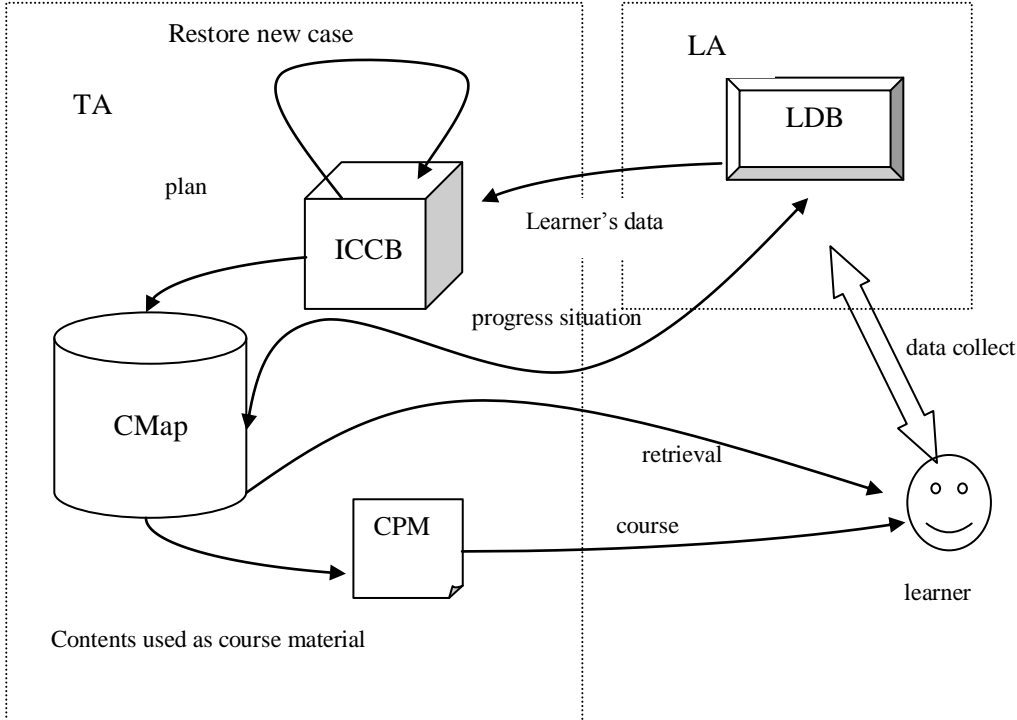
3.4.3 Examples in Use

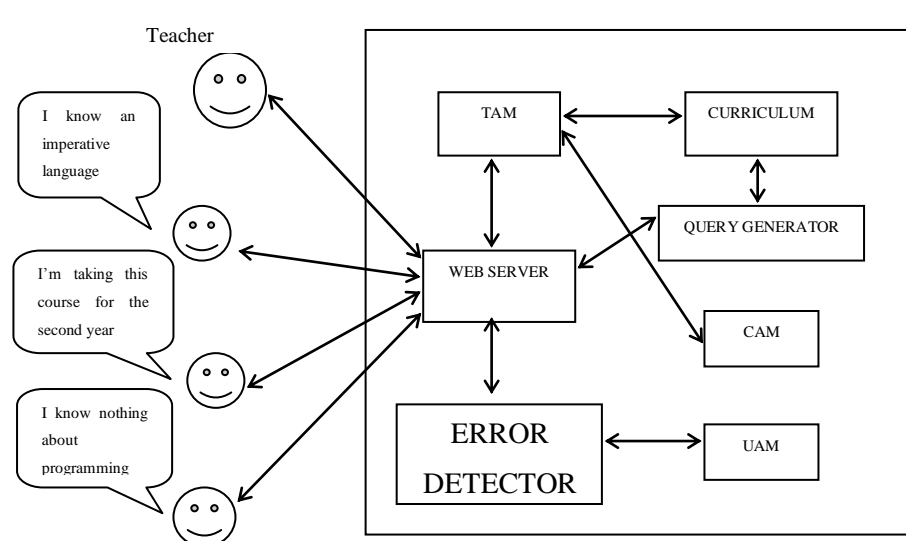
The examples evaluated in the investigation into the teaching of programming languages in a distance-based educational environment are discussed in tabular form, under the headings: System, Description and Components.

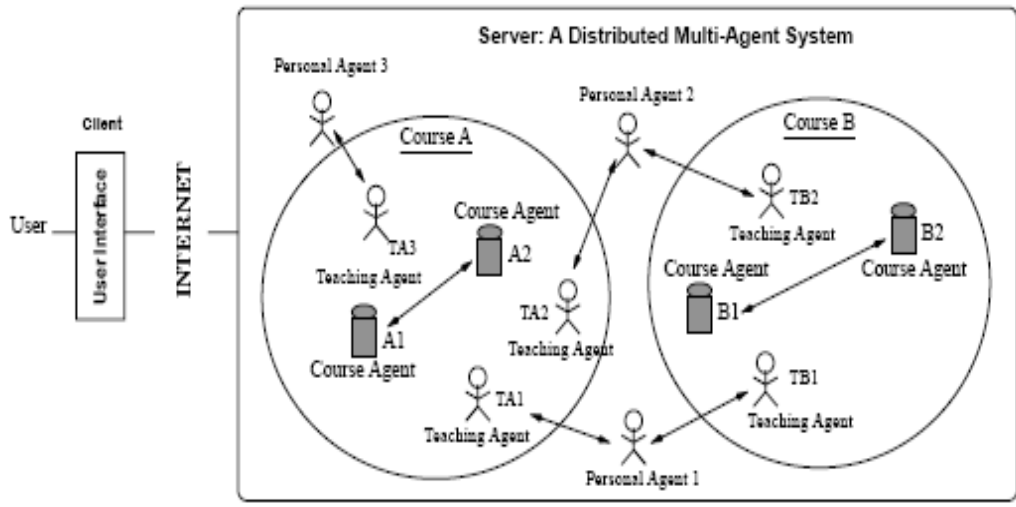
The first example, E-TCL (El-Khouly, 2000), is an expert tutoring system for teaching computer languages through the World Wide Web (WWW). This is followed by CODILESS (Wataba, 1995), which is conceptually based on the group collaboration support model. The third example, SQ3R (Zhang, 2002), is a study method that has been proved to be efficient in traditional education and which is based on the mechanisms of memorising, forgetting and cognitive processes of the human brain. The fourth example is an intelligent tutor for learning the functional programming language Haskell and is called WHAT (Lopez, 2002). IDEAL (Shang, 2001), and the fifth example is a multi-agent system, which supports student-centred, self-paced and highly interactive learning. The sixth example, KnowledgeTree (Brusilovsky, 2004), is an architecture for adaptive E-learning, based on distributed, reusable, intelligent learning activities. The second-last example, MLE (Roccetti, 2001), is a multimedia educational system, which has been designed to provide support for both synchronous and asynchronous distance-based learning. The last example is CIMEL ITS (Fang, 2005); it is an intelligent tutoring system, which provides one-on-one tutoring to help beginners learn object-oriented analysis and design, using elements of UML before any code is implemented.

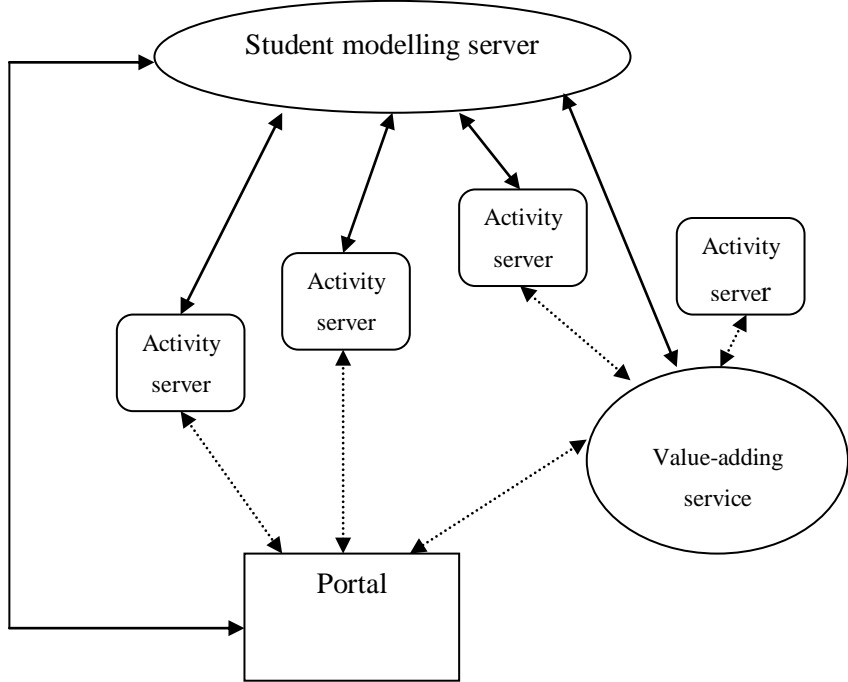
System	1. E-TCL (El-Khouly, 2000)
Description	<p>An expert tutoring system for teaching computer languages through the WWW. Many teachers can cooperate to place the curriculum of one/more computer programming language(s) on the system. Students can access the system through the WWW, select any language that they want to learn, as well as the style of presentation that they prefer, and can exchange their experiences.</p>
Components	<p>E-TCL consists of three agents, representing a client-server relationship:</p> <ol style="list-style-type: none"> 1) Tutoring agent (TA) as a server. 2) Personal assistant agent for teachers (PAA-T) as a client. 3) Personal assistant agent for students (PAA-S) as clients. <p>The PAA-S can communicate with the TA, through the WWW, to retrieve the tutoring dialog of the command(s) that a student wants to practice and to access the experiences of other students in the blackboard module. The PAA-T communicates with the TA to add/modify semantic rules of computer programming languages and to check for correctness of the contents of the blackboard.</p>  <p>The diagram illustrates the system architecture. At the top, the TA server contains a Blackboard, Previous dialogue, Tutoring module, and Semantic rules. The Internet connects this server to two client agents. The PAA-S agent (Personal Assistant Agent for Students) includes a Tutoring module, Student model, and User interface, interacting with two Students. The PAA-T agent (Personal Assistant Agent for Teachers) includes Semantic rules, Tutoring text, and an Expertise model, interacting with two Teachers. Dashed arrows indicate communication between the Internet and both client agents, and between the TA server and the PAA-S agent.</p>

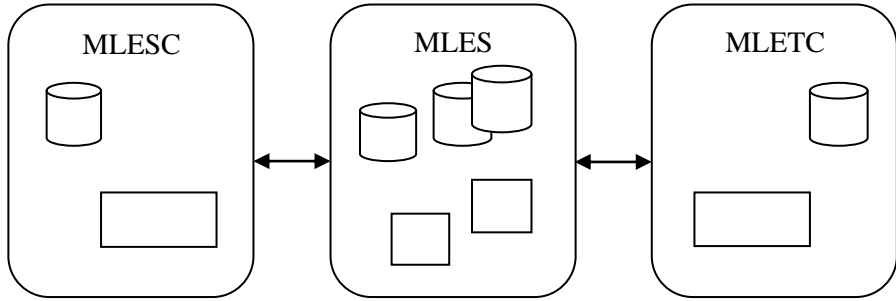
<p style="writing-mode: vertical-rl; transform: rotate(180deg);">System</p>	<p>2. CODILESS (Wataba, 1995)</p>
<p style="writing-mode: vertical-rl; transform: rotate(180deg);">Description</p>	<p>CODILESS is conceptually based on the group collaboration support model. This model provides effective and efficient support for collaborative distance-based learning by integrating synchronous and asynchronous communication and access to learning resources within a single environment. The asynchronous communication can contain text, graphics, images, etc, whereas the synchronous communication can include video, voice and sharing of workspace (such as a whiteboard) for joint work.</p>
<p style="writing-mode: vertical-rl; transform: rotate(180deg);">Components</p>	<p>The group collaboration function unit (GCFU) is meant for processing the knowledge/information stored in the knowledge/information base (KIB).</p> <p>The GCFU and KIB interact as follows:</p> <ul style="list-style-type: none"> • communication support \leftrightarrow communication information base • basic task support \leftrightarrow basic task information base • multiparty conference support \leftrightarrow conference knowledge base • collaborative application support \leftrightarrow application knowledge base <p>The integrated user interface (IUI) serves as a bridge between individual group members and the GCFU.</p>

System	3. SQ3R (Zhang, 2002)
Description	<p>SQ3R is a study method that has been proved to be efficient in the traditional education and which is based on the mechanisms of memorising, forgetting and cognitive processes of the human brain. SQ3R can be introduced as follows:</p> <ul style="list-style-type: none"> • Survey • Question • Read • Recite • Review
Components	<p>The system consists of a</p> <ul style="list-style-type: none"> • Teaching agent (TA) • Learner agents (LA) • TA and LA are constructed based on: <ul style="list-style-type: none"> • Learner Database (LDB) • Content Map (CMap) • Learning Course Case-Base (LCCB) • • course Presentation (CPM) <div style="border: 1px dashed black; padding: 10px; margin-top: 20px;">  <p>The diagram illustrates the system components and their interactions. On the left, the Teaching Agent (TA) is shown with a self-loop labeled 'Restore new case' and a box labeled 'ICCB'. Below the TA is a cylinder labeled 'CMap' and a box labeled 'CPM'. On the right, the Learner Agent (LA) is shown with a box labeled 'LDB'. At the bottom right is a smiley face labeled 'learner'. Arrows indicate the following interactions: 'plan' from TA to ICCB; 'Learner's data' from LDB to ICCB; 'progress situation' from ICCB to LDB; 'retrieval' from CMap to LDB; 'course' from CPM to learner; and 'data collect' from learner to LDB. A note at the bottom left states 'Contents used as course material' with an arrow pointing to CMap.</p> </div>

System	4. WHAT (Lopez, 2002)	
Description	<p>WHAT is an intelligent tutor for learning the functional programming language Haskell. WHAT adapts its behaviour not only individually for each student, but also by considering the performance of similar students. The core of WHAT's adaptive part is based on the classification of students into classes. Thereby, the behaviour of past students of the same class determine how WHAT interacts, in the future, with students of that class. That is, WHAT learns how to deal with each type of student. The main aim of the web-based system is to provide students with an easy-to-use tool, allowing them to practice the knowledge previously gained.</p>	
Components	<p>The <i>Class Adaptive Module (CAM)</i> gathers statistics about the interaction of the users with WHAT. The <i>User Adaptive Module (UAM)</i> deals with concrete information about each individual student. Since the static attributes of each student will be recorded at the beginning of the academic year, the mains tasks of the UAM are to compute the dynamic classes in which a student is included at each moment and to adapt the models created by CAM to fit the characteristics of the individual student better. The <i>Teacher Adaptive Module (TAM)</i> allows the teacher to change the teaching strategies of the system. The TAM allows modification, by hand, of both the design of the curriculum and the basic models for each class of students.</p>  <pre> graph TD Teacher[Teacher] --> WS[WEB SERVER] WS <--> TAM[TAM] WS <--> CURRICULUM[CURRICULUM] WS <--> QG[QUERY GENERATOR] WS <--> CAM[CAM] WS <--> ED[ERROR DETECTOR] WS <--> UAM[UAM] CURRICULUM <--> QG QG <--> CAM ED <--> UAM </pre>	

System	5. IDEAL (Shang, 2001)	
Description	<p>IDEAL is a multi-agent system, which supports student-centred, self-paced, and highly interactive learning. The system is based on a hybrid problem-based and case-based learning model, for both creative and problem solving and mechanical experience simulation. Implemented using the prevalent Internet Web, as well as digital library technologies, the system adopts an open architecture design and is targeted at large-scale distributed operations.</p>	
Components	<p>The system consists of the following elements:</p> <ul style="list-style-type: none"> • Intelligent agents for modelling pedagogy and students, managing and coordinating learning activities, and evaluating teaching results; • a Web-based interface for course delivery, assignment submission, and course-related communication; and • a digital library of student profiles and course materials, including examples and exercises. <p>The multi-agent system consists of five major components:</p> <ul style="list-style-type: none"> • Multi-agent software system • Pedagogical modelling • Web-based interface • Distributed computing environment • Digital library of courseware 	

System	6. KnowledgeTree (Brusilovsky 2004)	
Description	<p>KnowledgeTree is an architecture for adaptive E-learning, based on distributed, reusable intelligent learning activities. The goal of KnowledgeTree is to bridge the gap between currently popular approaches to Web-based education, which is centred on learning management systems versus the powerful but underused technologies in intelligent tutoring and adaptive hypermedia to address both component-based assembly of adaptive systems and teacher-level reusability.</p>	
Components	<p>The architecture assumes the presence of at least four kinds of server:</p> <ul style="list-style-type: none"> • Activity servers • Value-adding servers • Learning portals • Student model servers <p>These servers represent the interest of the three main stakeholders in the modern E-learning process:</p> <ul style="list-style-type: none"> • Content and service providers • Course providers • Students  <pre> graph TD Portal[Portal] --> SM[Student modelling server] Portal --> AS1[Activity server] Portal --> AS2[Activity server] Portal --> AS3[Activity server] Portal --> VAS((Value-adding service)) AS1 <--> SM AS2 <--> SM AS3 <--> SM VAS <--> SM AS1 -.-> VAS AS2 -.-> VAS AS3 -.-> VAS </pre> <p>The diagram illustrates the KnowledgeTree architecture. At the top is the 'Student modelling server' (oval). Below it are three 'Activity server' boxes (rounded rectangles) and one 'Value-adding service' oval. At the bottom is the 'Portal' box (rectangle). Solid arrows point from the Portal to each of the four servers above. Bidirectional solid arrows connect each of the four servers to the Student modelling server. Dotted bidirectional arrows connect each of the three Activity servers to the Value-adding service.</p>	

System	7. MLE (Rocchetti, 2001)
Description	<p>MLE is a multimedia educational system, which has been designed to provide support for both synchronous and asynchronous distance-based learning. By exploiting the potentiality of adaptive techniques, mark-up languages, and networked multimedia technologies, this tool is able to provide each individual student with educational support tailored to his/her personal profile and didactical needs.</p>
Components	<p>MEL consists of three main software components:</p> <p>The MLE Server (MLES), which:</p> <ul style="list-style-type: none"> • maintains the XML-based specification of the knowledge domain structure; • provides run-time support to the dynamic automated process of page creation; • maintains and manages the user mode; and • delivers synchronised multimedia presentations. <p>The MLE Student Client application (MLESC), which provides students with an integrated environment, is useful to display hypermedia contents. The MLESC also provides support to the real-time audio communications among different MLE users.</p> <p>The MLE Teacher Client application (MLETC) provides teachers with an integrated environment suitable for supporting all teacher activities.</p> <div style="text-align: center; margin-top: 20px;">  <pre> graph LR MLESC[MLESC] <--> MLES[MLES] MLES <--> MLETC[MLETC] </pre> </div>

System	8. CIMEL ITS (Fang, 2005)	
Description	<p>CIMEL ITS is an intelligent tutoring system, which provides one-on-one tutoring to help beginners to learn object-oriented analysis and design, using elements of UML before any code is implemented. The Student Model presented supports adaptive tutoring by inferring the problem-specific knowledge state from student solutions, the historical knowledge state of the student and cognitive reasons for the student making an error.</p>	
Components	<p>CIMEL ITS consists of four modules:</p> <ul style="list-style-type: none"> • The Curriculum Model, which represents the knowledge to be taught. • The Expert Evaluator observes the student’s work step by step and compares each step to its own solution(s) to the current problem. • The Student Model maintains a model of the student’s current knowledge state based on information from both CIMEL and the Expert Evaluator. • The Pedagogical Agent provides feedback to the student and tutoring when the student is in need of help. <p>The CIMEL ITS can interact with students either through CIMEL multimedia or the Eclipse IDE.</p> <pre> graph TD Learner((Learner)) -- develops in --> IDE[Eclipse IDE Plug-in] IDE -- Student solution --> EE[Expert Evaluator] EE -- evaluates solution --> SM[Student Model] CM[Curriculum Model] -.-> PA[Pedagogical Agent] CM -.-> EE CM -.-> SM PA -- Reasons --> SM PA -- Strategies --> EE SM -- Student answer --> CMedia[CIMEL Multimedia] CMedia -- learns with --> Learner CMedia -- tutors --> Learner </pre>	

3.4.4 Conclusion

The examples researched are classified as intelligent tutoring systems (ITS). An ITS for effective teaching in a distance-based educational environment is built on the client-server model and is reflected by the examples researched. In its basic form, the client side consists of a lecturing agent or agents and student agents, and the server provides the content and infrastructure needed to present a course.

Hartley and Sleeman (1973) proposed four components of an ITS:

- An *expert or domain knowledge module*, which contains the domain knowledge of a system.
- A *teaching or pedagogical module*, which chooses appropriate teaching strategies.
- A *student module*, which models the student knowledge and behaviour.
- A *graphical user interface module*, which handles the interaction between the user (tutor or student) and the system.

The table below summarises the systems evaluated according to the above four components.

Table 3-5. Teaching in a Distance-based Educational Environment: Summary of Examples

System	Expert or domain knowledge	Teaching or pedagogical module	Student module	Graphical user interface module
E-TCL	Tutoring Agent	Personal Assistant for Teachers	Personal Assistant for Students	
CODILESS	Knowledge Information Base	Group Collaboration Function Unit	Group Collaboration Function Unit	Integrated User Interface
SQ3R	Learner DB Content Map Learning Course Case-Base Course Presentation	Teaching Agent	Learning Agent	
WHAT	Class Adaptive Module	Teacher Adaptive Module	User Adaptive Module	
IDEAL	Digital library	Intelligent Agent	Intelligent Agent	WEB-based interface
KnowledgeTree	Activity servers Value-adding servers	Learning portal	Student Model servers	
MLE	MLE Server	MLE Teacher Client Application	MLE Student Client Application	
CIMEL ITS	Curriculum Model	Pedagogical Agent	Student model	Eclipse IDE

The columns *expert or domain knowledge*, *teaching or pedagogical module* and *student module* in Table 3-5 above show that all systems evaluated include the first three components described by Hartley (1973): an expert or domain knowledge module, a teaching or pedagogical module and a student module. Only three systems employ the last component, the graphical user interface module.

The teaching and learning of programming languages in a distance-based educational environment can be summarised in a model, depicted in the figure below.

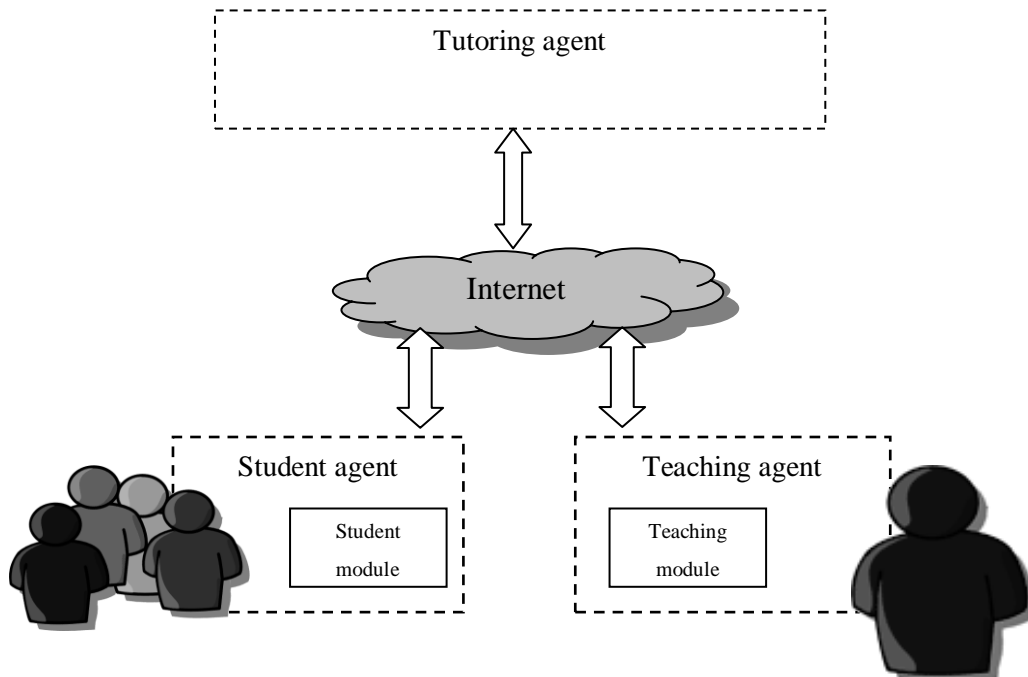


Figure 3-2. Teaching Programming in a Distance-based Educational Environment

A tutoring agent and the Internet have been added to the model presented in Figure 3-1. Generic Model for Teaching Programming Languages. The tutoring agent represents the knowledge to be taught.; it encompasses modelling of the knowledge and management and coordination of the learning activities. The Internet serves as a vehicle for communication.

The student and teacher communicate with the tutoring agent through the Internet, or WWW. Thus, the teaching module is contained in a teaching agent, whereas the student module is contained in a student agent. The student agent retrieves the tutoring dialog that a student wants to learn through the Internet. The teaching agent communicates with the tutoring agent to maintain the knowledge to be taught.

3.5 What do we know about Teaching Distributed Computing?

The objective of this section is to investigate the teaching of distributed computing. The same approach is followed as in the previous section. Before potential and existing problems are identified, distributed computing is defined, and this is followed by an investigation of several examples in use.

3.5.1 Distributed Computing Defined

Distributed computing can be defined in many different ways. General definitions of distributed computing are: *Any computing that involves multiple computers remote from each other that each has a role in a computation problem of information processing* (TechTarget, 2008), or *A type of computing in which different components and objects comprising an application can be located on different computers connected to a network* (Webopedia, 2008).

Distributed computing may also be defined as decentralised and parallel computing, using two or more computers communicating over a network to accomplish a common objective or task. The types of hardware, programming languages, operating systems and other resources may vary drastically.

Distributed computing is a science that solves a large problem by giving small parts of the problem to many computers to solve and, then, combines the solutions to the parts into a solution for the problem. Teaching distributed computing presents several challenges, and the problems identified are discussed below.

3.5.2 Problems Identified

Distributed systems are characterised by highly dynamic, concurrent and complex processes. As a result, training in this area requires great effort from both teachers and students, and the tools and methods available for presentation, explanation and exercises are limited. The motivation of teachers to develop tools to assist in the learning process stem from difficulties identified to facilitate and optimise the teaching of distributed systems.

Schreiner (2002) concludes that even though there are excellent textbooks on the topic of distributed computing, distributed algorithms are described in an abstract manner, rather far away from real programs and the lack of an easy to use and universally accessible platform for implementing the algorithms taught in class and investigating their dynamic behaviour. Since the available distributed message-passing libraries are "heavy weight", their use for education is limited – they force one to deal with a number of low-level technical details and do not allow easy observation of the "internals" of the execution.

Burger (2001) identified the complexities of algorithms and protocols in distributed systems. These complexities are due to the inherent dynamics and intertwining actions at different locations. Hence, the task of conveying material to learners, as well as understanding the material, is very difficult. Furthermore, the development of learning material and tools requires tremendous effort, since the properties of algorithms and protocols are manifold – from correctness to performance. Thus, it does not suffice to show effects in functionality only. In order to reach an overall understanding, all aspects of an algorithm or protocol have to be taken into account.

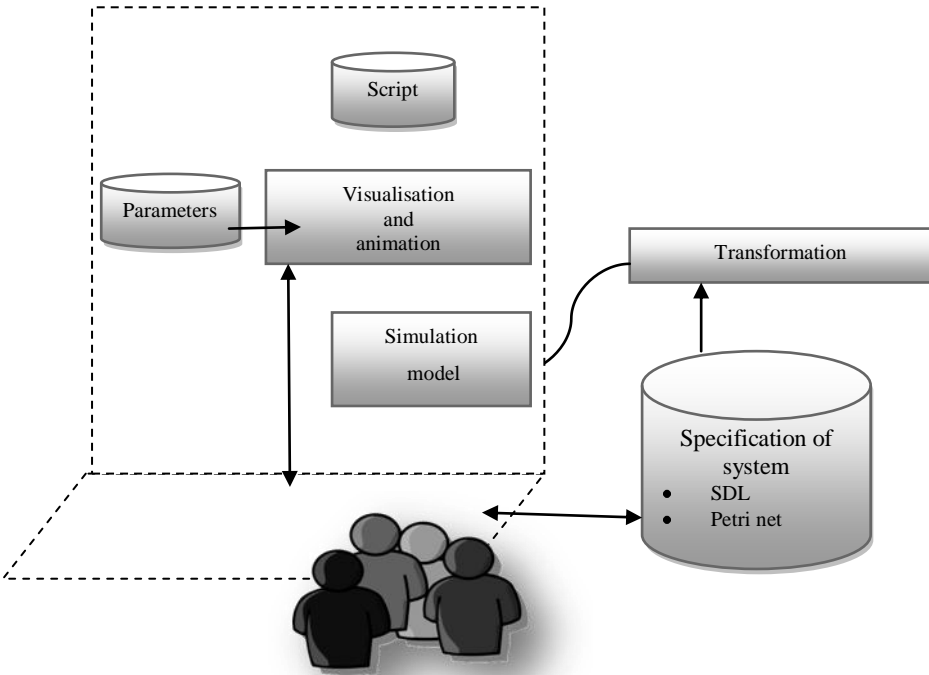
It is extremely important for the strange behaviour that distributed programming can produce to be demonstrated to students (Ben-Ari, 2001). Such programs, by definition, should run on separate central processing units (CPUs), but multiprocessor computers are rare. Although networks are common, it may be difficult to schedule time for demonstrations or assignments. In addition, network programming is tricky and non-portable; it is more suitable for large projects than for ordinary course work.

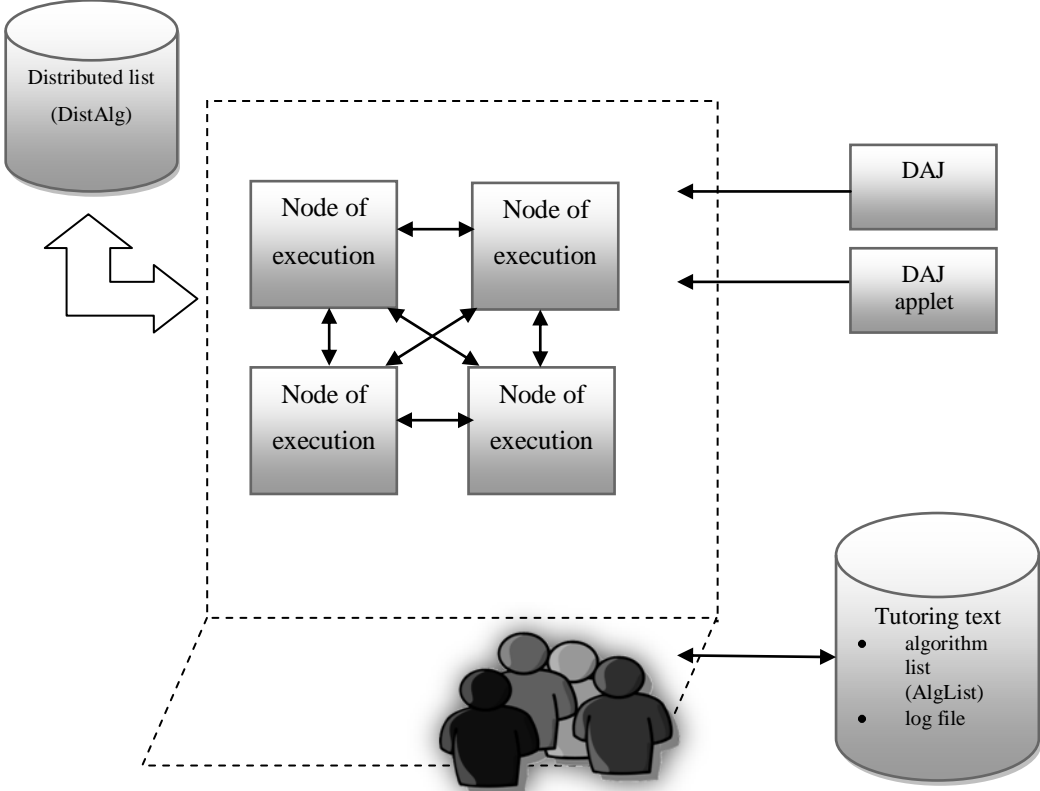
The following section summarises some systems developed to address the challenges in teaching distributed systems, and they are presented in tabular form under the headings: *System*, *Description* and *Components*.

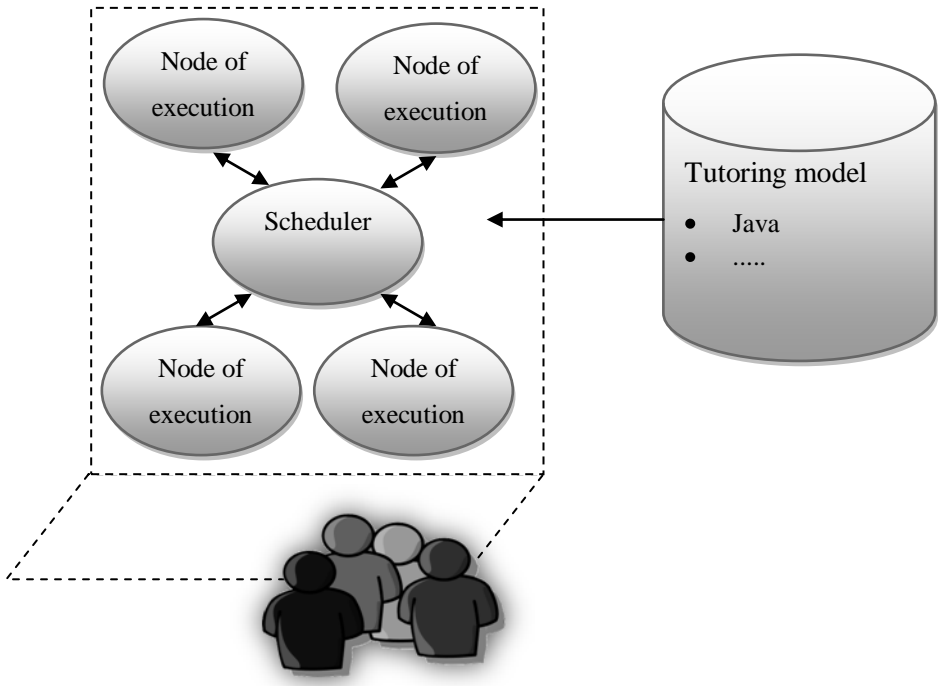
3.5.3 Examples in Use

The examples evaluated in the investigation into the teaching distributed computing are discussed in tabular form, under the headings: System, Description and Components.

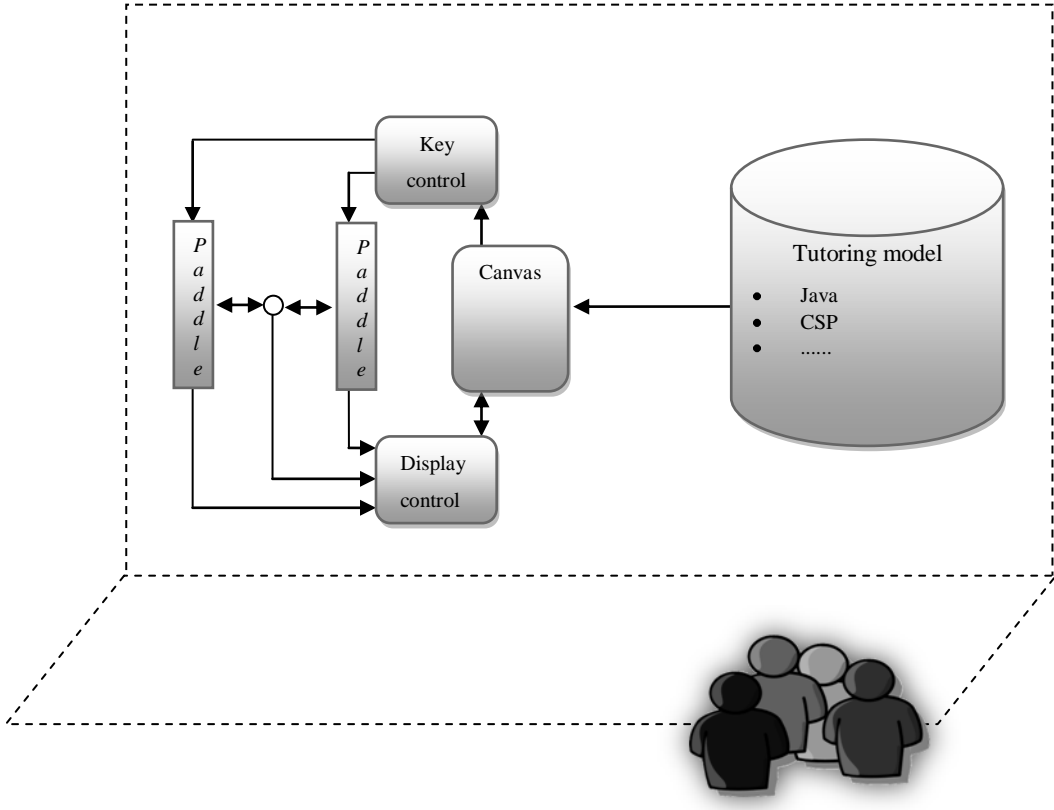
The first example is HiSAP (Burger, 2001). It is an architecture consisting of a framework to build simulations and generate applets from formally specified algorithms and protocols. The second example, DAJ (Ben-Ari, 2001), is a framework for writing Java programs to implement distributed algorithms. The third example is A Java Toolkit for teaching Distributed Algorithm (Schreiner, 2002). It is a toolkit for the development and visualisation of distributed algorithms in Java. The fourth example is VADE (Moses, 1998), an algorithm animation system for distributed algorithms. The second-last example is JCSP (Nevison, 2001), which takes advantage of Java's facilities for concurrent threads and communications among distributed processors and uses the constructs of CSP to manage the complexity of this concurrency. The final example is a simulation and visualisation environment for distributed algorithms, which provides students with an experimental environment to test and visualise the behaviour of distributed algorithms - Lydian (Lydian, 2005).

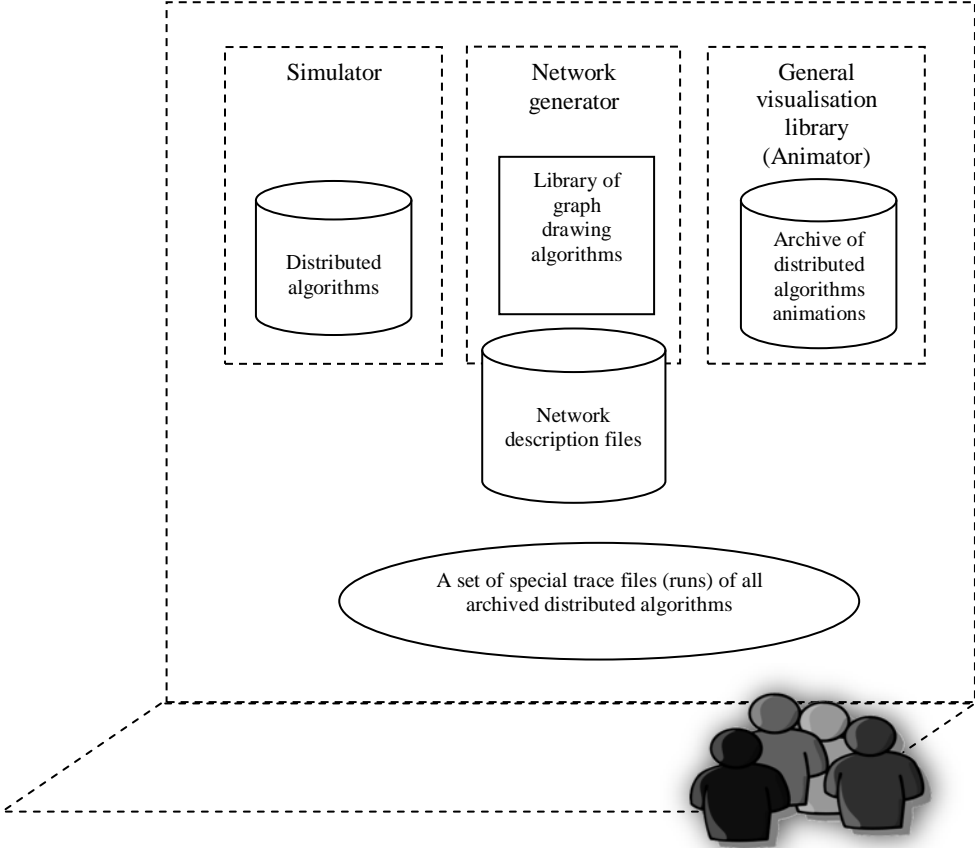
System	1. HiSAP (Burger, 2001)
Description	<p>HiSAP (Highly Interactive Simulation of Algorithms and Protocols) is an architecture consisting of a framework to build simulations and generate applets from formally specified algorithms and protocols. Modification of the specification and observation of the resulting behaviour enables teaching and learning in a constructive manner.</p>
Components	<p>The basic architecture consists of a simulation model. At run-time, it receives input either from a predefined script, or directly from the user. The simulation model runs through an algorithm or protocol step by step and triggers the component for visualisation and animation. The output is displayed to the user. The architecture is also well suited for integrating other tools that take care of properties such as correctness requirements and performance estimations.</p> 

<p style="writing-mode: vertical-rl; transform: rotate(180deg);">System</p>	<p>2. DAJ (Ben-Ari, 2001)</p>
<p style="writing-mode: vertical-rl; transform: rotate(180deg);">Description</p>	<p>DAJ (Distributed Algorithms in Java) is a framework for writing Java programs to implement distributed algorithms. A distributed system is simulated by a collection of Java applets, which are embedded in the same HTML document. The programs display the data structures at each node and enable the user to construct scenarios interactively. Programs have been implemented for commonly taught algorithms. Addition of a program for another algorithm requires only general Java programming experience.</p>
<p style="writing-mode: vertical-rl; transform: rotate(180deg);">Components</p>	<p>The basic architecture consists of a simulation model. At run-time, it receives input either from a predefined script, or directly from the user. The simulation model runs through an algorithm or protocol step by step and triggers the component for visualisation and animation. The output is displayed to the user. The architecture is also well suited for integrating other tools that take care of properties such as correctness requirements and performance estimations.</p>  <p>The diagram illustrates the DAJ architecture components. On the left, a cylinder labeled 'Distributed list (DistAlg)' has a large arrow pointing towards a central dashed box. Inside this box, four 'Node of execution' boxes are arranged in a 2x2 grid, connected by bidirectional arrows. To the right of the dashed box, two boxes labeled 'DAJ' and 'DAJ applet' have arrows pointing into the dashed box. Below the dashed box, a group of three stylized human figures is shown. To their right, a cylinder labeled 'Tutoring text' contains a bulleted list: 'algorithm list (AlgList)' and 'log file'. An arrow points from this cylinder to the human figures.</p>

<p>System</p>	<p>3. A Java Toolkit for teaching Distributed Algorithm (Schreiner, 2002)</p>
<p>Description</p>	<p>This is a toolkit for the development and visualising distributed algorithms in Java. It consists of a Java class library with a simple programming interface, which allows development of a distributed algorithm in a message-passing model. The resulting programs may be executed in standalone mode using a Java interpreter, or embedded as applets into HTML documents and executed by a Web browser.</p>
<p>Components</p>	<p>A distributed system is simulated by a multi-threaded application, which relies on a central scheduler. Each system node is represented by a separate thread, which returns control to the scheduler each time that a message-passing operation is performed. The program executed by each node is a conventional program, written in an imperative message-passing style. The graphical user interface allows the investigation of the states of the nodes and channels by a point-and-click operation.</p>  <pre> graph TD subgraph System S((Scheduler)) N1((Node of execution)) N2((Node of execution)) N3((Node of execution)) N4((Node of execution)) S <--> N1 S <--> N2 S <--> N3 S <--> N4 end TM[(Tutoring model)] --> S TM --- L[• Java •] UI[User Interface] </pre>

<p>System</p>	<p>4. Algorithm Visualization for Distributed Environments (VADE) (Moses, 1998)</p>
<p>Description</p>	<p>VADE is an algorithm animation system for distributed algorithms. VADE is constructed so that the algorithm runs on the server's machine, whereas the visualisation is executed on a web page, on the client's machine.</p>
<p>Components</p>	<p>The algorithm is executed on the server's machines, whereas the animation and the GUI are executed on the client's machine. The client-side processes run at a WWW browser. The various processes are written in Java. The processes on the server side are Java applications, whereas the processes on the client side are Java applets. Their code is downloaded by the WWW browser over the Internet, compiled and run within the browser. This allows Internet users to watch the animation in their browsers. The communication between the server processes and the client processes are performed with the TCP/IP protocol.</p> <p>The diagram illustrates the architecture of VADE. At the top, a 'Main client' box labeled 'WWW browser (java applets)' is connected to a 'Main server' box labeled '(Java applications)'. A cloud icon between them represents the network. A dashed box on the left contains 'GUI' and an icon of three people. Below the main client and server, a large dashed box contains multiple pairs of processes: 'Animation process 1' and 'Algorithm process 1', 'Animation process 2' and 'Algorithm process 2', and 'Animation process n' and 'Algorithm process n'. Horizontal double-headed arrows connect each animation process to its corresponding algorithm process. A legend at the bottom shows a solid arrow for 'Communication' and a hollow arrow for 'Threads'.</p>

System	5. Java with a CSP library - JCSP (Nevison, 2001)	
Description	<p>JCSP takes advantage of Java as a language with facilities for concurrent threads and for communications among distributed processors and uses the constructs of CSP to manage the complexity of this concurrency. Two libraries, JCSP and CJT, have been developed and provide the functionality of CSP within Java. These libraries provide the tools needed to teach parallel and distributed computing with control constructs based on CSP. The use of Java with CSP makes one of the most complex aspects of distributed and parallel computing much easier to manage and, thereby, much easier to teach effectively.</p>	
Components	 <p>The diagram illustrates the components of the JCSP system. It features two parallel processes, each labeled 'P a d d l e', connected to a central control unit. This unit includes 'Key control' and 'Display control' components. A 'Canvas' component is connected to the 'Key control' and 'Display control' components. The 'Canvas' component is also connected to a 'Tutoring model' database, which contains a list of items: 'Java', 'CSP', and '.....'. At the bottom right of the diagram, there is an icon representing a group of people, indicating user interaction with the system.</p>	

<p style="writing-mode: vertical-rl; transform: rotate(180deg);">System</p>	<p>6. Lydian (Lydian, 2005)</p>
<p style="writing-mode: vertical-rl; transform: rotate(180deg);">Description</p>	<p>Lydian is a simulation and visualisation environment for distributed algorithms that provides to the students an experimental environment to test and visualise the behaviour of distributed algorithms. It gives to the students the ability to create easily their own experiments and visualise the behaviour of the algorithms on their experiments. Lydian is easy to use and can also be extended.</p>
<p style="writing-mode: vertical-rl; transform: rotate(180deg);">Components</p>	<p>Lydian offers a database of basic distributed algorithms and protocols from which the student can select. Students can insert a new distributed algorithm into Lydian by using a high-level description language. The simulator of Lydian takes, as an input, the network description file and the distributed algorithm and creates an execution (run). This execution describes the behaviour of the algorithm for the specific execution parameters. For each protocol in the database, Lydian provides a continuous animation.</p> 

3.5.4 Conclusion

The above systems were all developed to address the challenges discussed in section 3.5.2. The systems under investigation are based on the following models: *Computer animation*, which is the art of using computers to create moving images; *simulation*, whereby a computer program or network of computers attempts to imitate real things, a state of affairs, or processes, thus being an abstract model of a particular system; *visualisation*, which is the process of transforming information into a visual form, enabling the user to observe the information. This is accomplished by using techniques of computer graphics and imaging. *Java Applet / HTML* loads a Java class and runs it by a Java application that is already running, such as a Web browser. *HTML* (Hypertext Markup Language) is the standard mark-up language used for documents on the World Wide Web. The HTML language uses tags to indicate how Web browsers should display page elements such as text and graphics, and how Web browsers should respond to user actions, such as hyperlink activation, by means of a key press or mouse click.

The systems under investigation are summarised according to the above models in Table 3-6.

Table 3-6. Teaching Distributed Computing: Summary of Systems Evaluated

Animation	Simulation	Visualisation	Java applets/ HTML
HiSAP		HiSAP	
	DAJ		DAJ
	Java Toolkit		Java Toolkit
		VADE	
	JCSP		JCSP
	Lydian	Lydian	

Since distributed algorithms are difficult not only to grasp, but also to implement and debug, visualisation, simulation and animation were identified to aid in the demonstration of complex relationships and dynamic processes and, thus, have the potential to support teachers and learners. Also, when the Internet is used, use of Java applets and HTML help to ease the complexities, since explicit installation is not needed on the student side; only a Java-enabled browser, which is available on most computers, is needed.

To conclude, a generic model to teach distributed computing can be presented, as in Figure 3-3:

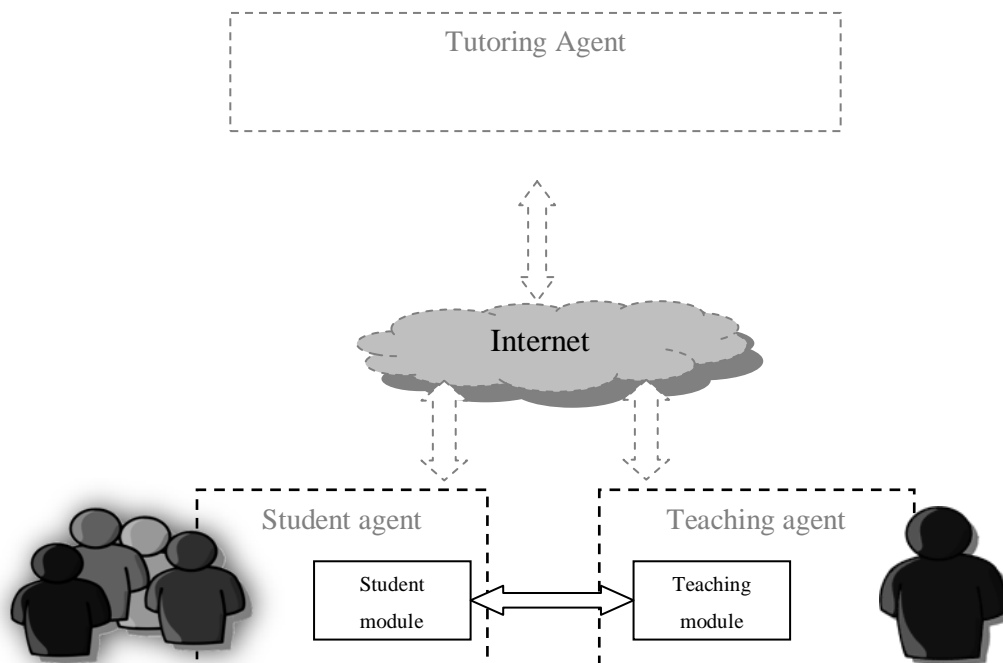


Figure 3-3. Teaching Distributed Computing

Most of the systems evaluated in section 3.4.3, use the model described at the beginning of this section, that is, a model consisting of two components, a teaching module and a student module, but the content is presented by visualisation, animation or simulation, or a combination thereof. Owing to the nature of distributed systems, however, some systems make use of the Internet to present the knowledge to be taught, mainly by using Java applets. A Java applet is a piece of Java code that is downloaded from a WWW server (the tutoring agent) and is executed on the client computer (the student agent).

3.6 What do we know about Teaching Distributed Systems in a Distance-based Educational Environment?

In order to answer the question, *What do we know about teaching distributed systems in a distance-based educational environment?*, the previous sections addressed the teaching of programming languages, the teaching of programming languages in a distance-based educational environment and the teaching of distributed systems. Before an attempt is made to identify systems in this category, the need to present distributed systems at an undergraduate level is defended.

3.6.1 **Background: COS3114**

COS3114 is a module offered by the University of South Africa as part of the undergraduate studies towards a Bachelor in Computer Science or Information Systems. It is a third-level advanced computer science elective module, named *Advanced Programming*. In order fully to comprehend the dynamic nature of this module, a short discussion of the changes in the computer science discipline will aid the motivation.

3.6.2 **Changes in the Computer Science Discipline**

The final report of the Computing Curricula 2001 project⁶ (CC2001) (2001) stated that computer science was an enormously vibrant field. Computers are integral to modern culture and are the primary engine behind much of the world's economic growth. The field is evolving at an astonishing pace as new technologies are introduced and existing ones become obsolete.

The rapid evolution of the discipline has a profound effect on computer science education, affecting both content and pedagogy. The CC2001 Task Force identified aspects of computer science that had changed over the past decade and categorised them as either technical or cultural. Each of these has a significant effect on computer science education. The major changes in each of these categories are described in the individual sections that follow.

3.6.2.1 *Technical Changes*

The CC2001 identified that the technical changes in computer science were both evolutionary (exponential increase in available computing power) and revolutionary (the rapid growth of networking after the appearance of the World Wide Web). These changes affect the body of knowledge required for computer science and the educational process. Thus, technical advances over the past decade have increased the importance of many curricular topics, such as the World Wide Web and its applications, networking technologies - particularly those based on TCP/IP, embedded systems, interoperability, object-oriented programming, the use of sophisticated application programmer interfaces (APIs) and security and cryptography

⁶ CC2001 is a joint undertaking of the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) and the Association for Computing Machinery (ACM) to develop curricular guidelines for undergraduate programmes in computing.

3.6.2.2 Cultural Changes

According to CC2110, computing education is also affected by changes in the cultural and sociological context in which it occurs. The following changes, for example, have all had an influence on the nature of the educational process: changes in pedagogy enabled by new technologies, the dramatic growth of computing throughout the world, the growing economic influence of computing technology, greater acceptance of computer science as an academic discipline and broadening of the discipline.

3.6.2.3 COS3114

The above discussion leads to the motivation for the module COS3114. According to CC2001, advanced courses serve three purposes. The first is to expose students to advanced material beyond the core. The second is to demonstrate applications of fundamental concepts presented in the core courses, and the last is to provide students with a depth of knowledge in at least one subarea of computer science.

It is through the use of this model that the purpose of COS3114 is to *increase the depth of students' insight into advanced programming principles and consolidate their competence therein*. Owing to the ever changing nature of the computer science field, the curriculum of this module is updated regularly (every four years). A decision was made to present distributed systems in COS3114, based on the recommendations made by the CC2001.

3.6.3 Conclusion

After the investigation into the systems in the previous three sections, the following conclusions have been reached regarding teaching of distributed systems in a distance-based educational environment. First, to our knowledge, systems could be identified to teach distributed systems only in a laboratory environment. No reference to teaching distributed systems in a distance-based educational environment could be found. Secondly, very few, if any, systems that take full advantage of the functionality available in teaching in a distance-based educational environment. These functionalities include, *inter alia*, electronic mail and discussion forums. Thirdly, the challenges identified in both teaching in a distance-based educational environment (3.4.2) and teaching distributed systems (3.5.2) together have to be addressed in a system that teaches distributed systems in a distance-based educational environment. Lastly, special reference and accommodation is to be made to the economic abilities of students, since commercially available software may be out of reach for an average student.

This gave rise to the research question:

How should distributed computing be taught in a distance-based educational environment?

The model proposed was used and tested to teach distributed systems in a distance-based educational environment. Chapter 5 is dedicated to a detailed discussion of the proposed model.

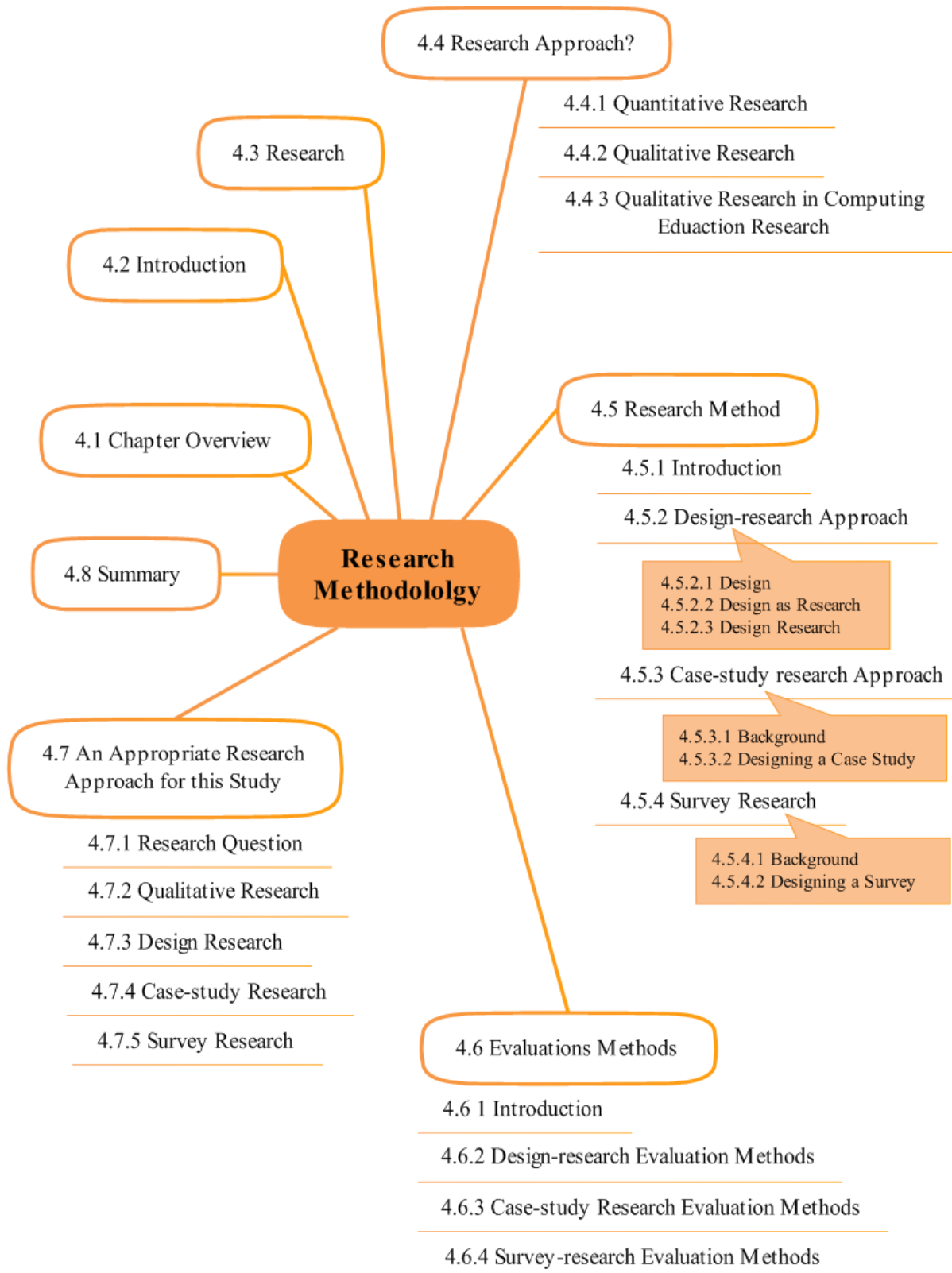
Since the research method that is followed is directly connected to the problem statement and goal of the research, Chapter 4 describes the research approach used in this study.

3.7 Summary

The chapter addressed the second subsidiary question, *What are the factors contributing to the success of teaching programming, and how can these factors be integrated into teaching distributed computing in a distance-based teaching model?* The objective was to explore the learning and teaching of programming languages. First, typical problems and solutions were discussed with regard to the learning and teaching of programming languages in environments where there is contact or laboratory sessions, as is the practice in residential universities. This was followed by a discussion of learning and teaching programming in a distance-based educational environment (DEE). Similarities to and differences from contact session were explored. The final examples that were explored were the teaching and learning of distributed computing in contact sessions and the challenges that this posed. The chapter concluded by exploring these challenges in a distance-based educational environment.

4. Research Methodology and Design

4.1 Chapter Overview



4.2 Introduction

The aim of this chapter is to describe the research approach used in this study. The first part of this chapter describes research in general, research approaches and methods and evaluation thereof. The second part of the chapter uses this information to describe and motivate the research design selected for this study.

4.3 Research

This section provides a short overview of what research is. The need for a formal approach, as well as the main research approaches, is then defined.

There are numerous competing definitions of the term research. A general definition for research, according to Kuhn (1996) and Lakatos (1987), is: *Research is an activity that contributes to the understanding of a phenomenon. A phenomenon is defined as a set of behaviours of some entity, or entities, found interesting by a researcher. The term understanding is defined as knowledge that allows prediction of the behaviour of some aspect of the phenomenon, and the set of activities to understand the phenomenon are the research methods or techniques.*

Another definition of the term “research” by Coryn (2006) is: *a truth-seeking activity that contributes to knowledge, aimed at describing or explaining the world, conducted and governed by those with a high level of proficiency or expertise.* Truth seeking is the search for or investigation into real things, events, or facts. Research describes: to describe involves representing or giving an account of a phenomenon. Research explains: to explain is to give the reason for or cause of a phenomenon. Research is conducted at a high level of proficiency or expertise, which results in a contribution to knowledge.

Olivier (2004) defines research as an *investigation to discover facts*. This definition does not require the researcher to discover facts that have not previously been discovered by someone else. Oates (2006) defines research *as the creation of new knowledge*, using an appropriate process, to the satisfaction of the users of the research and which is characterised by sufficient and appropriate data sources, which is accurately recorded and properly analysed; there are no hidden assumptions and the conclusions are well-founded and properly presented.

All the definitions of research above confirm that an approach is needed to produce research results. Therefore, it is imperative to follow a research approach.

4.4 Research Approach

Research-approach methods are classified in various ways. Distinctions made by Lakatos (1987) are between quantitative and qualitative research methods, objective versus subjective, discovery of general laws (*nomothetic*) versus the uniqueness of each particular situation (*idiographic*), prediction and control versus explanation and understanding, and the outsider (*etic*) versus the insider (*emic*) perspective. The most general distinction for a research approach, however, is made between quantitative and qualitative research methods.

4.4.1 Quantitative Research

Quantitative research methods were developed in the natural sciences to study natural phenomena. Quantitative research is the systematic scientific investigation of properties and phenomena and their relationships. The objective of quantitative research is to develop and employ mathematical models, theories, and/or hypotheses pertaining to natural phenomena. The main advantage of the quantitative approach is that it measures, for example, the evaluation of results, the modelling and analysis of data, the collection of empirical data and survey methods.

4.4.2 Qualitative Research

A detailed discussion on qualitative research can be found in Myers (1997). Qualitative research methods were developed in the social sciences to enable researchers to study social and cultural phenomena, situations in which people and different processes are involved. Qualitative research is the examination, analysis and interpretation of observations for discovering underlying meanings and patterns of relationships. Qualitative data sources include observation and participant observation (fieldwork), interviews and questionnaires, documents and texts, and the researcher's impressions and reactions.

All research, whether quantitative or qualitative, is based on some underlying assumptions about what constitutes 'valid' research and which research methods are appropriate. In order to understand qualitative research, it is important to know what these assumptions are. The most important assumptions are those that relate to the underlying epistemology, which guides the research (Myers, 1997). Epistemology refers to the theory of knowledge and how it can be obtained.

Guba and Lincoln (1994) suggest four underlying "paradigms" for qualitative research: positivism, post-positivism, critical-theory and constructivism. Orlikowski and Baroudi (1991), following Chua

(1986), suggested three categories, based on the underlying research epistemology: positivist, interpretive and critical.

4.4.3 Qualitative Research in Computing Education Research

Computing education research (CER) is a cross-disciplinary field, comprising computing and a wide range of other disciplines: pedagogy, psychology, cognitive science, learning technology and sociology (Berglund, 2006). As mentioned earlier, the qualitative research approach is typically used for the investigation of social phenomena, or, in other words, situations in which people are involved and different kinds of process take place. In CER, these processes involve the teaching and learning within computing. Therefore, qualitative research methodologies allow a researcher to obtain insights that are not accessible using research approaches commonly employed in the natural sciences. Examples of qualitative research methods are design research and case-study research.

4.5 Research Method

4.5.1 Introduction

The methods employed are *design research*, since an artefact is created, and a *case study*, since “how” and “why” questions need to be answered. Data collection was done through a *survey*. Each method was evaluated via its own well-established evaluation methods, since evaluation is a crucial component of the research process.

In section 4.5.2, design research is discussed; this is followed by a discussion of case-study research in section 4.5.3. Surveys are addressed in section 4.5.4, and evaluation methods for each method are discussed in section 4.6.

4.5.2 Design-research Approach

4.5.2.1 Design

To design, according to the Webster dictionary (Webster Dictionary and Theasaurus, 1992) is *to invent and bring into being*. Numerous activities are involved in designing an artefact. At an intellectual level, Simon (Vaishnavi, 2004/5) explains this design activity by making a clear distinction between “natural sciences” and “artificial science”, which is also known as design science and which is depicted in Figure 4-1.

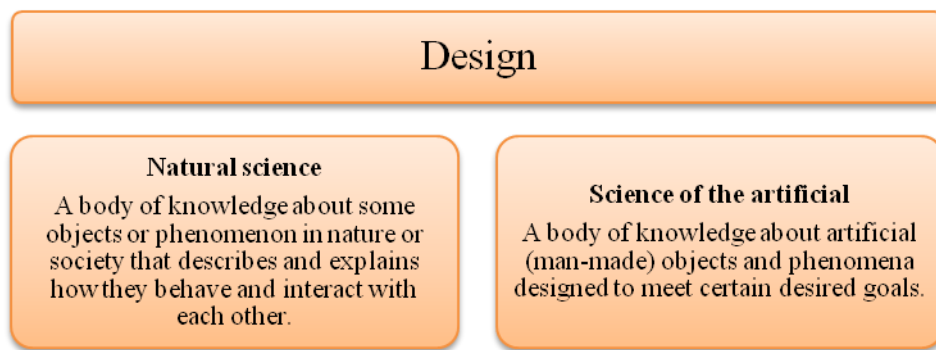


Figure 4-1. The Design Activity according to Simon (Vaishnavi, 2004/5)

Simon further refines the science of the artificial in an *inner environment*, an *outer environment* and the *interface* between the two that meets certain desired goals, as depicted in Figure 4-2.

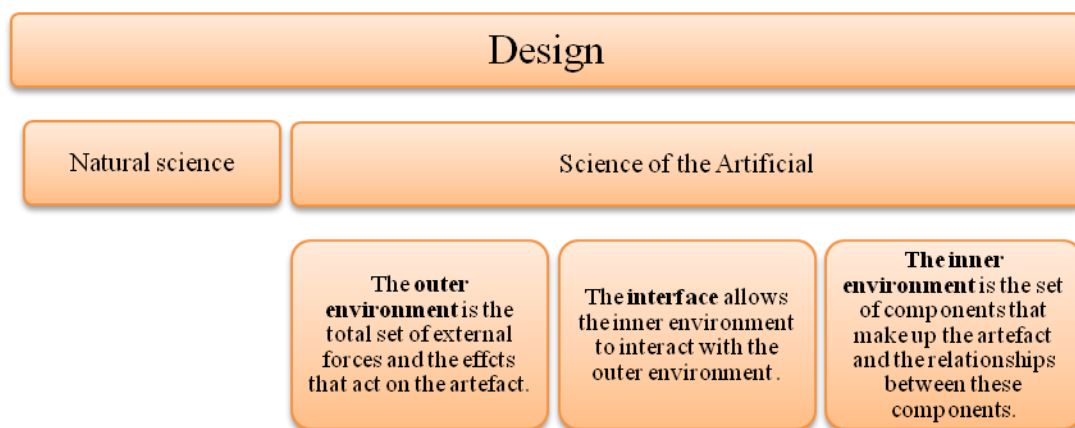


Figure 4-2. The Design Activity according to Simon (expanded)

The design activity is thus the knowledge needed in the form of techniques and methods to create an interface between the inner and the outer environment – the expertise for implementing an artefact that satisfies a set of functional requirements (Vaishnavi, 2004/5) (Owen, 1998). This formal process, dictated by the discipline under which it operates, is called design research.

4.5.2.2 Design as Research

As mentioned in the previous section, the design activity is the *knowledge* to create an interface between the inner and the outer environments. A model for generating and accumulating *knowledge*, as presented by Owen (1998), is helpful in understanding the design activity and is depicted Figure 4-3. This model is based on the concept of *doing something* and *judging the results*.

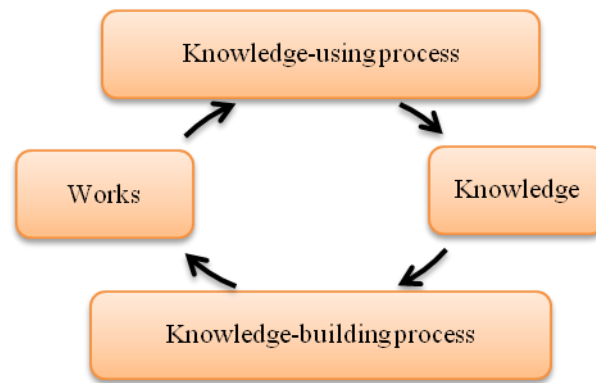


Figure 4-3. A General Model for Generating and Accumulating Knowledge (Owen, 1997)

Vaishnavi (2004/5) presents a general design cycle, which can be interpreted as an elaboration of the *knowledge-using process* and which is presented in Figure 4-4.

The reasoning or new knowledge generation that occurs in the general design cycle, as analysed by Takeda and presented by Vaishnavi (2004/5), is twofold: *circumscription* reasoning and *operational and goal knowledge* reasoning.

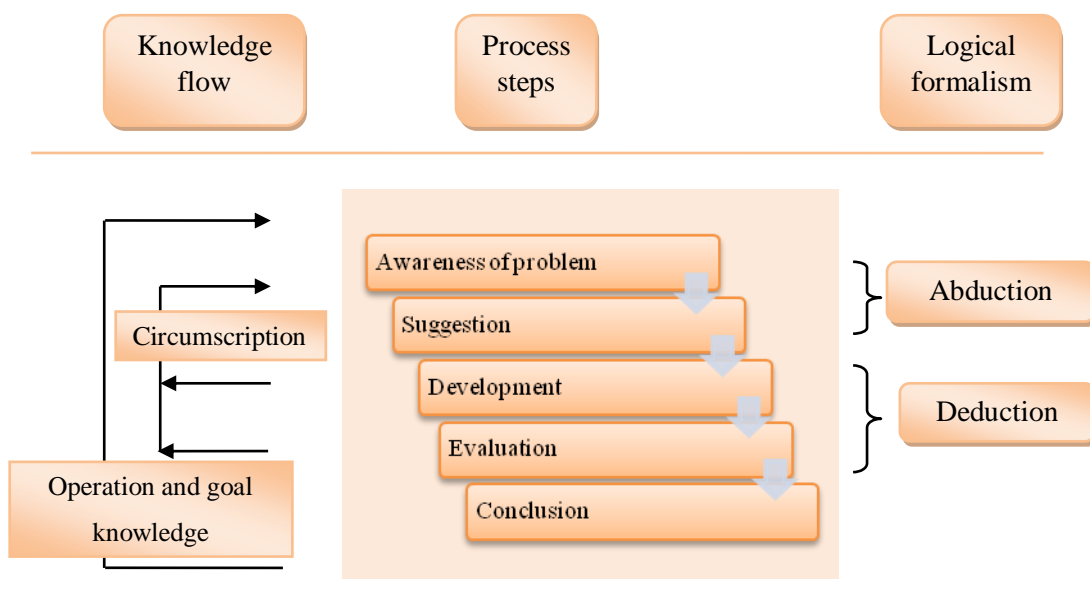


Figure 4-4. The Design Cycle

Circumscription reasoning is particularly important in order to understand design research, because circumscription reasoning generates an *understanding that can only be gained from the specific act of construction* (Vaishnavi, 2004/5). Circumscription, according to McCarthy (1980), is a formal logical

method, which assumes that every fragment of knowledge is valid only in certain situations. It is through detection and analysis of contradictions that one can determine the way in which this knowledge can be used. The reason for this process repeating itself is the incomplete nature of all knowledge bases. The process is depicted in Figure 4-5.

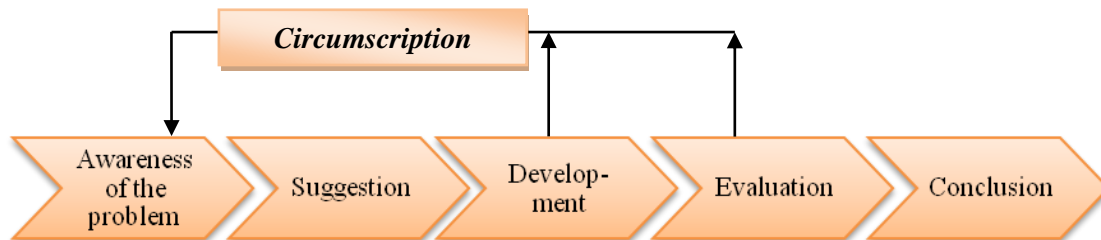


Figure 4-5. New Circumscription-knowledge Production

Operational and goal knowledge reasoning starts with *awareness of the problem*, working through the cycle and producing knowledge, which again serves as input to *awareness of the problem* of a next cycle. Operation and goal knowledge is knowledge gained, leading to the creation, manipulation and modification of the artefact - see Figure 4-6

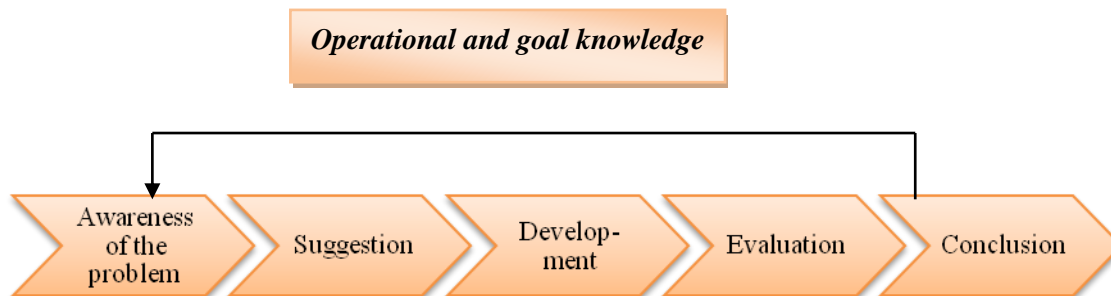


Figure 4-6. New Operational and Goal-knowledge Production

4.5.2.3 Design Research

Design research investigates the process of designing; main objective is to better understanding and to improve the design activity. Design research changes the state of the world through the introduction of novel artefacts. Henver (2004) proposed seven guidelines for effective design research, and these are presented in Table 4-1. These guidelines ensure that the artefact addresses an important and relevant problem, that the design is rigorously evaluated and that it is the best possible solution. The guidelines also ensure that the research contribution is verifiable and that the research relies on exact methods. The last guideline ensures effective communication of the research.

Table 4-1. Seven Guidelines for Effective Design Research

<p>Guideline 1: Design as an Artefact</p> <ul style="list-style-type: none">• A purposeful artefact has to be created to address an important problem.
<p>Guideline 2: Problem Relevance</p> <ul style="list-style-type: none">• The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
<p>Guideline 3: Design Evaluation</p> <ul style="list-style-type: none">• The utility, quality, and efficacy of a design artefact must be rigorously demonstrated via well-executed evaluation methods.
<p>Guideline 4: Research Contributions</p> <ul style="list-style-type: none">• Effective design-science research must provide clear and verifiable contributions in the areas of the design artefact, design foundations and/or design methodologies.
<p>Guideline 5: Research Rigor</p> <ul style="list-style-type: none">• Design-science research relies on the application of rigorous methods in both the construction and evaluation of the design artifact.
<p>Guideline 6: Design as a Search Process</p> <ul style="list-style-type: none">• The search for an effective artefact requires utilisation of available means to reach desired ends, while satisfying laws in the problem environment.
<p>Guideline 7: Communication of Research</p> <ul style="list-style-type: none">• Design science research must be presented effectively both to technology-oriented and management-oriented audiences.

March and Smith (1995) identify four design outputs or design artefacts to be produced by design research. The artefacts are summarised in **Table 4-2**. They are constructs, models, methods and instantiations.

Table 4-2. Four Design Outputs (Artefacts) to be produced by Design Research

Constructs
<ul style="list-style-type: none">•The language in which problems and solutions are defined and communicated; the conceptual vocabulary of a domain.
Models
<ul style="list-style-type: none">•Models use constructs to represent a real-world situation, the design problem and its solution space. Models aid problem and solution understanding and, frequently, represent the connection between problem and solution components, enabling exploration of the effects of design decisions and changes in the real world.
Methods
<ul style="list-style-type: none">•Methods defines processes. A set of steps is used to perform a task – how-to knowledge. Methods provide guidance on how to solve problems, that is how to search the solution space. These can range from formal, mathematical algorithms to informal, textual descriptions of "best practice" approaches, or some combination thereof.
Instantiations
<ul style="list-style-type: none">•The final output from a design research effort is an instantiation, which "operationalises constructs, models and methods." It is the realisation of the artefact in an environment.

Vaishnavi (2004/5) refer to a fifth output, *better theories*⁷. Design research can contribute to better theories in at least two distinct ways, both of which may be interpreted as analogous to experimental scientific investigation in the natural science sense.

Table 4-3. Vaishnavi's Addition to the Outputs of Design Research

Better Theories
<ul style="list-style-type: none">•Artefact construction as analogous to experimental natural science.

⁷Vaishnavi refers to the research of Rossi and Sein (2003) and Purao (2002), who set forth their own list of design research outputs. All but *better theories* can be mapped directly to March and Smith's list.

First, since the methodological construction of an artefact is an object of theorising for many communities (for example, how to build more maintainable software), the construction phase of a design research effort can be an experimental proof of method, or an experimental exploration of method, or both.

Second, the artefact can expose relationships between its elements. An artefact functions as it does because the relationships between its elements enable certain behaviours and constrain others. If the relationships between artefact (or system) elements, however, are less than fully understood and if the relationship is made more visible than previously during either the construction or evaluation phase of the artefact, then the understanding of the elements has been increased, potentially falsifying or elaborating on previously theorised relationships.

4.5.3 Case-study Research Approach

4.5.3.1 Background

There are several qualitative research strategies. These include case studies, experiments, histories, surveys and analysis of archival information. According to Yin (2003), it is a common misconception that the various research strategies should be arrayed hierarchically. A more appropriate view of these different strategies is an inclusive and pluralistic one. This leads to the identification of some situations in which all research strategies might be relevant and other situations in which two strategies might be considered to be equally attractive. Multiple strategies can also be used in any given study, for example, a case study within a survey, or a survey within a case study.

Yin (2003) also stated that each strategy can be used for the purpose of exploration, description or explanation, and the boundaries between the strategies are not always sharp. He proposed three (Becker, 2005) conditions that distinguish between the strategies. They are the type of research questions posed, the extent of control that an investigator has over actual behavioural events and, lastly, the degree of focus on contemporary, as opposed to historical, events.

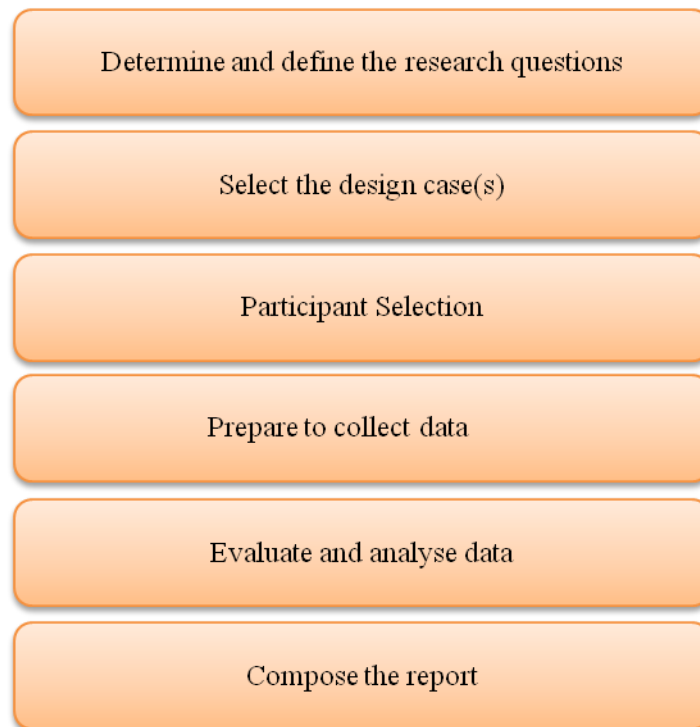
4.5.3.2 Designing a Case Study

Yin (2003) defines a case study as twofold. First, *a case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between the phenomenon and context are not clearly evident*. Secondly, *the case study inquiry copes with the technically distinctive situation in which there will be many more variables of interest than data points, as one result relies on multiple sources of evidence, with data needing to converge in a*

triangulating fashion, and as another result benefits from the prior development of theoretical propositions to guide data collection and analysis.

Components identified that guide the design of a case study are presented in the table below, followed by a discussion.

Table 4-4. Components of the Case-study Design Process



In order to determine and define the research question, the question is to be formulated in terms of *who*, *what*, *where*, *how*, and *why*, since such questions provide guidance in determining the strategy to follow. *How* and *why* questions indicate that the case study strategy is relevant.

Four basic designs for case studies exist and are depicted in Figure Figure 4-7. The general characteristics and potential use is stated in each box.

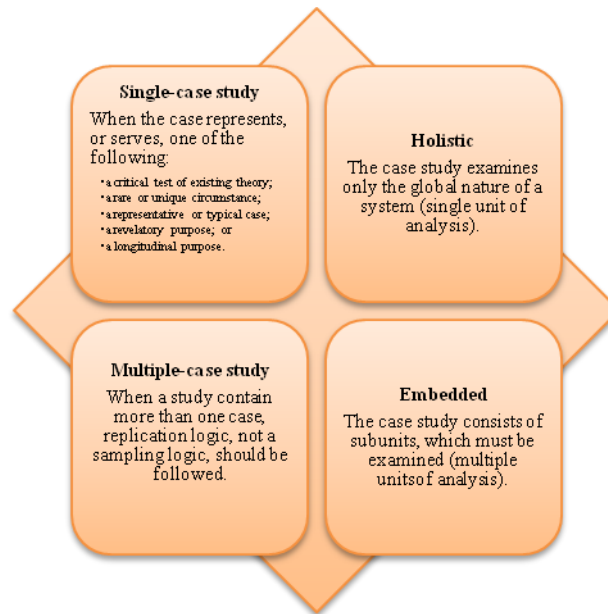


Figure 4-7. Basic Types of Design in Case Studies

The participant selection in case studies can use one participant, or a small group of participants. It is important that the participant pool remain relatively small.

Data can be drawn either from one primary source (for example, oral interviews, journals, or essays), or from multiple sources. As in ethnography, bringing together (*triangulating*) multiple perspectives, methods and sources of information (for example, from interviews, observations, field notes, self-reports, or think-aloud protocols, tests, transcripts and other documents) adds texture, depth, and multiple insights to an analysis and can enhance the validity or credibility of the results. Observations and data-collection settings may range from natural to artificial, with relatively unstructured to highly structured elicitation tasks and category systems, depending on the purpose of the study and the disciplinary traditions associated therewith.

As the information is collected, researchers strive to make sense of their data. Generally, researchers interpret their data in one of two ways: holistically or through coding. Holistic analysis does not attempt to break the evidence into parts, but rather to draw conclusions based on the text as a whole. Composition researchers commonly interpret their data by coding, that is, by systematically searching data to identify and/or categorise specific observable actions or characteristics. These observable actions then become the key variables in the study.

In the many forms it can take, "a case study is generically a story; it presents the concrete narrative detail of actual, or at least realistic events, it has a plot, exposition, characters, and sometimes even

dialogue" (Boehrer 1990). Generally, case-study reports are extensively descriptive, "the most problematic issue often referred to as being the determination of the right combination of description and analysis". Typically, authors address each step of the research process and attempt to give the reader as much context as possible for the decisions made in the research design and for the conclusions drawn.

4.5.4 Survey Research

4.5.4.1 Background

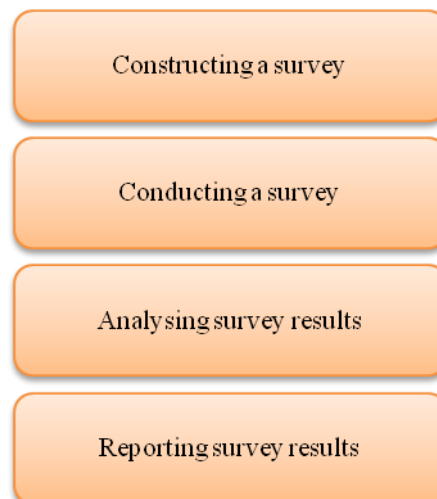
According to the (Webster Dictionary and Theasaurus, 1992) to survey is *to query (someone) in order to collect data for the analysis of some aspect of a group or area.*

Survey research is the method of gathering data from respondents thought to be representative of some population, using an instrument consisting of closed structure or open-ended items (questions) (Garson, 2008).

4.5.4.2 Designing a Survey

Components identified as guiding the design of a survey are summarised in Table 4-5.

Table 4-5. Components of the Survey-design Process



Selection of the type of survey is a critical and complex decision. Numerous issues have to be addressed in the selection process, and typical questions that can be asked to guide in the decision process are available. Surveys are roughly divided into two broad categories: questionnaires and interviews. Questionnaires are usually a paper-and-pen instrument, which the respondent completes,

whereas, in the case of interviews, the interviewer completes the survey based on what the respondent says.

Construction of a survey instrument is an art itself. Researchers should carefully consider and decide on numerous small aspects – about content, wording, format, placement – since these may have important consequences for the entire study. Three areas are involved in writing a question. The first is determining the question content, scope and purpose. This is followed by choosing the response format to use for collecting information from the respondent. The final area involved in writing a question for a survey is deciding how to word the question to get at the issue of interest. Another aspect to be considered is the issue of how to best place the questions and responses in the survey (Trochim, 2006).

Once the survey has been constructed, plans on how and to whom it must be administered have to be set up. A number of options are available in order to find a relevant sample group amongst the survey population. Furthermore, various considerations are involved in administering the survey itself.

Analysis of survey data is the process of transforming raw data into useable information, which is often presented in the form of an analytical report. The basic steps in the analytical process consist of identifying an issue, asking meaningful questions, developing answers to the questions through examination and interpretation of data and communicating the message to the reader. Analytical results can underscore the usefulness of data sources by shedding light on issues. Data analysis also plays an important role in the survey development and revision process.

When the results of a survey are reported, the minimum amount of data that communicates the overall findings effectively should be included. Clear distinction has to be made between ‘objective’ and ‘subjective’ reporting. It is important for the outcome of a survey to be able to distinguish between factual or numerical results and the researcher’s *interpretation* of the results. A sound way to report on the results of a survey is to have a section titled *results* and one titled *discussion*. The first of these is meant for plain, factual results, whereas the second is meant for interpretation and conjecture.

The next section addresses the evaluation methods of each of the research methods discussed. This is followed by a detailed discussion of how these methods are used in this study.

4.6 Evaluation Methods

4.6.1 Introduction

Evaluation is a crucial component of the research process (Hevner, 2004). Established evaluation procedures and processes are now discussed for each of the three approaches discussed in the previous sections.

4.6.2 Design-research Evaluation Methods

The utility, quality and efficacy of a design artefact must be demonstrated rigorously via well-executed evaluation methods. Evaluation of the artefact is established by the requirements of the artefact. The environment determines the requirements and includes the technical infrastructure. Thus, evaluation of an artefact includes integration of the artefact into the technical infrastructure of the environment.

Evaluation of a designed artefact requires definition of appropriate metrics and, possibly, the gathering and analysis of appropriate data. A design artefact is complete and effective when it satisfies the requirements and constraints of the problem that it was meant to solve Hevner (2004). Methodologies available to evaluate design artefacts given by Hevner (2004) are summarised in Table 4-6. These methodologies are observational, analytical, experimental, testing and descriptive.

Table 4-6. Design-research Evaluation Methods

Design-research Evaluation Methods	
1. Observational	<p>Case study: Study artefact in depth in environment.</p> <p>Field study: Monitor use of artefact in multiple projects.</p>
2. Analytical	<p>Static analysis: Examine structure of artefact for static qualities (for example, complexity).</p> <p>Architecture analysis: Study fit of artefact into technical IS architecture.</p> <p>Optimisation: Demonstrate inherent optimal properties of artefact, or provide optimality bounds on artefact behaviour.</p> <p>Dynamic analysis: Study artefact in use for dynamic qualities (for example, performance).</p>
3. Experimental	

Design-research Evaluation Methods	
	Controlled experiment: Study artefact in controlled environment for qualities (for example, usability). Simulation -- Execute artefact with artificial data.
4. Testing	
	Functional (black box) testing: Execute artefact interfaces to discover failures and identify defects. Structural (white box) testing: Perform coverage testing of some metric (for example, execution paths) in the artefact implementation.
5. Descriptive	
	Informed argument: Use information from the knowledge base (for example., relevant research) to build a convincing argument for the artefact's utility. Scenarios: Construct detailed scenarios around the artefact to demonstrate its utility

4.6.3 Case-study Research Evaluation Methods

The final deliverable of case-study research is the case-study report. A comprehensive report can serve as a checklist for the evaluation of a case study and should include the elements in the table below.

Table 4-7. Case-study Evaluation Methods

Case-study Evaluation Methods
Statement of the study's purpose and the theoretical context.
Problem or issue being addressed.
Central research questions.
Detailed description of the case(s) and explanation of decisions related to sampling and selection.
Context of the study and case history, where relevant.
Issues of access to the site/participants and the relationship between yourself, as researcher, and the research participant (case).
Duration of the study.
Evidence that you, as researcher, obtained informed consent that the participants' identities and privacy are protected, and, ideally, that participants benefited in some way from taking part in the study.
Methods of data collection and analysis, either manual or computer-based data management and analysis, or other equipment and procedures used.

Case-study Evaluation Methods

Findings, which may take the form of major emergent themes, developmental stages, or an in-depth discussion of each case in relation to the research questions; and illustrative quotations, or excerpts, and sufficient amounts of other data to establish the validity and credibility of the analysis and interpretations.

Discussion of factors that might have influenced the interpretation of data in undesired, unanticipated, or conflicting ways.

Consideration of the connection between the case study and larger theoretical and practical issues in the field.

4.6.4 Survey-research Evaluation Methods

The two aspects that need to be evaluated are the quality of the survey instrument and the quality of the survey data.

Designing the perfect survey questionnaire is impossible. Effective surveys can be created, however. An instrument to determine the effectiveness of a survey questionnaire is a pretest. Pretesting can help to identify the strengths and weaknesses of a survey as regards question format, wording and order. (Barribeau, 2005)

Barribeau (2005) identifies two types of survey pretest: *participating* and *undeclared*. Participating pretests dictate that the respondents are informed that the pretest is a practice run. Rather than the respondents being requested to simply fill in the questionnaire, participation in a pretest usually involves an interview setting, where respondents are asked to explain reactions to the question form, wording and order. This kind of pretest will help to determine whether the questionnaire is understandable. When an undeclared pretest is conducted, the respondents are not informed that it is a pretest. The survey is given as if for real. This type of pretest allows checking the choice of analysis and the standardisation of the survey.

Whether or not a participating or undeclared pretest is used, pretesting should ideally also test specifically for question variation, meaning and task difficulty, as well as respondent interest and attention.

The reliability and validity of the survey questions can also be pretested. To be reliable, a survey question must be answered by respondents in the same way each time. Researchers can assess reliability by comparing the answers that respondents give in one pretest with answers in another pretest. Then, a survey question's validity is determined by how well it measures the concept, or

concepts, that it is intended to measure. Both convergent validity and divergent validity can be determined, first, by comparing answers to another question measuring the same concept and, then, by measuring the answer to the participant's response to a question that asks for the exact opposite answer (Barribeau, 2005).

This concludes the evaluation procedures and processes of design research, case-study research and survey research. The next section addresses the appropriate research approach for this study.

4.7 An Appropriate Research Approach for this Study

4.7.1 Research Question

The research question for this study posed the question: *How should distributed computing be taught in a distance-based educational environment?* This question gave rise to four subsidiary questions, which relate to the research objectives, namely:

- **SQ1:** *What is a distributed system, and what are the aspects to be considered in designing a distributed system?*
- **SQ2:** *What are the factors contributing to the success of teaching programming, and how can these factors be integrated into teaching distributed computing in a distance-based teaching model?*
- **SQ3:** *What guidelines can the teacher involved in teaching distributed computing in a distance-based educational environment use to direct the teaching and learning process?*
- **SQ4:** *How can the proposed model for effective teaching and learning of distributed computing be implemented at a distance-based institution of higher education?*

To address the above questions, the following research strategies were followed:

4.7.2 Qualitative Research

The motivation for doing qualitative research, as opposed to quantitative research, comes from the observation that qualitative research methods are designed to help researchers to understand people and the social and cultural contexts within which they live.

4.7.3 Design Research

This study uses the seven guidelines proposed by Hevner (2004):

Guideline 1: Design as an artefact

The first identifiable deliverable produced in this research is an *artefact* in the form of an environment that can run on a stand-alone desktop computer to aid in the learning of distributed computing concepts. A second research deliverable is workflow management. The term workflow management refers to the complete process followed to guide a student either through a learning curve of a concept, or in producing a deliverable in the form of a system or assignment.

Guideline 2: Problem relevance

The teaching and learning of programming is challenging. The challenges increase in a distance-based-based educational environment. The teaching and learning of distributed programming is challenging in any environment. Development of the artefact will address this challenge.

Guideline 3: Design evaluation

The systems were used by third-year university students at a distance-based-based educational environment. Evaluation was done via, *inter alia*, questionnaires.

Guideline 4: Research contributions

The presentation of a distinctive approach when teaching distributed computing in a distance-based educational environment is relevant and significant, since it identifies the factors that facilitate a teaching and learning environment where it is possible to achieve quality of learning. The significance of the model is that it also addresses a gap in the literature.

Guideline 5: Research rigour

Construction of the design artefact is based on software engineering concepts in design, and evaluation of the design artefact incorporates evaluation of the case study.

Guideline 6: Design as a search process

Testing is done in the specific application domain.

Guideline 7: Communication of research

The target audiences are researchers, developers and practitioners who are studying and implementing distributed systems.

4.7.4 Case-study Research

A case study is used to identify whether the proposed model satisfies the requirements and constraints to teach distributed computing effectively in a distance-based educational environment.

As stated in the discussion on case studies, a *how* question indicates that the case study strategy is relevant. The *how* question for this study is formulated as follows: *How* should distributed computing be taught in a distance-based educational environment?

The study focuses on the embedded single-case design. It is a single-case design, a model to teach distributed computing in a distance-based-based educational environment, but there are other units of analysis, for example, the student and the technology (Yin, 2003).

The participants in this case study were third-year university students at a distance-based-based educational environment. The data were collected via questionnaires, which were delivered to them via email, being a facility offered by the University's Administrative Department.

4.7.5 Survey Research

The survey instrument used in this study was a short questionnaire based upon the *independent distributed learning model*. The following questions were asked:

- Do you think that our teaching methods of *Distributed Computing in a Distance-based Educational* environment were effective?
- How did you experience the following tools? Did you find it useful?
 - the editor: DevC++
 - make facility
 - minGW
 - tutorial letters
 - module web site
 - module forum
 - module email
- What methods or tools did **NOT** work well for you?
- Which aspects of distributed computing do you think need more attention?
- How is the content applicable in your working environment?

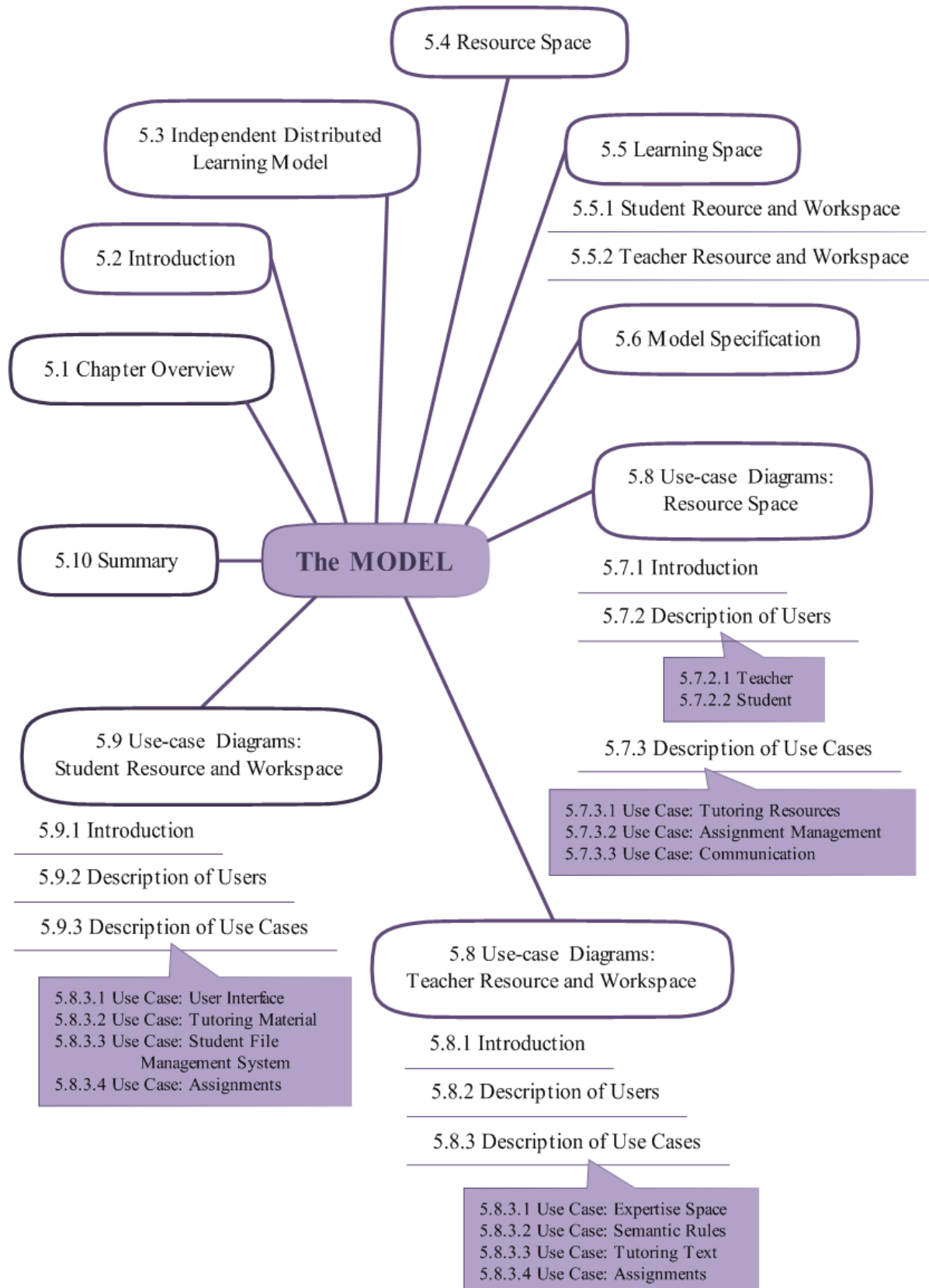
The goal of these questions was to determine the student's experience using the *independent distributed learning model*.

4.8 Summary

The chapter begins with a discussion on research in general, concluding in that a research approach is necessary. The most general distinction for a research approach is made between quantitative and qualitative research methods. Quantitative research methods were developed in the natural sciences to study natural phenomena. Qualitative research methods were developed in the social sciences to enable researchers to study social and cultural phenomena; situations in which people and different processes are involved. This study employed qualitative research, as opposed to quantitative research, as qualitative research methods are designed to help researchers to understand people and the social and cultural contexts within which they live. The research methods employed are *design research*, since an artefact is created, and a *case study*, since “how” and “why” questions need to be answered. Data collection was done through a *survey*. Each method was evaluated via its own well-established evaluation methods, since evaluation is a crucial component of the research process.

5. The Model

5.1 Chapter Overview



5.2 Introduction

The third subsidiary question, *What guidelines can the teacher involved in teaching distributed computing in a distance-based educational environment use to direct the teaching and learning process?*, is addressed in chapter. To answer this question, a model was proposed based on the guidelines identified by the first and second subsidiary question.

A model abstracts the essential details of the underlying problem from its usually complicated real world. As mentioned in section 3.6, a model was developed to teach distributed systems in a distance-based educational environment. The model is based on the client-server software architecture model. The term *client-server* refers to the popular model for computer networking that utilises client and server devices, each designed for specific purposes. Standard networked functions, such as email exchange, web access and database access, are based on the client-server model. The remainder of this chapter discusses this model, named the *independent distributed learning* model. This study is focused specifically on the student in a distance-based educational environment. It is beyond the scope of this dissertation to address learning models.

5.3 Independent Distributed Learning Model

An overview of the *independent distributed learning model* is presented in Figure 5-1.

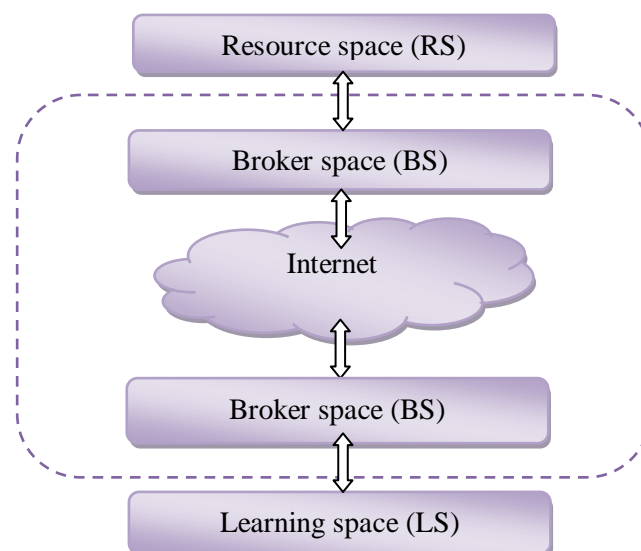


Figure 5-1. Distributed Learning Model

In the model, the **resource space (RS)** acts as a server and facilitates an environment in which all the resources and functionality needed to accommodate the learning experience reside. These resources are made available to the learning space, in which the student and the teacher reside, through the broker space.

The **broker space (BS)** acts as a *middleman*, which pairs requests from the learning space with the resource space. The learning space makes its functionality and needs known to the broker space. The main responsibility of the broker space is to identify and match these requests to the resource space. Thus, the broker space is responsible for communication between the resource space and the learning space.

The **learning space (LS)** facilitates an environment in which the teacher and the student can execute their respective tasks.

Since the focus of this study is the teaching of distributed computing in a distance-based educational environment, we acknowledge the role of the broker space, but shall not discuss the functionality and workings of the broker space further.

5.4 Resource Space

Any e-learning environment requires support infrastructure, such as remote servers, databases and software systems, to create the learning space in which teaching and learning can take place. This environment might be synchronous, necessitating continuous remote resource support, whereby the learner and teacher interact in the same time frame with their respective learning or teaching environments and with each other. The environment might also be an asynchronous learning environment in which remote resources are responsible for initial set-up of the learning environment without continuous monitoring and support of this environment. In an asynchronous environment, students work offline for most of their learning session, but might at any time choose to reconnect to the resource to interact with the resources for tasks such as queries, assignment submission and so forth.

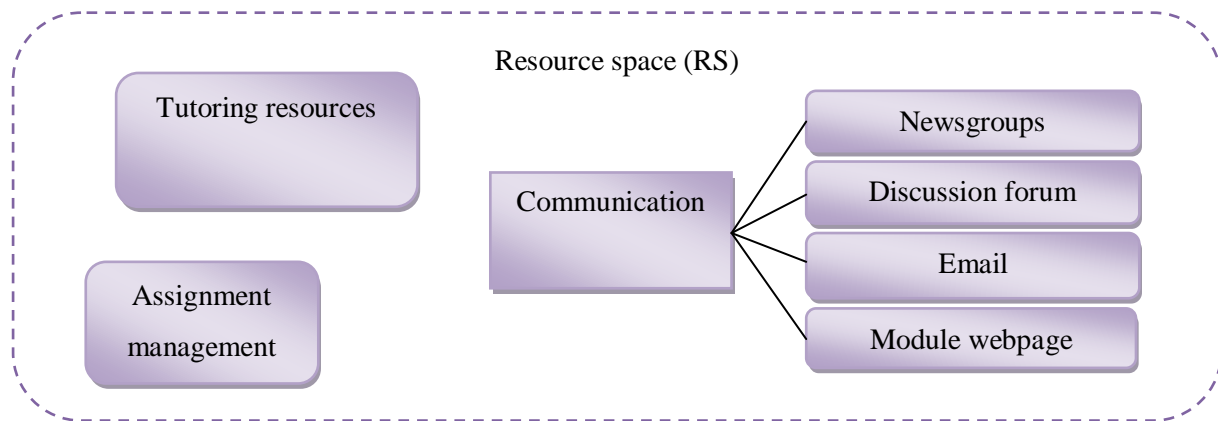


Figure 5-2. Resource Space

The significance of the resource space is that it combines the different resource into an area in which technical support is maintained. Although the infrastructure and the systems within this space might be heterogeneous and distributed as regards specific location, this space consists of the different resource components. Preservation and, consequently, maintenance of a well-defined and specialised space is less complex than when these resources are conceptually scattered throughout every space. The components of the resource space are depicted in Figure 5-2; these components are tutoring resources, assignment management and communication, which includes the module webpage, newsgroups, email and a discussion forum.

5.5 Learning Space

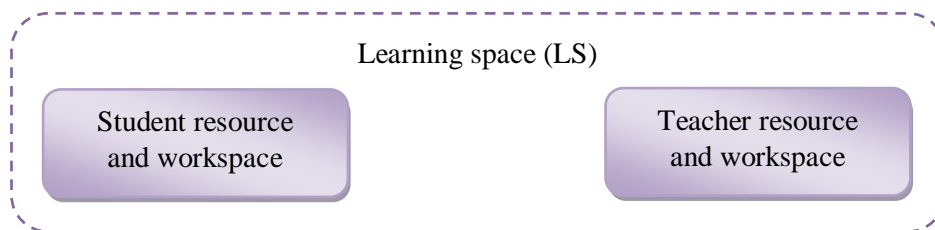


Figure 5-3. Learning Space

The **learning space** consists of both the *student resource and workspace* and the *teacher resource and workspace* and can be viewed as a facility that allows the student and teacher to fulfil their tasks. The significance of the learning space is that it localises the workspace and resources of both the teacher and the student.

5.5.1 Student Resource and Workspace

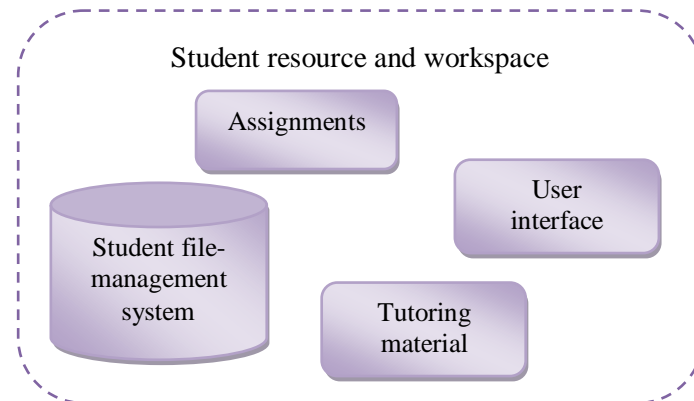


Figure 5-4. Student Resource and Workspace

The *student resource and workspace* facilitate an environment in which the student can communicate with the resource space, via a user interface, through the Internet. The components of the student resource and workspace are a user interface, tutoring material and a student file-management system.

5.5.2 Teacher Resource and Workspace

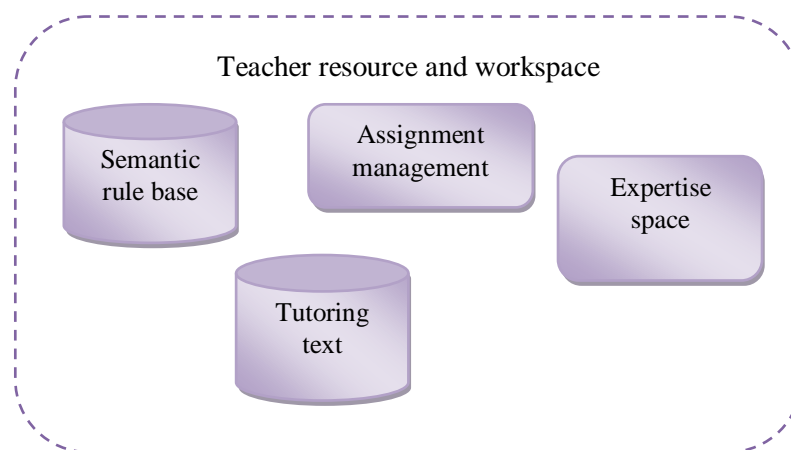


Figure 5-5. Teacher Resource and Workspace

The *teacher resource and workspace* facilitate an environment in which the teacher can communicate with the resource space through the Internet. The components of the teacher resource and workspace are an expertise space, semantic rule base, tutoring text and assignment management.

5.6 Model Specification

The Unified Modeling Language was proposed by Grady Booch, Ivar Jacobson, and James Rumbaugh as a standard notation for object-oriented analysis and design (Booch, 1999). UML provides various description techniques, for example, use-case diagrams, collaboration diagrams and class-state diagrams. The basic strategy that will be followed is to present the model as two high-level models. In this way, novice and experienced readers can start with common, easily understandable base techniques and proceed to more detailed descriptions only if required. The two description techniques used in this study are *use-case diagrams*, which model the users and their interactions with the system at a very high level of abstraction, and *activity diagrams*, which are used to describe the workflow behaviour of a system.

The use-case model shows the *users* and *uses* of the system. Use-case models are usually easy to understand, since they do not have any complex syntax, and are concerned with tasks and processes with which the intended users are familiar from their everyday work. Use cases are accompanied by a scenario. A scenario is a brief narrative, or story, which describes the hypothetical use of a system. In one or more paragraphs, a scenario tells who is using the system. Furthermore, a use case elaborates on what the system is trying to accomplish and provides a realistic, fictional account of users' constraints – when and where they are working, why they are using the system, and what they need the system to do for them. Use cases also describe any relevant aspects of the context in which the user is working with the system, including which information the user has on hand when beginning to use the system. A use case gives the user a fictional name, but it also identifies the user's role, such as student, or teacher, and indicates what the user regards as a successful outcome of using the system.

Activity diagrams make it easy to do a step-by-step simulation of a system's dynamics and, thus, allow a reader gradually to derive a global understanding from local insights. As stated in the previous section, a use case *encompasses an interaction between a user and a system*. An activity diagram captures the *behaviour* of a single case by showing the collaboration of the objects in the system to accomplish the task. Activity diagrams are used in the next chapter as a description technique in the application of the model in a case study.

5.7 Use-case Diagrams: Resource Space

5.7.1 Introduction

The use-case model shows the users and uses of the system. The use cases of the resource space, as identified in Figure 5-2, are the *tutoring resources*, *assignment management* and *communication*, respectively. The users are the *teachers* and *students*.

The first use case is that of the *tutoring resources*, which contain resources, software and procedures to accommodate the learning of distributed systems. The second basic use case is *assignment management*, which contains functionality for downloading, uploading and management of assignments. The last use case is responsible for the *communication* in the model and entails the module webpage, email, and discussion forum and newsgroups facilities.

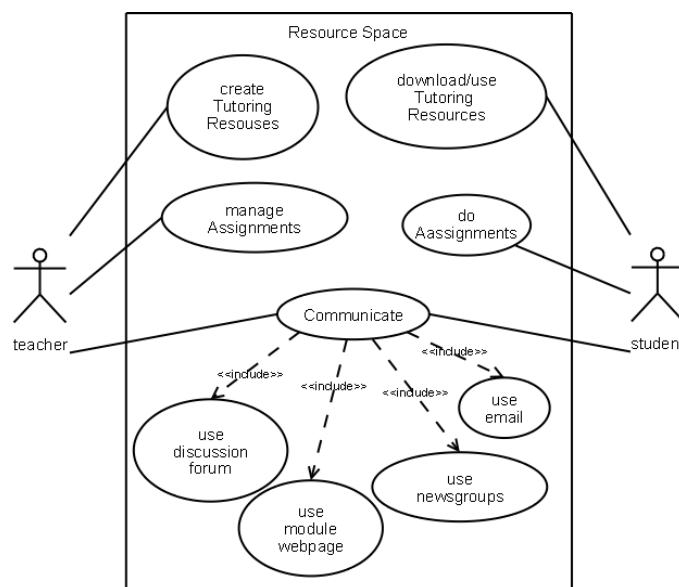


Figure 5-6. Use-case Diagram - Resource Space

5.7.2 Description of Users

5.7.2.1 Teacher

The teacher is responsible for maintaining the tutoring resources. These resources comprise software, references and tutorial letters. The software includes several CORBA developing platforms, as well as several IDEs. References include both books and Internet URLs. Besides containing academic content, tutorial letters also contain procedures and processes. Procedures refer to the installation of the

software systems, getting started with the systems and using the resources available. The processes refer to the various aspects of studying. The teacher also manages the assignments by uploading student assignments. The teacher uses the newsgroup to send electronic messages to all students via the module webpage. These messages include, *inter alia*, announcements, new references of information resources and changes in submission dates. Email is an asynchronous way of communication and the preferred way. The teacher views the discussion forum from time to time and intervenes if necessary, but the discussion forum is mainly a sounding board for students.

5.7.2.2 Student

The student is responsible for obtaining the tutoring resources either via the Internet or the CD-Rom provided upon registration. All software and tutorial letters, as well as all references, are available. The students download their assignments and make use of assignment management. It is the students' responsibility to read the module newsgroup at regular intervals. Since email is the preferred way of communication, the student uses the module email address to communicate with the teacher. The discussion forum is an active way of communication between students.

5.7.3 Description of Use Cases

Tutoring resources provide the resources, tutorial letters and software to accommodate the learning of distributed systems.

Assignment Management provides the functionality for downloading, uploading and management of assignments.

Communication provides the ways of communication between teacher and student, as well as between student and student, and includes:

- *Module webpage*: informative facilities, for example, the prescribed works and teachers' availability.
- *Email*: facilitates asynchronous communication between student and teacher, as well as between student and student.
- *Discussion forum*: facilitates synchronous and asynchronous discussions between students, with teacher intervention if needed.
- *Newsgroups*: facilitates communication, since the newsgroup allows the teacher to broadcast messages to students.

5.7.3.1 Use Case: Tutoring Resources

As mentioned in the description of users, the tutoring resources comprise software, tutorial letters and references. The software includes several CORBA developing platform,s as well as several IDEs.

References include both publications and Internet URLs. The tutorial letters contain all information needed to study the module.

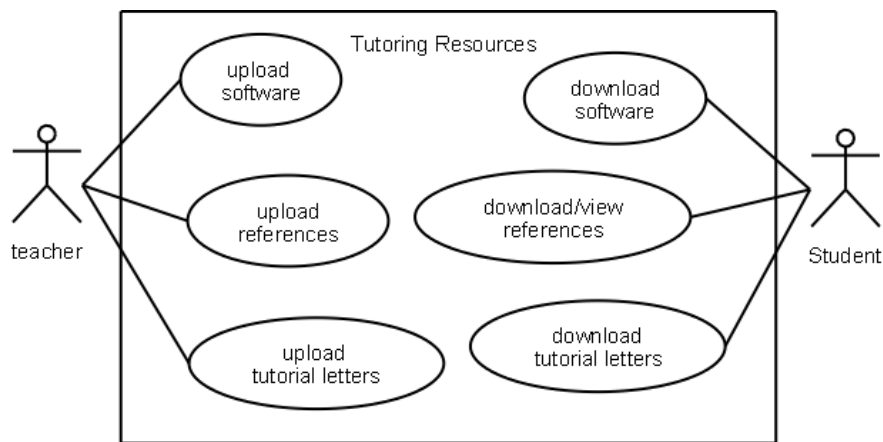


Figure 5-7. Use-case Diagram - Resource Space: Tutoring Resources

Examples include assignments, solutions, procedures to install the software systems, getting started and processes that refer to the different aspects of studying. The references include additional and supportive material for the module. Examples are the OMG CORBA specification, the OMG C++ mapping specification and the CORBA NameService specification.

5.7.3.2 Use Case: Assignment Management

Assignment management involves the downloading and uploading of assignments. Students download their assignments to enable the teacher to assess the assignments electronically.

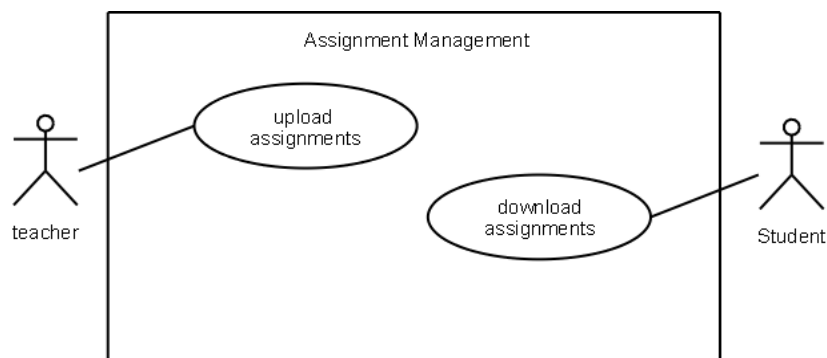


Figure 5-8. Use-case Diagram - Resource Space: Assignment Management

5.7.3.3 Use Case: Communication

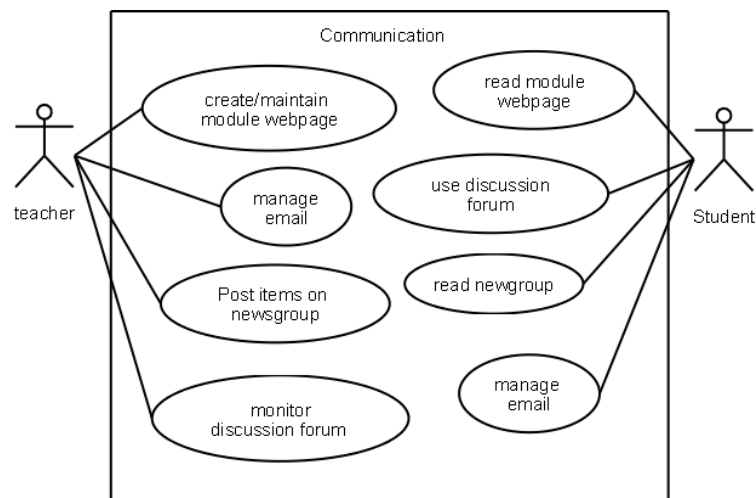


Figure 5-9. Use-case Diagram - Resource Space: Communication

5.7.3.3.1 Module Webpage

The module webpage provides the student with online information on the module. Examples are the prescribed works for the module, availability of the teachers, contact information of the teachers and other interested parties and links to email, discussion forums and newsgroups.

5.7.3.3.2 Email

Email is an asynchronous way of communication and the preferred way of communication. The student uses the module email address to communicate with the teacher. A student can also request email addresses from students according to criteria, for example, demographics.

5.7.3.3.3 Discussion Forum

The teacher views the discussion forum from time to time and intervenes if necessary, but the discussion forum is mainly a sounding board for students. The discussion forum is an active means of communication between students.

5.7.3.3.4 Newsgroups

The teacher uses the newsgroup to send out electronic messages to all students via the module webpage. These messages include, *inter alia*, announcements, new references of information resources and changes in submission dates. It is the students' responsibility to read the module newsgroup at regular intervals.

5.8 Use-case Diagrams: Teacher Resource and Workspace

5.8.1 Introduction

The teacher resource and workspace facilitate a local environment, which enables the teacher to accomplish the goal of teaching distributed computing. The components of the teacher resource and workspace are an expertise model, semantic ruler, tutoring text and assignments.

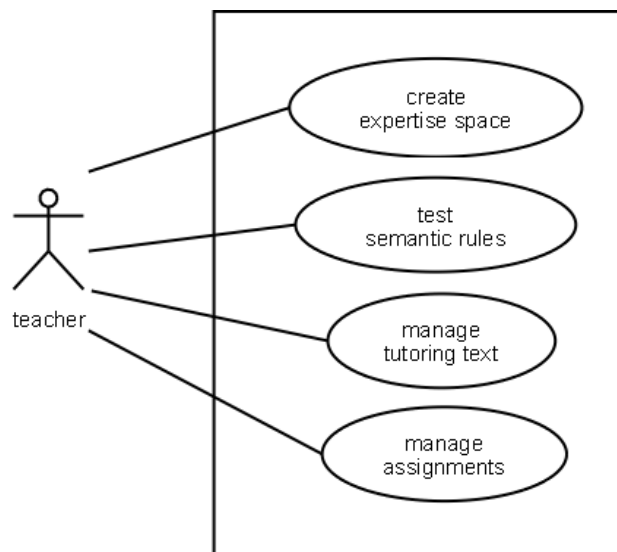


Figure 5-10. Use-case Diagram - Teacher Resource and Workspace

5.8.2 Description of Users

The teacher is responsible for the creating of all tutoring text. An expertise space is created where different software platforms are tested against one another to create the working environment for the students and scenarios are developed and tested against one another. The teacher uses semantic rules to ensure the correctness of the programming languages taught. All tutoring text is created and maintained here and includes inter alia, tutorials, references, etc. The teacher creates assignments, solutions and does assignment assessment.

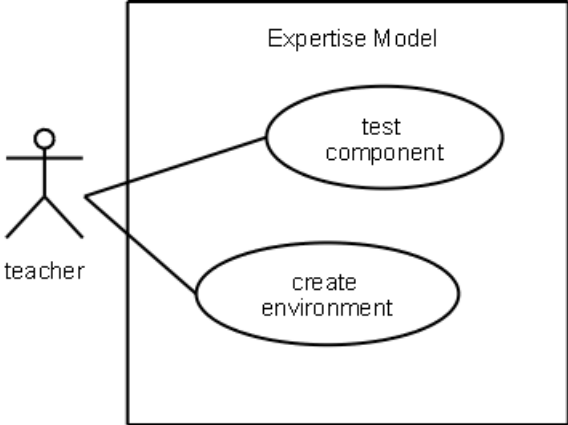


Figure 5-11. Use-case Diagram - Teacher Resource and Workspace: Expertise Space

5.8.3 Description of Use Cases

5.8.3.1 Use Case: Expertise Space

As mentioned in the previous section, an expertise space is created and, there, different software platforms are tested against one another to create the working environment for students. Different ORBs are tested with different compilers. Also, several IDEs are used with the different ORB/compiler combinations. An attempt is made to optimise the selection of the combination ORB/compiler/IDE. (See Appendix A)

5.8.3.2 Use case: Semantic Rules

The language that is taught has to be semantically and syntactically correct. Here the correctness is ensured and tested.

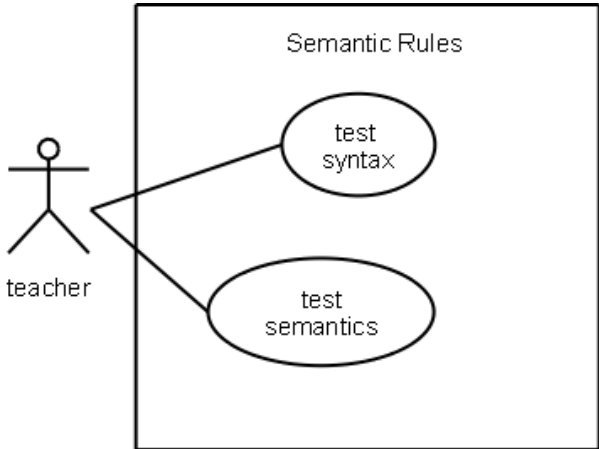


Figure 5-12. Use-case Diagram - Teacher Resource and Workspace: Semantic Rules

5.8.3.3 Use Case: Tutoring Text

All tutoring text is created here. Tutoring text include, *inter alia*, tutorial letters. Tutorial letters are the primary way of delivering information to students. All tutorial letters are dispatched to students via land mail. Electronic copies are also available for downloading from the module webpage at the moment that the dispatch process starts. It is for this reason that the electronic version is preferred. Several tutorial letters are created and contain all information needed to attempt the module successfully, as well as all assignments, solutions and examination information. Procedures and processes are also explained in tutorial letters. Tutoring text also includes all resources that are investigated to aid students in the learning process.

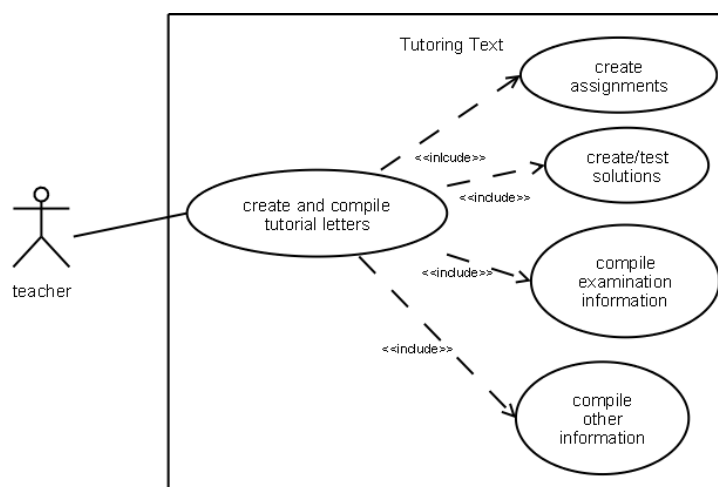


Figure 5-13. Use-case Diagram - Teacher Resource and Workspace: Tutoring Text

5.8.3.4 Use Case: Assignments

As mentioned earlier, assignments are the primary form of assessment during a course of study. The student has to submit an assignment, which is assessed and, then, returned to the student. Assessment is done both traditionally (by marking) and electronically. When a student submits an assignment either by mail or electronically, the teacher receives a printed copy, which can then be evaluated. When, however, a project (in the form of an assignment) is submitted, the teacher receives an electronic version of the code that the student has developed, and this version can be electronically assessed.

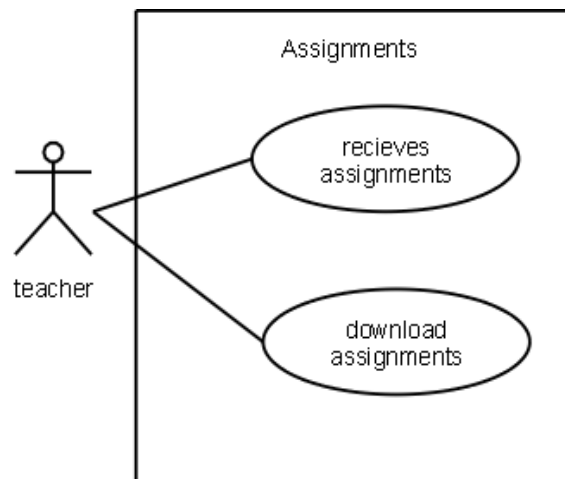


Figure 5-14. Use-case Diagram - Teacher Resource and Workspace: Assignments

5.9 Use-case Diagrams: Student Resource and Workspace

5.9.1 Introduction

The student resource and workspace facilitate is a local user environment, which enables a student to accomplish the goal of learning distributed computing.

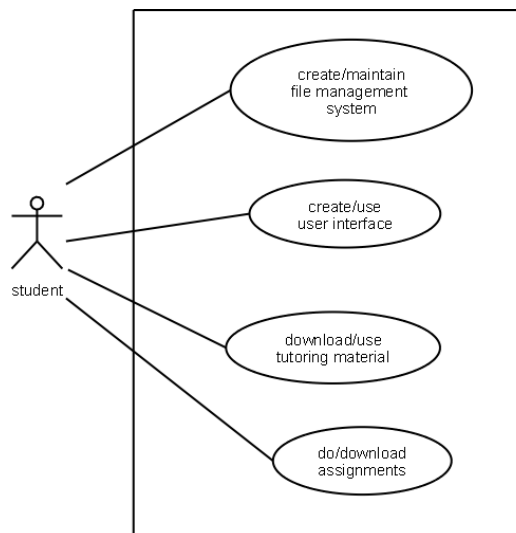


Figure 5-15. Use-case Diagram - Student Resource and Workspace

5.9.2 Description of Users

The student uses this space to develop, test and practice all required assignments for the successful completion of this module. It is the responsibility of the student to access, organise and use all available tutoring material. The student has to create the user interface, which consists of, *inter alia*, an IDL with an ORB and compiler, browser with email facilities and a DOS prompt. The student also has to create and maintain a file-management system for all tutoring material and materials created during the duration of the module.

5.9.3 Description of Use Cases

The four use cases identified in the student resource and workspace are:

- The *user interface*, which accomplishes the communication between student and environment and which is the environment in which the student can create and implement assignments.
- The *tutoring material*, which consists of tutorial letters and a library of tutoring material to be referenced during the studies.
- A *file-management system*, which is a system that the operating system or programs use to organise and keep track of files. For example, a *hierarchical file system* is one that uses directories to organise files into a tree structure.
- *Assignments*. The study of distributed systems is guided by the help of assignments. Before an important assignment is attempted, a tutorial is provided and should be worked through. An assignment is then attempted and submitted.

5.9.3.1 Use Case: User Interface

The *user interface* refers to all the various components that are needed to simulate a distributed computing environment for a student in order effectively to write, compile and execute programming assignments. The user interface is a functionality that students install themselves. Installation of the user interface, however, is a once-off procedure. The five components that follow are the minimum requirements of the user interface.

An *IDE* (*Integrated Development Environment*) of choice. An IDE is a set of programs that run from a single user interface, and has to include a text editor, compiler and debugger, which are all activated and function from a common menu.

An *ORB* (*Object Request Broker*). An ORB is streaming software that enables users to access other software remotely.

A *compiler*. This is a special program, which processes statements written in a particular programming language and turns them into the machine language or "code" that a computer's processor uses.

A *browser*. A browser is an application program that provides a means to look at and interact with all the information on the World Wide Web. A Web browser is a client program that uses HTTP (Hypertext Transfer Protocol) to make requests to Web servers throughout the Internet, on behalf of the browser user.

The *DOS prompt*. It either can be an icon on the desktop, or can be opened when needed.

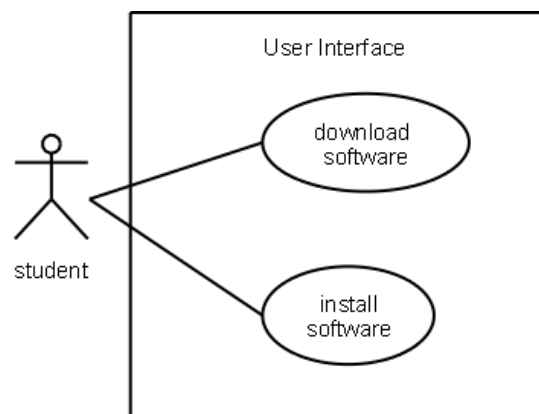


Figure 5-16. Use-case Diagram - Student Resource and Workspace: User Interface

5.9.3.2 Use Case: Tutoring Material

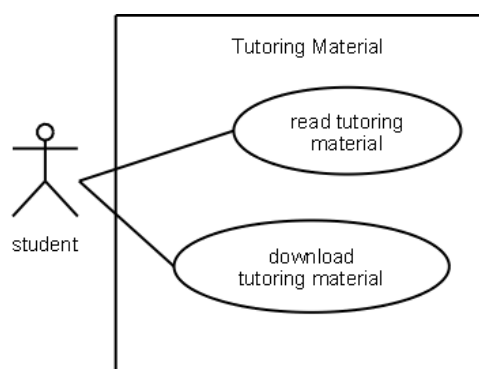


Figure 5-17. Use-case Diagram - Student Resource and Workspace: Tutoring Material

The *tutoring material* consists of tutorial letters and reference works. The tutorial letters contain all information pertaining to the module. The reference works are articles, publications and URL

addresses and include, *inter alia*, references to the *CORBA NameService specification*, the *OMG C++ mapping specification* and the *OMG CORBA specification*.

5.9.3.3 Use Case: Student File Management Systems

A *file-management system* is a system that the operating system or programs uses to organise and keep track of files. For example, a *hierarchical file system* is one that uses directories to organise files into a tree structure.

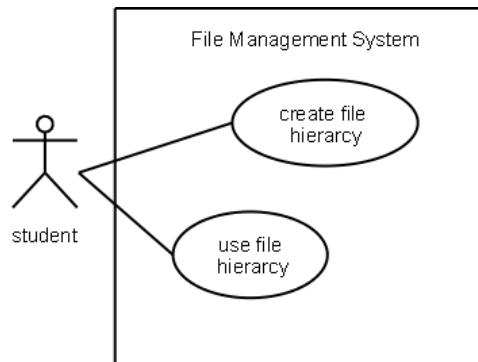


Figure 5-18. Use-case Diagram - Student Resource and Workspace: File-management System

5.9.3.4 Use Case: Assignments

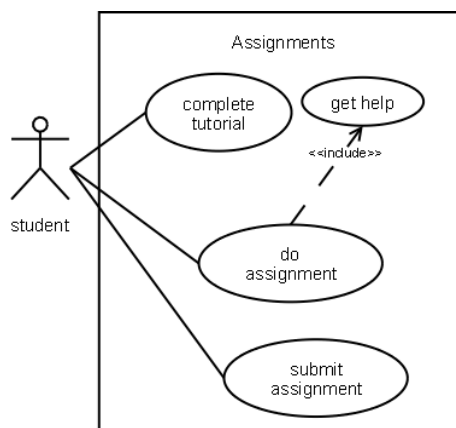


Figure 5-19. Use-case Diagram - Student Resource and Workspace: Assignments

The assessment approach used in this model is based on assignments. Before an important assignment is attempted, a tutorial is provided and should be worked through. An assignment is then attempted and submitted.

The assignment can be submitted either by mail or electronically. Both, however, will result in a printed copy being delivered to the teacher. A final and last assignment, a project, has to be uploaded

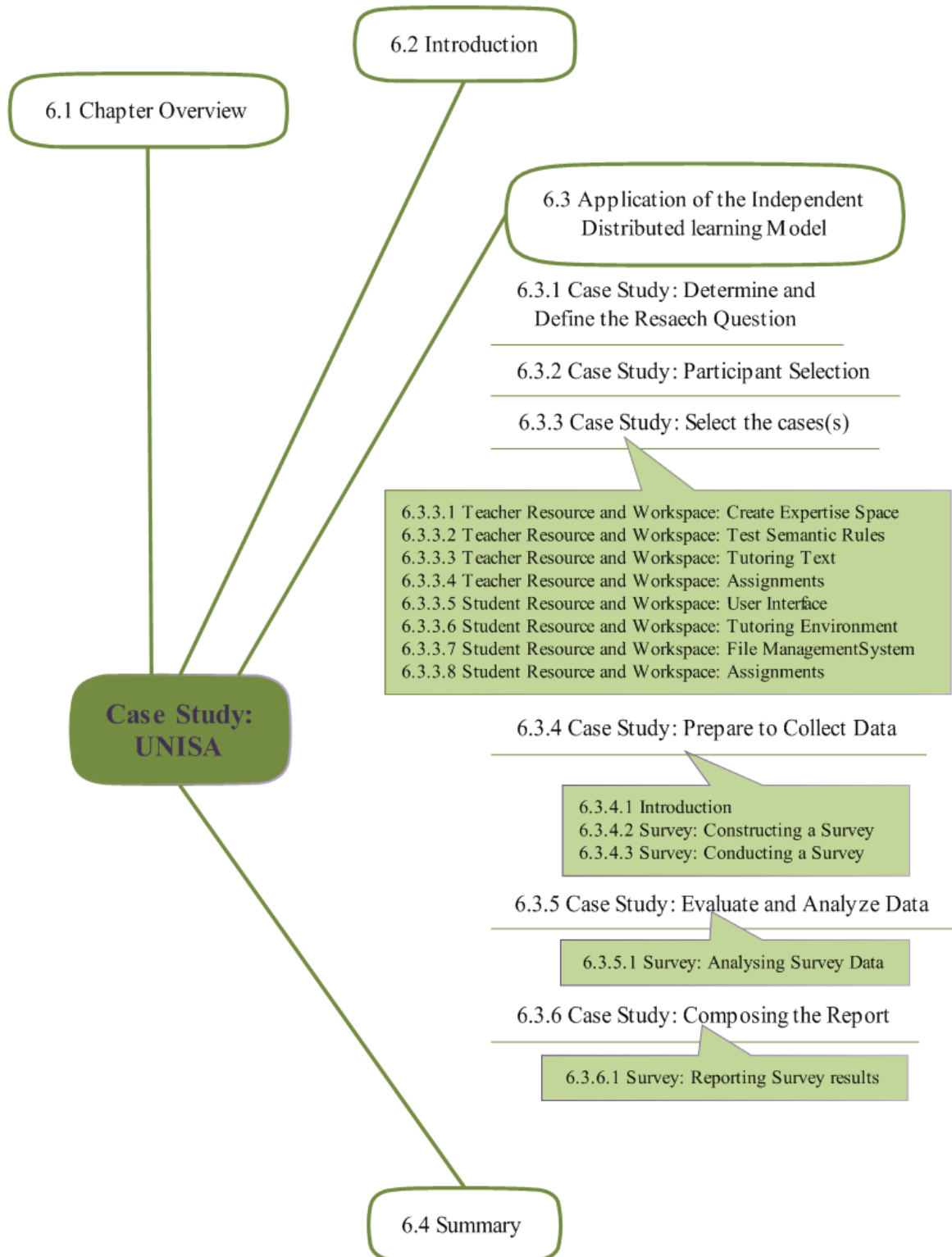
to the module webpage so that the code can be assessed electronically. The solutions to the assignments are uploaded to the module webpage and can be downloaded by the student.

5.10 Summary

The *independent distributed learning model* is based on the client-server software architecture model. The components of the model, named the *independent distributed learning model*, are the *resource space*, which acts as a server and facilitates an environment in which all the resources and functionality needed to accommodate the learning experience reside. These resources are made available to the learning space, where the student and teacher reside, through the broker space. The second component, the *broker space*, acts as a middleman, which pairs requests from the learning space with the resource space. Thus, the broker space is responsible for communication between the resource space and the learning space. The last component is the *learning space*, which facilitates an environment in which teacher and student can execute their respective tasks. The resource space combines the various resource into an area in which support is maintained. The components of the resource space are tutoring resources, assignment management and communication, which includes the module webpage, newsgroups, email and a discussion forum. The learning space consists of both the *student resource and workspace* and the *teacher resource and workspace* and can be viewed as a facility that allows the student and the teacher to fulfil their tasks. The student resource and workspace facilitates an environment in which the student can communicate, via a user interface, with the resource space, through the Internet. The components of the student resource and workspace are a user interface, tutoring material and a student file-management system. The teacher resource and workspace facilitates an environment in which the teacher can communicate with the resource space, through the Internet. The components of the teacher resource and workspace are an expertise space, semantic rule base, tutoring text and assignment management.

6. Case Study: UNISA

6.1 Chapter Overview



6.2 Introduction

The final subsidiary question, *How can the proposed model for effective teaching and learning of distributed computing be implemented at a distance-based institution of higher education?* is addressed by evaluating the proposed model through a case study and subsequent survey. The model was used in a distance educational environment and as guideline in presenting the COS3114 course to third-year students at the University of South Africa (Unisa). COS3114 is a module offered by Unisa as part of the undergraduate studies towards a Bachelor in Computer Science or Information Systems; it is a third-level advanced computer science elective module called *Advanced Programming*. Section 3.6 introduced the dynamic nature of this module, as well as the motivation for the curriculum.

As discussed in Chapter 4, a case study was used to investigate whether the proposed model satisfied the requirements and constraints identified in order effectively to teach distributed computing in a distance-based educational environment. The remainder of this chapter will show how the *independent distributed learning model* is applied in practice by implementing the six components of a case study, as identified in section 4.5.3.2 of the case-study design process. The first of the six components, as discussed in section 4.4.3.2, is to determine and define the research questions. This was already done in the first chapter, but the primary question is repeated for the sake of comprehensiveness in section 6.3.1.1 The next component is to select the design case(s) in section 6.3.2. Section 2.3 addresses participant selection, which is followed by preparation to collect data in section 6.3.4. The final two components are evaluation and analysis of data in section 6.3.5 and composition of the report in section 6.3.6.

6.3 Application of the Independent Distributed Learning Model

6.3.1 Case Study: Determine and Define the Research Question

The primary research question of this dissertation, as determined and defined in Chapter 1, is *How should distributed computing be taught in a distance-based educational environment?*

6.3.2 Case Study: Participant Selection

It is important, in case studies, that the participant pool remain relatively small (Becker, 2005). The study focuses on the embedded single-case design.

In the case of the *independent distributed learning model*, the focus was on a third-year module, COS3114: Advanced Programming. The student numbers varied from 125 to 200⁸, and two lecturers were responsible for the module. The rationale for selecting this case was discussed in section 3.6.

6.3.3 Case Study: Select the Case(s)

The selected case-study environment is a single-case design: a model to teach distributed computing in a distance-based educational environment.

Chapter 5 stated that activity diagrams capture the *behaviour* of a single case by showing the collaboration of the objects in the system to accomplish the task. Activity diagrams are now used to describe the cases identified in chapter 5.

The possible cases were identified in section 5.7 to section 5.8. Since all activities are initiated through the users in the learning space, the cases of the *teacher resource and workspace* and the *student resource and workspace* are now discussed using activity diagrams. These cases are summarised in Table 6-1. The two case diagrams are repeated below to place the discussion into context

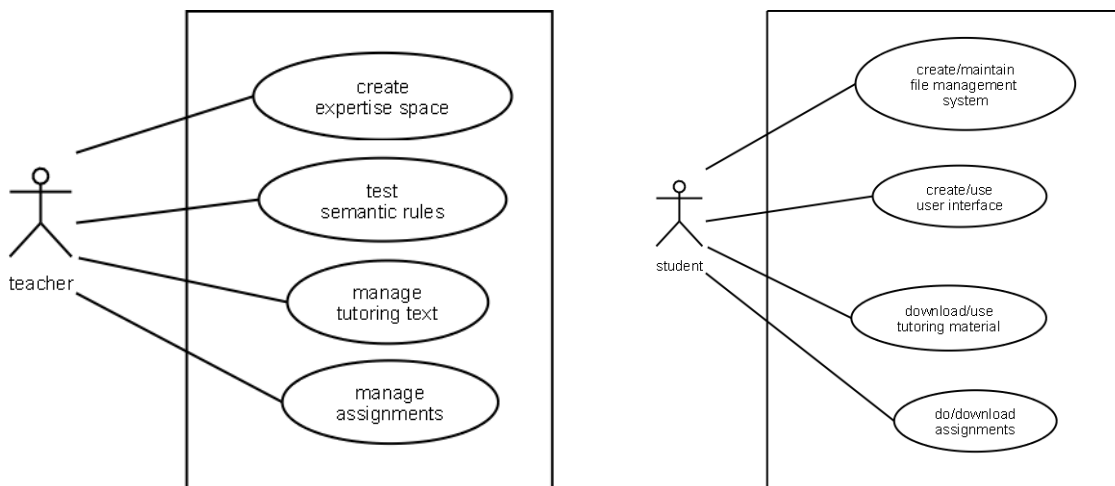


Figure 6-1. Teacher and Student Resource and Workspace Case Diagrams

⁸ Numbers gathered from statistic data between 1997 and 2005.

Table 6-1. All Possible Cases that are identified

Case	Description
Teacher Resource and Workspace	Create expertise space
	Test semantic rules
	Manage tutoring text
	Manage assignments
Student Resource and Workspace	Create/use user interface
	Download/use tutoring material
	Create/maintain file-management system
	Do/download assignments

6.3.3.1 Teacher Resource and Workspace: Create Expertise Space

According to the use-case diagram in Figure 5-11, a software environment for the *independent distributed learning model* is to be created by the teacher to accommodate the teaching and learning of distributed computing in a distance-based educational environment. Several factors need to be considered when such an environment is compiled, for example, availability of software, Windows and C++ language support, etc.

In order to create such an environment, several possibilities and combinations were tried and tested. The activity diagram below reflects an iterative process of obtaining the tutoring material, testing and integrating several possibilities before the material was packaged and downloaded for students.

The minimum software environment consisted of several components. These components include an object request broker, compiler, integrated development environment and several reference works. The specific environment for COS3114 is discussed below.

The first component in the software environment is an *ORB* (*object request broker*). The CORBA implementation of the ORB was selected. CORBA standardises interfaces and semantics for object-oriented middleware. It includes a specification for the ORB, a software library, which has standardised CORBA object interfaces and which allows clients and targets to communicate with each other across a network in a well-defined way. The acronym MICO expands to MICO Is CORBA. The intention of the project was to provide a freely available and fully compliant implementation of the CORBA standard. MICO has become fairly popular as an Open Source project and is widely used for different purposes. As a major milestone, MICO was branded as CORBA compliant by the Open Group, thus demonstrating that Open Source can indeed produce industrial strength software. The goal

is to keep MICO compliant to the latest CORBA standard. The sources of MICO are placed under the GNU-copyright notice. For a detailed discussion of how this ORB was selected, see Appendix A.

The compiler that was selected was the *minimalistic GNU for Windows* [minGW]. The reason was that MinGW provides a complete Open Source programming tool set, which is suitable for the development of native Windows programs that do not depend on any third-party C run-time DLLs. It is a command-line C/C++ compiler and utility. It compiles and links code to be run on Win32 platforms (Win95 through XP). MinGW comes with everything that is needed to compile and link both console-mode and GUI programs. MinGW does not have an accompanying *integrated development environment* [IDE].

For the *integrated development environment*, DevC++ was chosen. It is a full-featured IDE, which is able to create Windows or console-based C/C++ programs using the MinGW version of C++. An IDE is a program that allows one to edit, compile and debug one's C++ programs without using the command line. The program itself has a look and feel similar to that of the more widely used Microsoft Visual Studio. Features include C and C++ compiler for Win32 (MinGW), a debugger (GDB or Insight), customisable syntax highlighting, a powerful multi-window editor with many options, work in source file or project mode and Makefile automatic creation.

Figure 6-2 depicts the activities involved when an expertise space is created. First, tutoring material, which include OR's, IDEs, compilers, etc. were obtained. This was followed by extensive testing of the various components and various combinations of the components. An environment was identified, packaged and downloaded to the resource space.

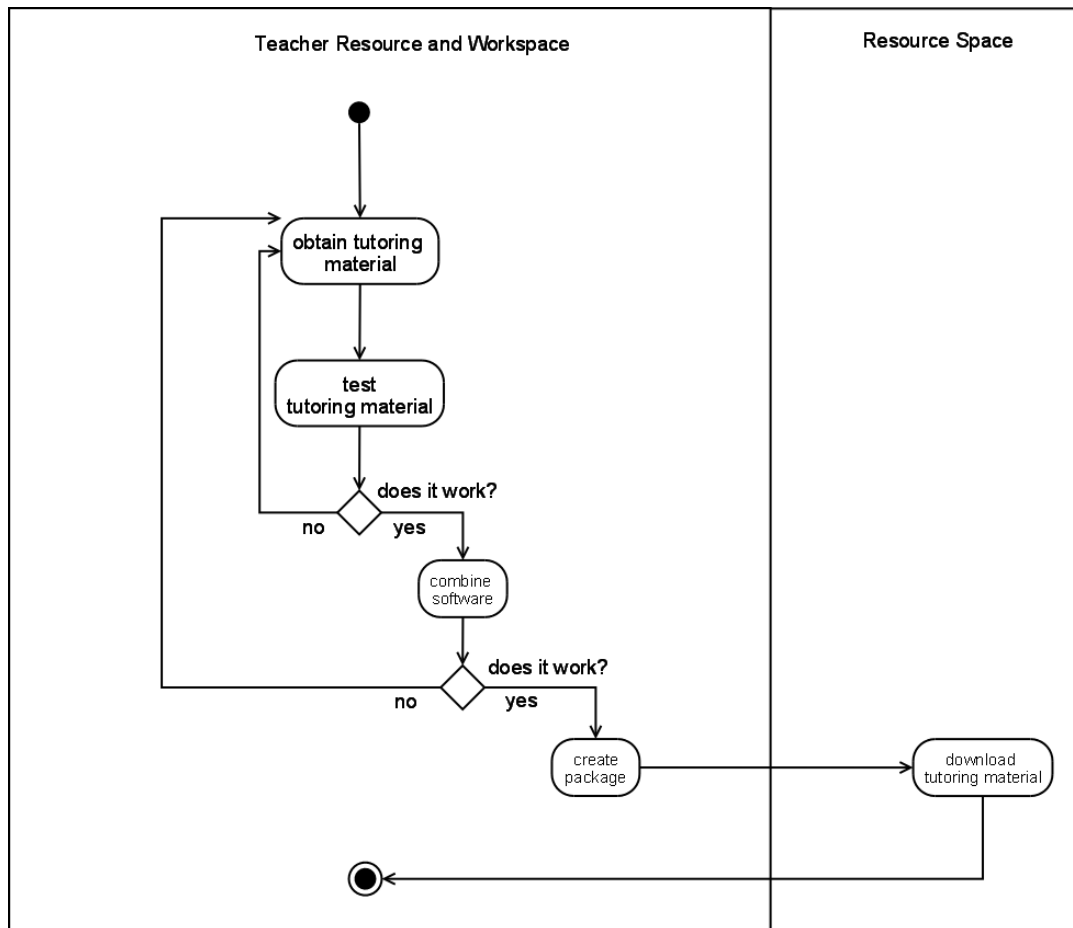


Figure 6-2. Activity Diagram - Teacher Resource and Workspace: Create Expertise Space

6.3.3.2 *Teacher Resource and Workspace: Test Semantic Rules*

The next case discussed is the testing of semantic rules in the teacher resource and workspace.

Programming in this module was done in C++. Since it is at an advanced level, testing the semantics of C++ forms a considerable part of the module. Again, it was an iterative process based on trial, error and research and is depicted in the activity diagram in Figure 6-3.

This was an ongoing activity throughout the course of the module and was usually the consequence of creating tutoring text or queries from students.

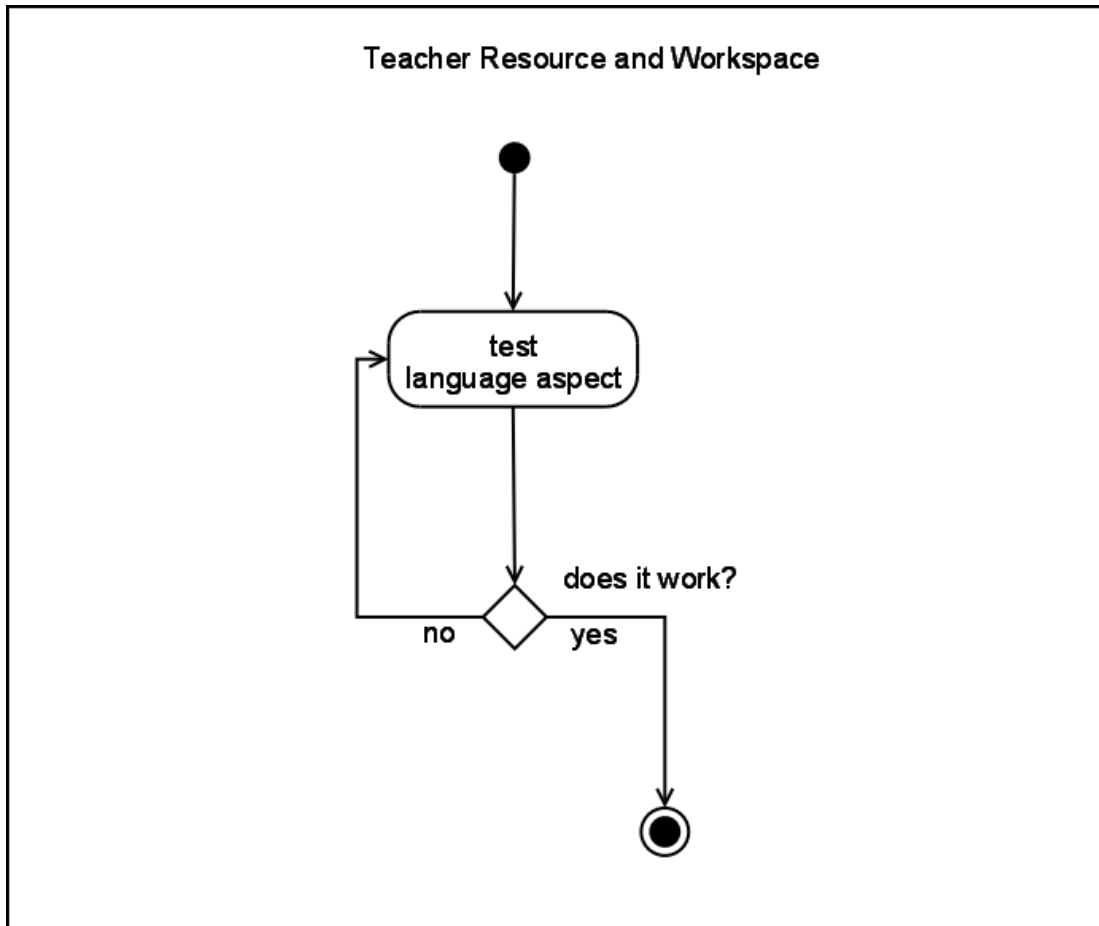


Figure 6-3. Activity Diagram - Teacher Resource and Workspace: Test Semantic Rules

6.3.3.3 Teacher Resource and Workspace: Tutoring Text

All communication with students was in the form of tutorial letters. Students were informed about the numbering of tutorial letters, since the numbering suggested the contents. **Table 6-2** depicts a summary of some of the notations.

Table 6-2. Tutorial Letter Notation

Series	Example	Description
100	101, 102, 103, 104, ...	All information to students is conveyed through this series of tutorial letters.
200	201, 202, 203, 204, ...	All solutions to assignments are published in the 200 series of tutorial letters.
500	501, 502, 503, 504, ...	This series of tutorial letters refers to study guides that accompany the prescribed works of a module.

Tutorial letter 101 formed part of the material that students received upon registration and contained, *inter alia*, information on administrative arrangements and all assignments. Another important 100 series tutorial letter was the examination tutorial letter, which stated the scope of the paper and contained sample questions.

The 200 series of tutorial letters corresponded to the assignment numbers. For example, the solutions to Assignment 1 were in tutorial letter 201.

Once a tutorial letter had been completed, it was downloaded to the Resource Space and was available to students to upload from Tutorial Material. The tutorial letter was also despatched to students by mail, commonly referred to as snail mail. It is for that reason that students preferred the electronic route.

The process is depicted in the activity diagram in Figure 6-4, and all activities were ongoing through the lifetime of the module.

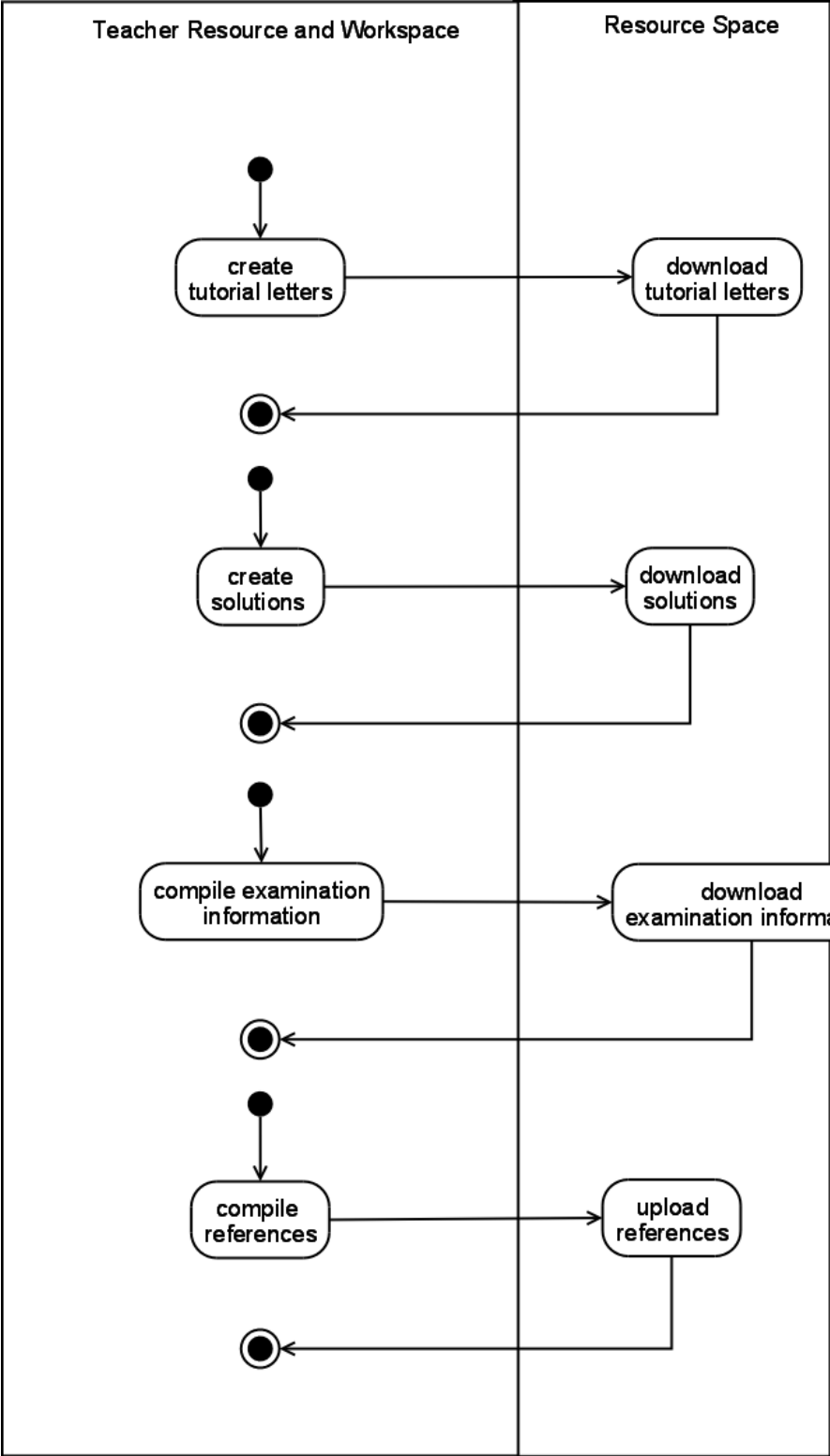


Figure 6-4. Activity Diagram - Teacher Resource and Workspace: Tutoring Text

6.3.3.4 Teacher Resource and Workspace: Assignments

The last case discussed is assignments in the teacher resource and workspace.

The module had three assignments, the last of which was compulsory. One assignment had to be submitted before a certain date, usually after two to three months of registration, to determine the active student count. This count was required by law. The first two assignments consisted of both theoretical and practical questions, which are addressed in the section 6.3.3.8. The final assignment was a project and entailed the development and implementation of a system.

The first two assignments were submitted as *written* copies and assessed accordingly. The last assignment, however, was submitted and assessed electronically. It was downloaded from the server and unzipped in a directory, which had the name of the student's number. The assignment was then compiled. If the code did not compile, a feedback report was generated and 0 marks were allocated for the workings of the program. At that stage of a student's studies, the submission of code that could not compile was not acceptable. The code was then executed, and if it did not execute correctly, three attempts were made to rectify the code. Thereafter, a feedback report was generated, and marks were allocated accordingly.

The complete process is depicted in Figure 6-5.

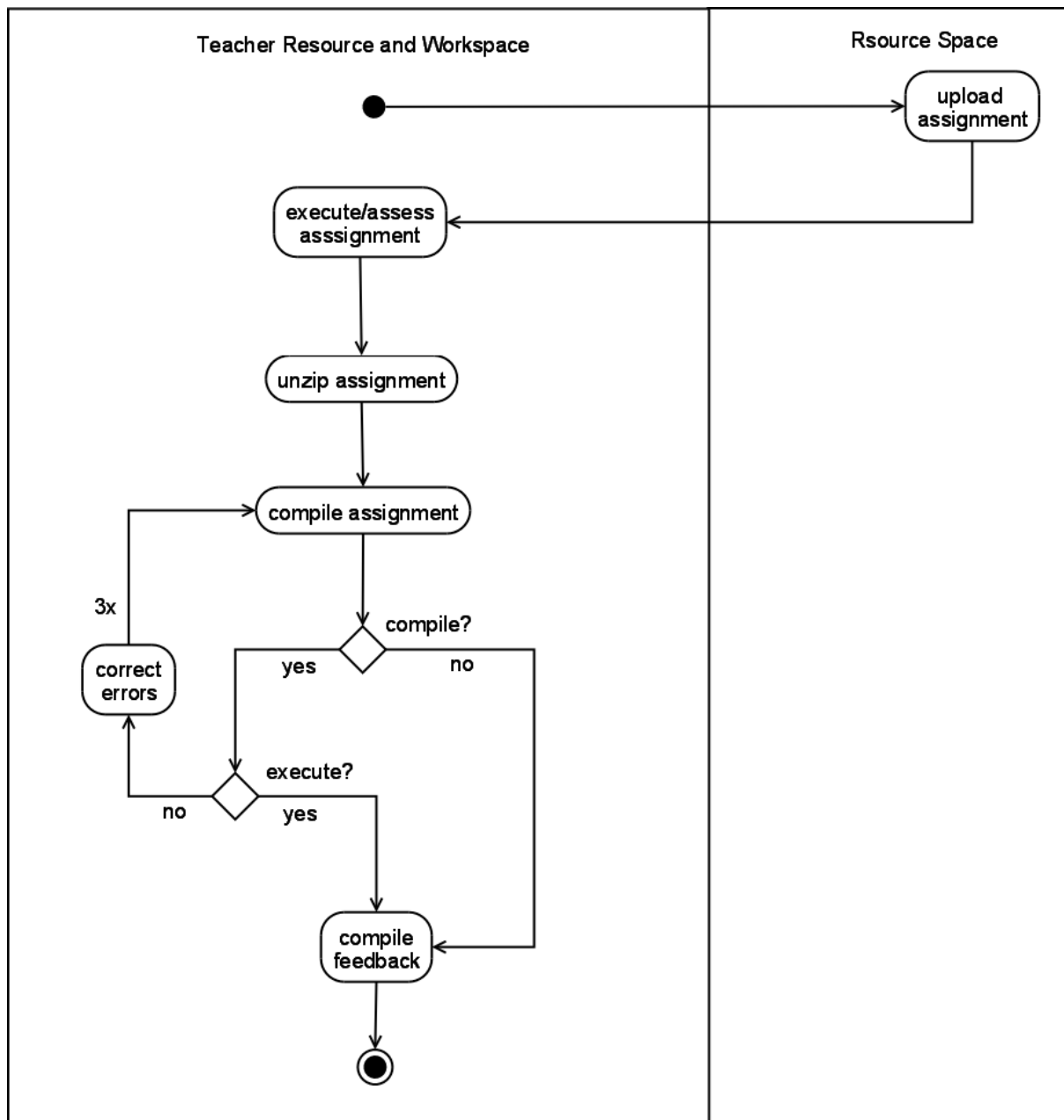


Figure 6-5. Activity Diagram - Teacher Resource and Workspace: Assignments

6.3.3.5 Student Resource and Workspace: User Interface

The user interface refers to all the different components that were needed to simulate a distributed computing environment for a student in order effectively to write, compile and execute programming assignments. After the tutoring resources had been downloaded from the module webpage, or had been received on a CDRom, the student was responsible for the creation of a user interface and installation of the software. The activity diagram below identifies the activities involved.

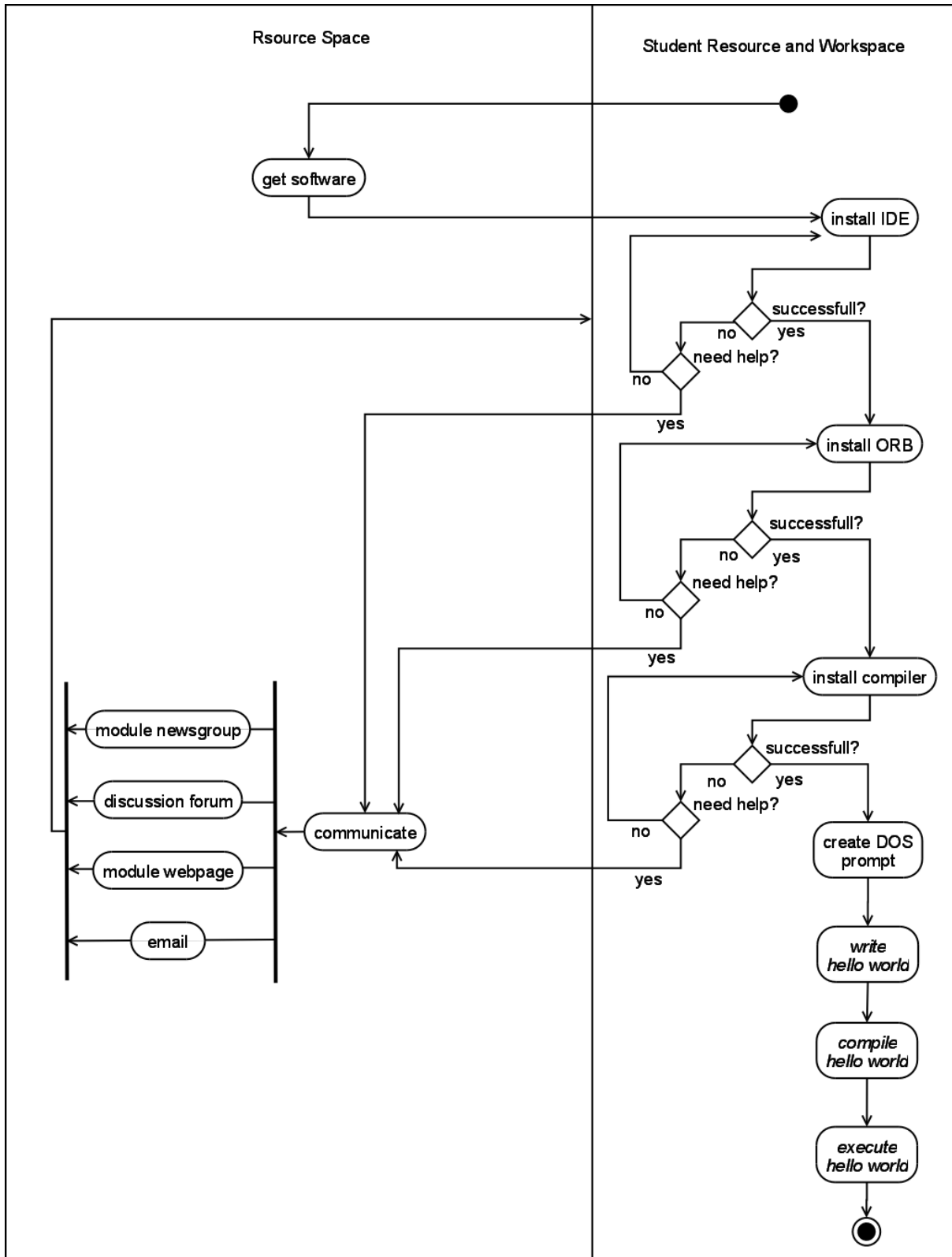


Figure 6-6. Activity Diagram - Student Resource and Workspace: User Interface

Since each activity involves a process of trial and error, each process is described in detail, in an appendix. The table below references the appendices, which appendices guide the student through the installation of the different software components. Since all good programming manuals begin with a `HelloWorld` example, this will be no exception. Appendix D gives a detailed description of writing the first *"Hello World"* program. The activity diagram indicates all channels of communication open to both student and teacher for help and information transfer.

Table 6-3. Activities of the Student Resource and Workspace: User Interface

	Activity	Appendix
1	Install IDE	C
2	Install Dev C++	C
3	Install ORB	C
4	Install compiler	C
5	Install DOS prompt	C
6	<i>Hello World</i>	D

The deliverables from the process described above were tested in Exercise 01 of the first assignment.

6.3.3.5.1 Exercise 01

The first exercise was intended primarily to test that the environment had been set up properly and to start on the road to get comfortable within it. There were no substantial deliverables that had to be submitted. A brief note, stating that the exercise had been completed and how the students had experienced it, for example, what had been difficult, impossible, useful, etc., was the only deliverable. The students were encouraged to use the discussion forum to discuss their difficulties, frustrations, successes and discoveries. Completion of the exercise should have given the students a feeling for what was involved in running and testing CORBA applications.

6.3.3.6 Student Resource and Workspace: Tutoring Environment

Although a student received all tutorial letters in hardcopy format, all tutorial letters were also available electronically. Great effort had been made to provide all reference works to students on the CDROM that they received at registration, but it was not possible. The reference works, which constituted mainly of articles, publications, other resources and URL addresses, however, could be downloaded from the resource space. Again, all channels of communication were open to both student and teacher for help and information transfer.

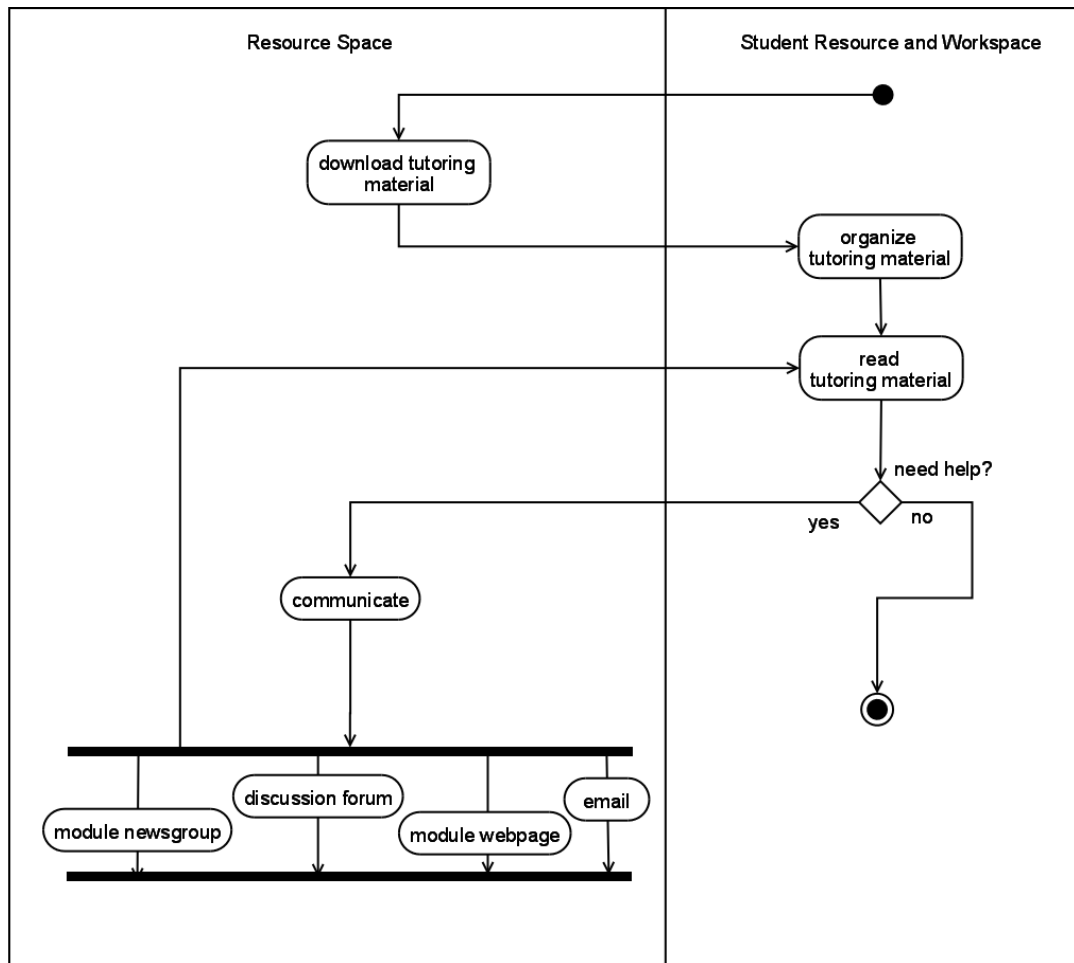


Figure 6-7. Activity Diagram - Student Resource and Workspace: Tutoring Environment

6.3.3.7 Student Resource and Workspace: File Management System

It was of the utmost importance for a file-management system to be created and maintained, because of the intricate file structure of the software and to simplify the study process. The students were also advised to keep to the suggested hierarchy to aid in the communication process when a student needed some help. The activity diagram in Figure 6-8 depicts the activities involved.

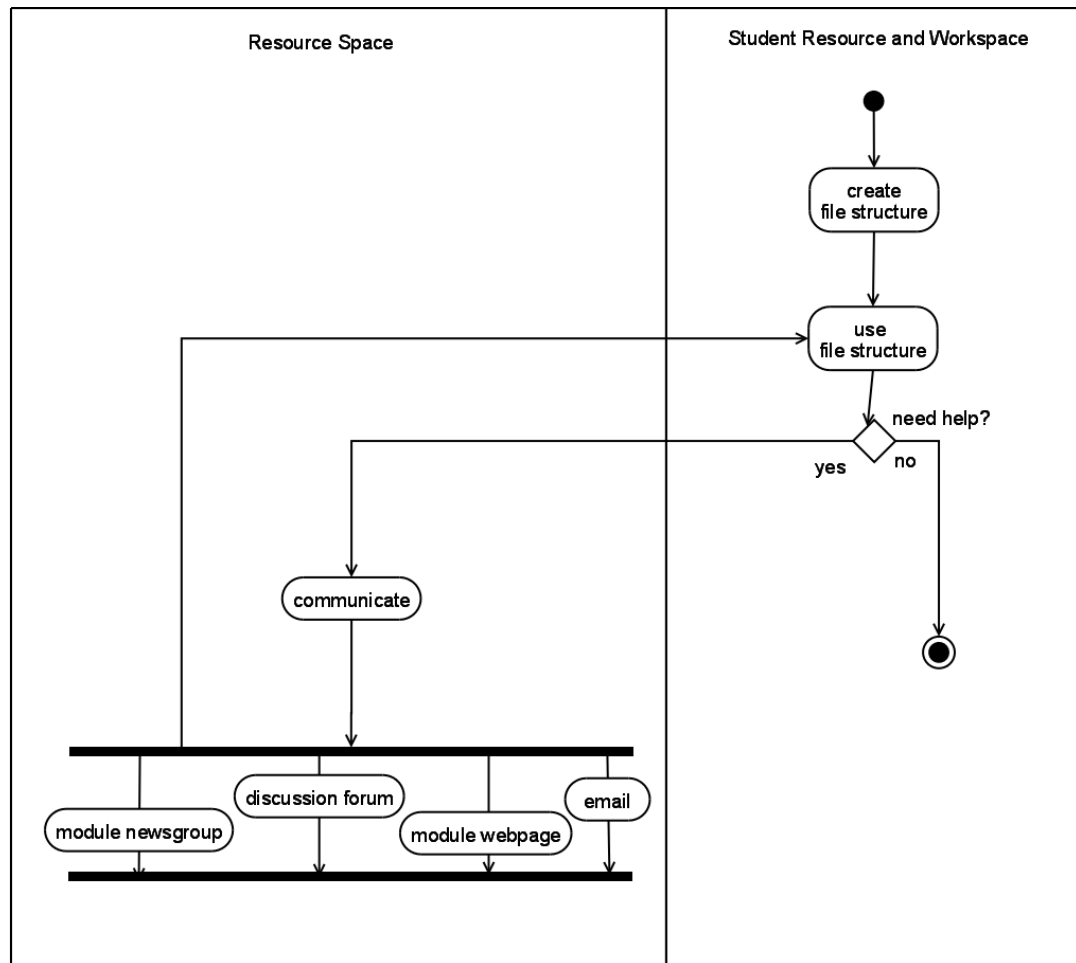


Figure 6-8. Activity Diagram - Student Resource and Workspace: File-management System

6.3.3.8 Student Resource and Workspace: Assignments

The assessment during the year was based on assignments. Before the students attempted an important assignment, a tutorial was provided, and the students have to work through this tutorial. The students, then, attempted the assignment. During the development of the assignment, the student could communicate with both the teacher and fellow students, through email and the discussion forum. Important communication also took place through the module webpage and, in particular, the module newsgroup. This was an iterative process. After completion of the assignment, it was submitted either by mail or electronically. Except for the project (last assignment), both forms of submission resulted in a printed copy being delivered to the teacher. The students could then download the solutions to the assignment from the module webpage. The activity diagram in Figure 6-9 reflects the process of assignment compilation.

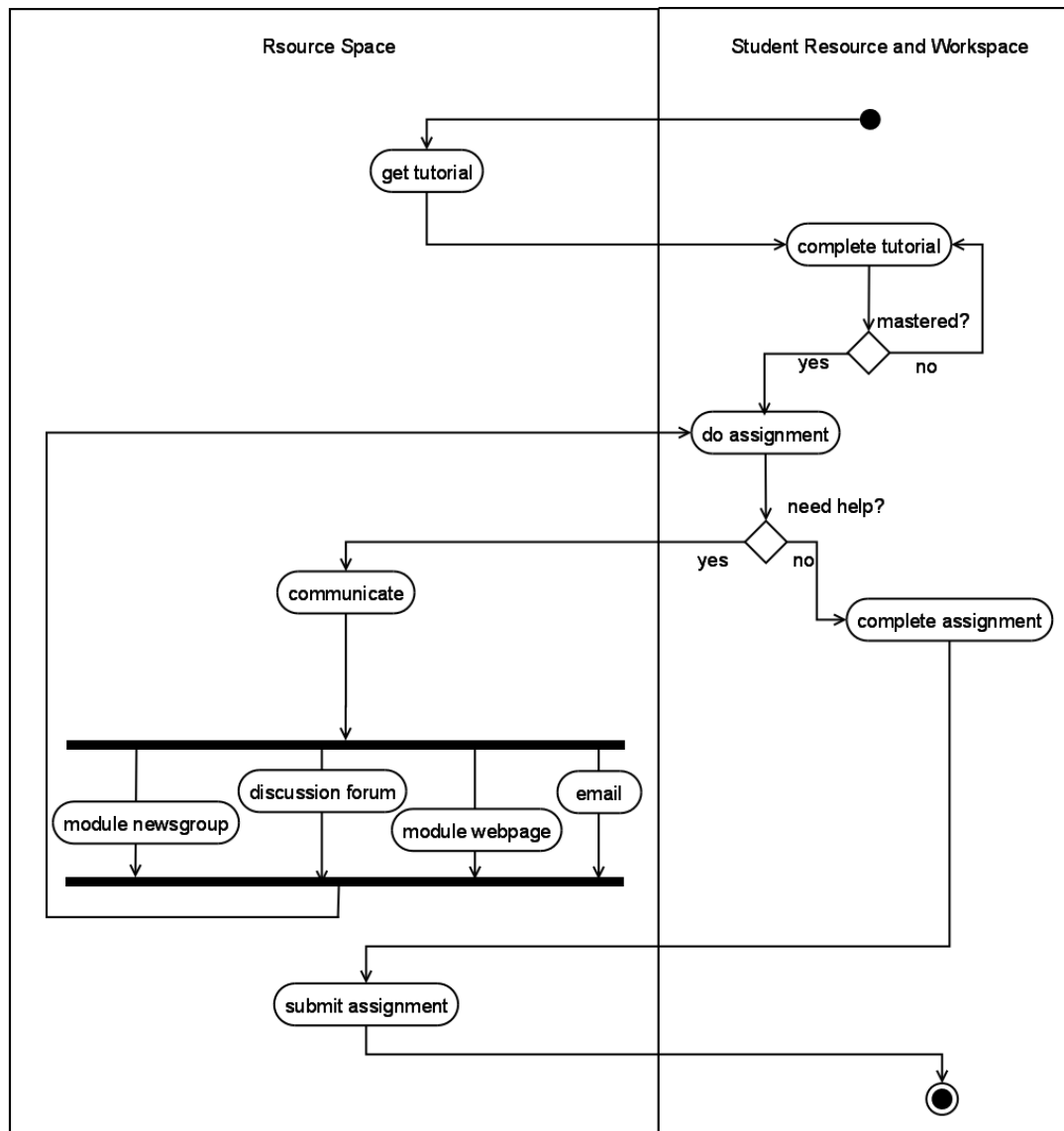


Figure 6-9. Activity Diagram - Student Resource and Workspace: Assignments

As mentioned before, the students, upon registration, received tutorial letter 101, which contained, *inter alia*, the assignments. Each assignment consisted of theoretical and practical exercises. Four exercises are now discussed. These four practical exercises formed a unit, since they built onto one another and led to the final project. They were named Exercise 02 through to Exercise 06 and were all based on the concept of a currency convertor. The table below serves as a reference.

Table 6-4. Exercises in Module COS3114

Exercise 01	Establish and test the software environment (see previous section).
Exercise 02	Create the basis and starting foundations of the Basic Currency Converter.
Exercise 03	Add functionality to the Basic Currency Converter to create an Enhanced Currency Converter.
Exercise 04	Build on the Enhanced Currency Converter
Exercise 05	Further exploration of the Enhanced Currency Converter
Project	Bank.

The students were encouraged to use the discussion forum and email to discuss their difficulties, frustrations, successes and discoveries. If a common problem occurred, the newsgroup and discussion forum were used to broadcast messages to the students. The most general problem that occurred at that stage was that the majority of students that experienced problems and difficulties had not completed the first exercise successfully.

6.3.3.8.1 Exercise 02

The purpose of the second exercise was to implement a program performing conversions from one currency to another, called the *Basic Currency Converter*. A server, which could convert between South African rand, Botswana pula and US dollars, was to be implemented, and the respective country codes to be used were “ZAR”, “BWP” and “USD”, respectively. The objective was for the server to convert backwards and forwards between the three currencies using appropriate exchange rates.

After successful completion of Exercise 01, the students were encouraged to work through the tutorial, *Running a CORBA client and server*, in Appendix E, and before attempting Exercise 02.

The implementation follows the steps of writing a distributed application, as discussed in section 2.5.3.2. These steps include the definition of the interfaces using the standard IDL and the implementation these interfaces with C++ classes. This is followed by creation of the server, which creates instances of the classes, and registration of the server. The final step is to write a client’s main function to connect to the server and to use the server’s objects.

The IDL description below was provided to the students and defined the simple currency convertor.

```
// currency.idl
interface Convertor
{
    double convert    ( in double amount,
                       in string fromCode,           // eg. ZAR
                       in string toCode              // eg. BWP
                       );
}; //interface
```

The assignment required the students to write a simple client program to test the functionality of the convertor. The deliverables of the assignment were code listings of the implementation files and the Makefile.

6.3.3.8.2 Exercise 03

A problem with the example of Exercise 02 was that there was no way of signalling an error if the client specified an invalid country code. The next IDL enhanced the previous example by providing a user exception, which could be raised in the instance of an error. Thus, the Enhanced Currency Convertor is to make provision for exception handling. The following was provided to the students:

The Assignment: Providing the following IDL,

```
// currency.idl
module Currency
{
    interface Convertor
    {
        exception unknownCode
        {
            string currencyCode;
        };
        double convert    ( in double amount,
                           in string fromCode,           // eg. ZAR
                           in string toCode              // eg. BWP
                           ) raises (unknownCode) ;;
    }; //interface
}; // module
```

implement the server and client code to raise and catch the exception in case of invalid input.

The deliverables were code listings.

6.3.3.8.3 Exercise 04

The fourth exercise corresponded to the previous exercise. The exercise continued with the theme of the currency converter, by making various enhancements along the way. In this exercise, the NamingService was to be used. Before attempting Exercise 04, the students were encouraged to work

through the tutorial *CORBA with Mico*, in Appendix F. In the exercise, the currency convertor was refined so that the server registered the convertor object with a running nameserver, and the client resolved a reference to the convertor, by looking it up on the nameserver. Exceptions had to be caught where appropriate.

Code listing had to be submitted.

6.3.3.8.4 Exercise 05

A more informative currency converter could return information about the countries about which the server knew. The assignment required implementation of a more informative currency converter by considering the IDL description provided. A new function was added to the interface, `getDetails()`, which returned a `struct`, containing information about the countries about which the server knew.

The IDL provided was:

```
// currency.idl
module Currency
{
    struct Country
    {
        string name;           // eg. South Africa, Botswana
        string currencyCode;  // eg ZAR, BWP, etc
        string denomination;  // eg. Rand, Pula etc
        double rate;          // eg. Rand equivalent
    };
    interface Convertor
    {
        exception unknownCode
        {
            string currencyCode;
        };

        double convert( in double amount,
                       in string fromCode, // eg. ZAR
                       in string toCode   // eg. BWP
                       ) raises(unknownCode);
        Country getDetails ( in string currencyCode ) raises(unknownCode);
    }; //interface
}; // module
```

The students had to implement a server that read in the information regarding the supported countries, either from a file or from `std` input, and had to implement the new method. As before, use had to be made of the name server. The client had to test the new functionality. Deliverables included code listings of the client and convertor implementation files.

6.3.3.8.5 Exercise 06

The sixth exercise assumed implementation of some sort of a container (for example, `std::list<Country>` or `std::map<string, Country>`). In the IDL provided, an additional enhancement was added in the form of a new method, `getCountries()`, which returned the list of all the supported countries using a CORBA sequence variable. The following was provided to the students:

```
// currency.idl
module Currency
{
    struct Country
    {
        string name;           // eg. South Africa, Botswana
        string currencyCode;   // eg ZAR, BWP, etc
        string denomination;   // eg. Rand, Pula etc
        double rate;           // eg. Rand equivalent
    };
    typedef sequence<Country> Countries;
    interface Convertor
    {
        exception unknownCode
        {
            string currencyCode;
        };

        double convert( in double amount,
                       in string fromCode, // eg. ZAR
                       in string toCode   // eg. BWP
                       ) raises(unknownCode);
        Country getDetails ( in string currencyCode ) raises(unknownCode);
        Countries getCountries();

    }; //interface
}; // module
```

The assignment required the implementation of the new functionality and enhancement of the client to test the functionality. The student was encouraged to scan through some of the recommended documentation and sample code provided to find examples of using the sequence. Client and convertor implementation files had to be submitted.

6.3.3.8.6 Project

The project was an extension of a simple bank-account example used in the tutorials and theoretical questions in the assignments. The students were required to implement a server conforming to the IDL specification below. An important difference from previous assignments was that the students were required to submit their source code in electronic form (see below), so that it would have been compiled and tested. The student's server was tested with our own test client. For that reason, it was

important to follow the instructions to letter. In particular, no changes were to be made to the provided IDL:

```
interface Account {

    struct Transaction {
        enum ttype {deposit, withdraw} direction;
        unsigned long amount;
    };

    typedef sequence<Transaction> Statement;

    exception Bankrupt {
        unsigned long balance;
        unsigned long amount;
    };
    void deposit (in unsigned long amount);
    void withdraw (in unsigned long amount) raises (Bankrupt);
    long balance ();
    void destroy ();
    Statement getStatement();

};

interface Bank {
    exception NotAuthorized {};
    Account open (in string name, in string password)
        raises (NotAuthorized);
};
```

There were three additional features to the original example. The first was that the application made use of a simple authentication scheme. When a new account was to be created, using the `Bank::open()` method, a name and a password had to be provided. When a subsequent call to the `Bank::open()` method was made using that name, the server had to check that the password was correct before returning a reference to the previously created account object. If the password was incorrect, a `NotAuthorized` exception had to be raised. If a previously unused name was provided, a new account had to be created.

The second additional feature was that if a client attempted to withdraw an amount greater than the available balance, a `Bankrupt` exception had to be raised, resulting in the balance remaining unchanged. The fields of the exception had to contain the available balance and the amount requested.

Lastly, the account was to keep a record of all transactions made on that account. The `Account::getStatement()` method was to return a `CORBA::Sequence` of the transactions to the client.

The students were required to submit their code electronically, so that it would have been compiled and tested. The students had to create a zipped file containing only their source code and a `Makefile`. This was not to include any executable file, compiled object file or file that had been generated by the IDL compiler. The zipped file was not to contain any subdirectory within it. If the student wished to include a short note, it was to be included as a text file called `readme.txt` (not a word-processor document). When the students' files were unzipped, the command "gmake" (or "make" on Linux/FreeBSD) was to result in an executable file, called `server.exe`. This would then have been tested for the functionality outlined above.

There were two ways to submit the zipped file, in decreasing order of preference. The first was submission via the Internet, using the procedure outlined on the official module webpage, whereas the second was submission of a CDROM, using the UNISA postal assignment-submission system.

This concludes the various activities in the teaching and learning of distributed computing in a distance-based educational environment. The next section addresses the collection and analysis of data.

6.3.4 Case Study: Prepare to Collect Data

6.3.4.1 Introduction

The survey instrument used in this study was a short questionnaire. The steps identified in section 4.5.4, that is construction of a survey, conducting of a survey, analysis of survey results and reporting of survey results, were followed.

6.3.4.2 Survey: Constructing a Survey

The following questions were asked.

- Do you think that our teaching methods of **distributed computing** in a **distance-based educational** environment were effective?
- How did you experience the following tools? Did you find it useful?
 - the editor: DevC++
 - make facility
 - minGW
 - tutorial letters
 - module web site
 - module forum
 - module email

- What methods or tools did **NOT** work well for you?
- Which aspects of distributed computing do you think need more attention?
- How is the content applicable in your working environment?

6.3.4.3 Survey: Conducting a Survey

The participants in this case study were third-year university students at a distance-based-based educational environment. The data were collected via questionnaires and were delivered to them via email, a facility offered by the University's Administrative Department. The table below presents the student numbers.

Table 6-5. Student Numbers in Survey

Description	Numbers
Number of students registered	247
Number of students cancelled	78
Total number of students	169
Number of students who were sent survey questionnaire by email	151
± 20% unrelieved survey email returns	30
Number of students with exam admission	110
Possible survey email returns	121
Survey email returns	33

The number of students registered for the module in question was 247. Seventy-eight (78) students cancelled their studies during the year, leaving a total of 169 students. The number of students who received the questionnaire via email was 151. Thirty (30) of the emails were returned as undelivered mail, possibly because students do not update their personal information. This left a possible return of 121 questionnaires. The number returned was 33, a total of 27 %.

A motivation for the size of the survey being acceptable follows. According to Kitchenham (2002), it is useful to consider the target population from the viewpoint of data analysis. If an analysis leads to any meaningful conclusion, the sample size is sufficient. Two questions arise: *Will the analysis results address the study objectives?* and *Can the target population answer our research questions?*

To answer the first question, the objective of this study was to establish and verify a model for the effective teaching and learning of distributed computing in a distance-based educational environment. Analysis of the data proved to achieve this goal.

Since all respondents were COS3114 students, the sample adequately represented the target population, and, therefore, the target population was in a position to answer the survey questions.

Thus, the sample size of 33 is acceptable.

6.3.5 Case Study: Evaluate and Analyse Data

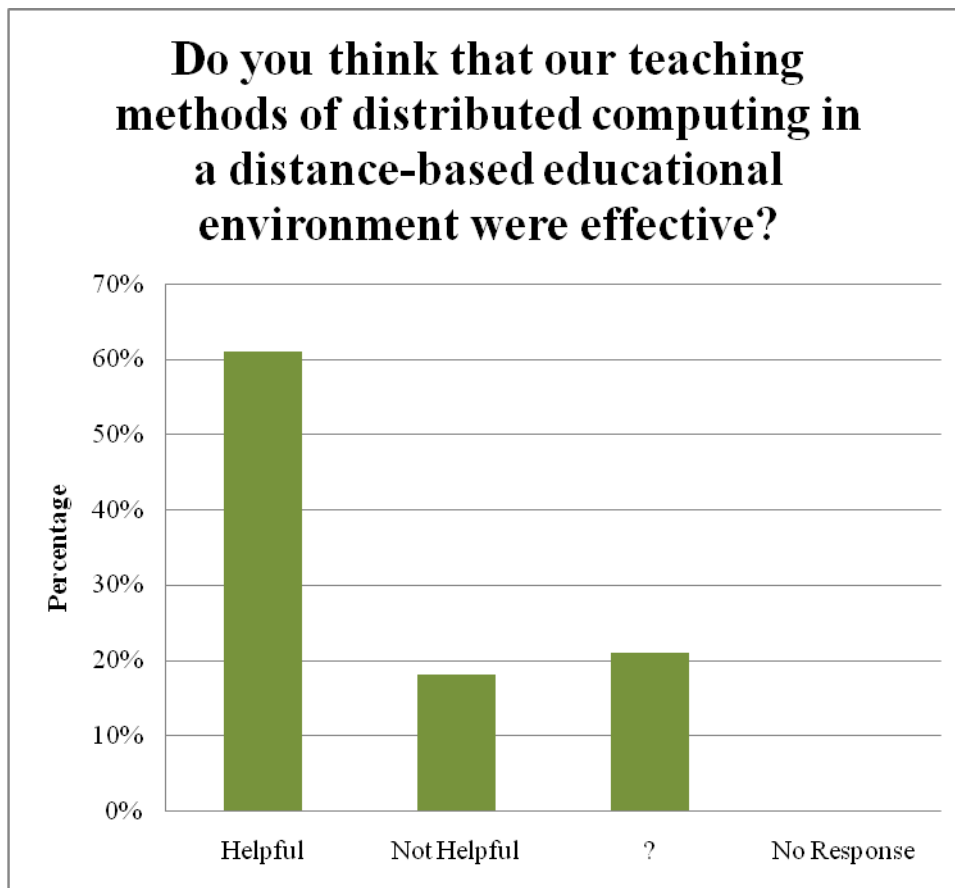
6.3.5.1 Survey: Analysing Survey Results

One of the steps identified in the process of conducting a survey is analysing the survey results. The first question in the survey, *Do you think that our teaching methods of distributed computing in a distance-based educational environment were effective?*, was aimed at the student's perception and effectiveness of the teaching methods used in the module. In the two tables below, the first presents the number, whereas the second presents the findings graphically.

Table 6-6. Survey Question #1.

Do you think that our teaching methods of distributed computing in a distance-based educational environment were effective?		
		%
Helpful	20	61%
Not Helpful	6	18%
?	7	21%
No Response	0	0%
	33	

A very high percentage (61 %) of students believed that the teaching methods of distributed computing in a distance-based educational environment were effective. The percentage of students that did not find the teaching methods effective and those who were indecisive were, respectively, 18 % and 21 %, totalling 39 %.

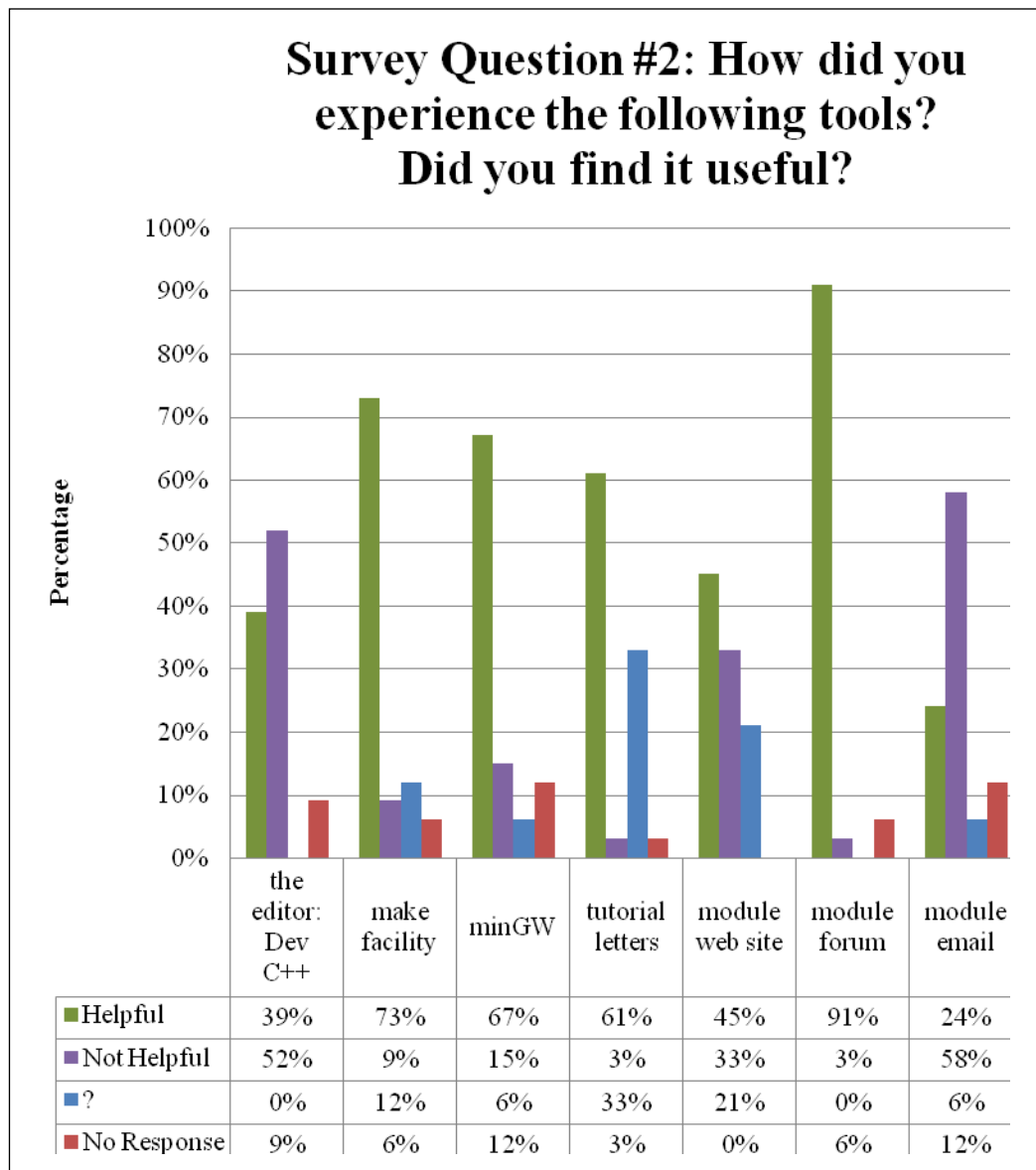
Table 6-7. Graphical Representation of the Response to Survey Question #1

The next question was aimed at obtaining insight into the use of the tools provided during the course of the module. The table below summarises the responses to survey question #2.

Table 6-8. Survey Question #2.

How did you experience the following tools? Did you find it useful?							
	the editor: Dev C++	make facility	minGW	tutorial letters	module website	module forum	module email
Helpful	13	24	22	20	15	30	8
Not Helpful	17	3	5	1	11	1	19
?	0	4	2	11	7	0	2
No Response	3	2	4	1	0	2	4
	33	33	33	33	33	33	33

The above table represents the actual respondent numbers, as graphically represented in Table 6-9 below.

Table 6-9. Graphical Representation of the Response to Survey Question #2

The first three issues addressed in the survey referred to the tools available for the development of programs and systems in distributed computing. The final four issues referred to the communication between lecturer and student, as well as student and student.

The first issue under tools available is the editor. In the case of the *independent distributed learning model*, the chosen editor was Dev C++. It is clear from the graphical representation above that only about half of the responders found it helpful. The next issue is the make facility. A high percentage (73 %) of students believed that the make facility contributed to the effective teaching of distributed computing in a distance-based educational environment. Eighteen (18) % did not respond or were indecisive. The last of the programming tools was the compiler. In the case of the *independent*

distributed learning model, the compiler was minGW. Sixty-seven (67) % of the students did find it helpful, whereas 18 % did not respond or were indecisive.

The final four issues referred to the communication between lecturer and student, as well as student and student. From the graph, it is clear that the module forum was rated as the most indispensable tool as regarded communication (94 %). This was followed by tutorial letters (61 %), the module website (45 %) and email (24 %).

Since the responses to the last three questions was in a narrative style, they are addressed in the next section, regarding composing the report. Survey question #3 was aimed at identifying the methods or tools that did **NOT** work well for students. The next question, survey question #4, was aimed at identifying aspects of distributed computing that a student perceived as needing more attention. The last question, survey question #5, was aimed at identifying the applicability of the content of the module to the student's working environment.

6.3.6 Case Study: Composing the Report

6.3.6.1 Survey: Reporting Survey Results

Survey question #1: This question was aimed at the student's perception and effectiveness of the teaching methods used in the module. The methods described in the *independent distributed learning model* were found to be helpful by 61 %, opposed to 18 % of students who found the methods not to be helpful. The number of students who did not have strong feelings about whether the methods contributed to successful learning was 21 %.

Survey question #2: As the question indicated, the aim was to obtain data on the use of tools provided in the *independent distributed learning model*. The first three tools, namely the Dev C++ editor, minGW and the make facility, are the software needed to develop distributed applications. The conclusions can thus be twofold:

- 1) The extent to which students used the software provided versus using own software.
- 2) The rating of the existing software in terms of ease of use.

The data gave strong evidence that the majority of the students made use of the software provided, but found the editor to be least helpful and made use of their own editor. After an initial installation process, which some students found to be cumbersome and difficult, the majority again found the software easy to use.

The last four tools surveyed were tutorial letters, the module webpage, the discussion forum and the module email. The results indicated that the discussion forum was found to be most helpful by 91 % of

students, whereas the module email was found to be least helpful and was used by 24 % of students. Tutorial letters, as opposed to the module website, remained the preferred way of communication.

Survey question #3: This question was aimed at identifying the methods or tools that did **NOT** work well for students. The two aspects identified were the level of the textbook and the practical component of the module, since students had to setup/install a software environment before being able to start with the module.

Survey question #4: This question related to the personal experience of the student with the module. The survey indicated that the practical part needed more attention, both in respect of tutoring text (additional exercises, more detailed discussions, etc.) and software tools.

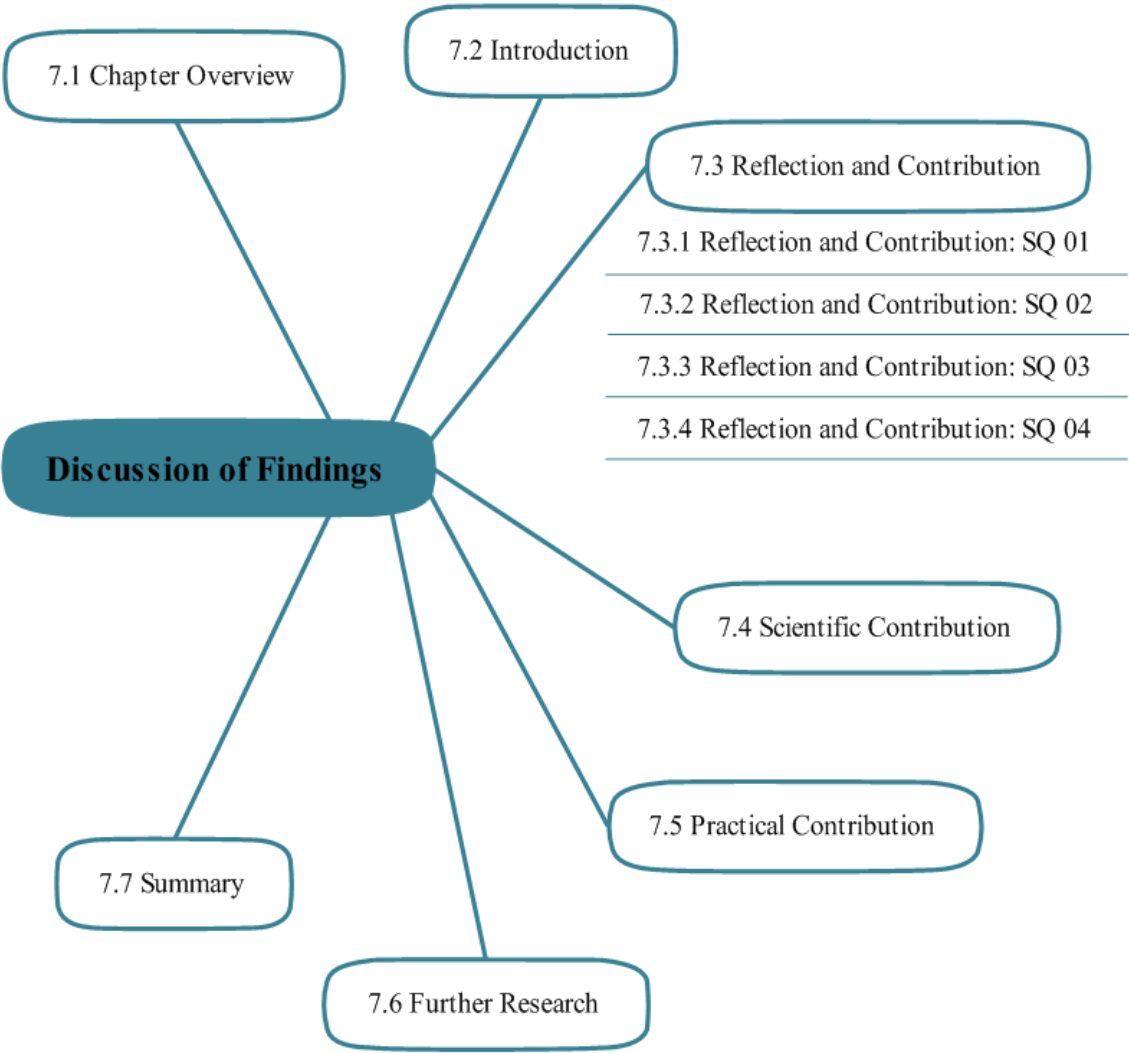
Survey question #5: Most of the students did not work in an environment in which the development of distributed systems was applicable.

6.4 Summary

The *independent distributed learning model* was applied in a third-year university module. Various scenarios/cases were implemented, presented and discussed in the form of activity diagrams. A survey was conducted, and the data were presented. The first question indicated that 61 % of the students taking part in the survey believed that the teaching methods used in the module were helpful. The next question surveyed the tools employed in the survey. The data gave strong evidence that the majority of the students made use of the software provided, but found the editor to be least helpful and used their own editor. Also, the results indicated that the discussion forum was found to be most helpful by the students. The responses to the third question indicated that the level of the textbook and the initial installation of a software environment before being able to start with the module were two aspects that the students found difficult. The second-last question indicated that the practical part was in need of more attention and, lastly, the survey concluded that most of the students did not work in an environment where the development of distributed systems was applicable.

7. Discussion of Findings

7.1 Chapter Overview



7.2 Introduction

The purpose of this study was to answer the question *How should distributed computing be taught in a distance-based educational environment to ensure effective and quality learning for the students?*

The reason for the investigation was that several models for teaching computer programming, as well as teaching programming in a distance-based educational environment, exist. These models aid in the teaching and learning of programming and ease the learning of the programming language by eliminating complexities that do not directly contribute to achieve the learning objective. Also, several technologies and languages exist for the development and implementation of distributed systems. These technologies address the complexities of designing and implementing distributed systems, being systems in which all the components are objects that can communicate with one another across boundaries such as networks, different operating systems and different programming languages. Limited literature, however, is available on models for teaching distributed computing in a distance-based educational environment.

Thus, presenting complex study material in an environment in which there are no contact sessions with the students falls within the field of education of distributed computing in a distance-based educational environment.

The research objectives were stated in chapter 1. The first objective was to identify the elements of distributed computing in order to compile a body of knowledge needed to be present in a model in order effectively to teach and learn distributed computing. The second objective was to investigate available models for the teaching of computer programming, programming in a distance-based educational environment and distributed computing in order to identify success factors for teaching distributed computing in a distance-based educational environment. The final objective was to establish and verify a model for the effective teaching and learning of distributed computing in a distance-based educational environment.

The purpose of this study was concerned with how distributed computing should be taught in a distance-based educational environment so as to ensure effective and quality learning for the students and to ensure that the required effectiveness and quality were comparable to those for students exposed to laboratories as commonly found at residential universities. Therefore, the work was subdivided into four subsidiary questions relating to the research objectives of this study. The questions were:

- **SQ1:** *What is a distributed system, and what are the aspects to be considered in designing a distributed system?*
- **SQ2:** *What are the factors contributing to the success of teaching programming, and how can these factors be integrated into teaching distributed computing in a distance-based teaching model?*
- **SQ3:** *What guidelines can the teacher involved in teaching distributed computing in a distance-based educational environment use to direct the teaching and learning process?*
- **SQ4:** *How can the proposed model for effective teaching and learning of distributed computing be implemented at a distance-based institution of higher education?*

Each of these subsidiary questions are now discussed in terms of reflection and contribution.

7.3 Reflection and Contribution

7.3.1 Reflection and Contribution: SQ 01

The first subsidiary question, *What is a distributed system and what are the aspects to be considered in designing a distributed system?* was addressed in chapter 2. In the process of answering this question, the theory of distributed systems was researched. First, the history of distributed systems showed the five generations of software systems and how these evolved to accommodate the design and development of distributed systems. Non-functional requirements of distributed systems were identified, and the various technologies that a developer has to consider when designing and implementing distributed applications were acknowledged. Since distributed systems usually employ some form of middleware, layered between the distributed system components and the network operating components, middleware was researched. Middleware hides the complexity of using a network operating system from application designers and allows the user to perceive the distributed system as a single computing facility. Research on the subject matter also revealed that the design of distributed systems and distributed objects posed new challenges. Design of distributed objects is concerned primarily with each object's interface, the object model's granularity and how the complete system is partitioned, and the development process is facilitated by object-oriented middleware.

The contribution of this research led to an understanding of the complex nature of distributed computing. This entails a unique design process for distributed system software and specialised software to accommodate development in a specialised environment of distributed systems.

7.3.2 Reflection and Contribution: SQ 02

The second subsidiary question, *What are the factors contributing to the success of teaching programming, and how can these factors be integrated in teaching distributed computing a distance-based teaching model?* was addressed in chapter 3. In order to answer this question, the learning and teaching of programming languages were explored. First, the typical problems and solutions were discussed with regard to the learning and teaching of programming languages, in environments in which there is contact or laboratory sessions, as is the practice in residential universities. Secondly, the learning and teaching programming in distance-based educational environment were discussed by researching available environments and investigating the similarities to and differences from contact sessions. The final issue that was explored was the teaching and learning of distributed computing. The available examples investigated were in laboratory environments with contact sessions.

The contribution of this research led to the identification of factors that contribute to the success of teaching programming in different environments. These factors were integrated into a set of guidelines to propose a model for teaching distributed computing in a distance-based educational environment.

7.3.3 Reflection and Contribution: SQ 03

The third subsidiary question, *What guidelines can the teacher involved in teaching distributed computing in a distance-based educational environment use to direct the teaching and learning process?*, was addressed in chapter 5. To answer this question, a model was proposed based on the guidelines identified by the first and second subsidiary question.

The components of the model, named the *independent distributed learning model*, are the *resource space*, which acts as a server and facilitates an environment in which all the resources and functionality needed to accommodate the learning experience reside. These resources are made available to the learning space, where the student and teacher reside, through the broker space. The second component, the *broker space*, acts as a middleman, which pairs requests from the learning space with the resource space. The learning space makes its functionality and needs known to the broker space. The main responsibility of the broker space is to identify and match these requests to the resource space. Thus, the broker space is responsible for communication between the resource space

and the learning space. The last component is the *learning space*, which facilitates an environment in which teacher and student can execute their respective tasks.⁹

The resource space combines the various resource into an area in which support is maintained. The components of the resource space are tutoring resources, assignment management and communication, which includes the module webpage, newsgroups, email and a discussion forum.

The learning space consists of both the *student resource and workspace* and the *teacher resource and workspace* and can be viewed as a facility that allows the student and the teacher to fulfil their tasks. The significance of the learning space is that it localises the workspace and resources of both the teacher and the student. The student resource and workspace facilitates an environment in which the student can communicate, via a user interface, with the resource space, through the Internet. The components of the student resource and workspace are a user interface, tutoring material and a student file-management system. The teacher resource and workspace facilitates an environment in which the teacher can communicate with the resource space, through the Internet. The components of the teacher resource and workspace are an expertise space, semantic rule base, tutoring text and assignment management.

After researching subsidiary question 3 and with the deliverables of the previous two subsidiary questions as input, the contribution of this research led to proposition of a model to teach distributed computing in a distance-based educational environment.

7.3.4 Reflection and Contribution: SQ 04

The final subsidiary question, *How can the proposed model for effective teaching and learning of distributed computing be implemented at a distance-based institution of higher education?* was addressed by evaluating the proposed model through a case study and subsequent survey.

The *independent distributed learning model* was applied to a third-year university module. Various scenarios/cases were implemented, and a survey was conducted.

The deliverable of this subsidiary question was a survey completed by third-year university students who had been exposed to the *independent distributed learning model* for a year.

⁹ Since the focus of this study is the teaching of distributed computing in a distance-based educational environment, we acknowledge the role of the broker space, but do not discuss the functionality and workings of the broker space further.

7.3.5 Conclusion

The research objectives were met, as discussed, namely: to identify the elements of distributed computing to compile a body of knowledge needed to be present in a model in order effectively to teach and learn distributed computing; to investigate available models for the teaching of computer programming, programming in a distance-based educational environment and distributed computing in order to identify success factors for teaching distributed computing in a distance-based educational environment; and to establish and verify a model for the effective teaching and learning of distributed computing in a distance-based educational environment.

7.4 Scientific Contribution

With regard to a scientific contribution, this study contributes to the field of teaching a complex subject, namely, distributed computing in a distance-based educational environment. The complexity of the subject was identified in chapter 2. The complexity lies in the fact that presenting this subject matter requires a *specialised developing environment* to develop *specialised software* using a *specialised design process*.

During use of the suggested model in a one-year experiment, the most pertinent challenges included:

- The first challenge was to create a specialised developing environment. This environment, which was given to the students upon registration, had to be affordable, understandable and easy to use.
- The second challenge was to provide supportive material on a regular basis. This supportive material were presented in various formats, such as tutorial letters, module webpage announcements and forum discussions.
- The third challenge was to create the student's initial environment and to get the student to write his/her first application. This challenge was due to the various platforms that exist.
- The conclusive challenge was to keep the communication channels alive. Since the forum proved to be the single most useful communication medium between students and teacher and students and students, it had to be monitored on a regular basis, especially when assignments or the project was due. This placed a strain on the available resources.

Students also perceived this, as was reflected in the surveys done after the course. Primarily, the majority of the students believed that the environment was adequate, but to get started proved to be challenging owing to the various existing platforms. The supportive material contributed to the continuity of the module. The single most used communication medium was the forum, without which students felt that the successful completion of distributed computing in a distance-based educational environment would not have been possible.

7.5 Practical Contribution

The model presented enables teachers within the field of distributed computing and specifically in distanced-based education to present complex subject matter in such a way that students can work *asynchronously*, at their *own time* and *pace*. Therefore, use of this model in presenting courses of this nature in this way should enable students to master the complexity.

7.6 Further Research

Although the existence and value of other middleware platforms are recognised, the model of this research is based on the CORBA specification. Future research includes exploration and investigation of other middleware platforms and possible software environments combinations.

A further field for research comprises an investigation into inclusion of the electronic assessment of a programming assignment, from compilation to execution, in the *independent distributed learning model*.

Most of the systems in the teaching of distributed systems present their content by visualisation, animation or simulation, or a combination thereof. The current version of the *independent distributed learning model* does not contain a visual presentation of its contents. This challenge presents a further research opportunity.

7.7 Conclusions

The research objectives were met as a result of research into each of the objectives. The objectives were to identify the elements of distributed computing to compile a body of knowledge needed to be present in a model in order effectively to teach and learn distributed computing; to investigate available models for the teaching of computer programming, programming in a distance-based educational environment and distributed computing in order to identify success factors for teaching

distributed computing in a distance-based educational environment; and to establish and verify a model for the effective teaching and learning of distributed computing in a distance-based educational environment.

During use of the suggested model in a one-year experiment, the challenges were creation of a specialised developing environment, provision of supportive material on a regular basis, initial creation of the student's working environment and the significant value of the forum. All of the above were addressed by the *independent distributed learning model*. Therefore, use of the proposed model in presenting distributed computing in a distance-based educational environment should enable students to master the complexity of distributed computing.

8. Works Cited

- 2001, C. C. *The Joint Task Force on Computing Curricula*.
- 2005, C. C. (2005). *The Joint Task Force for Computing Curricula*.
- Aversano, L. C. (2002). Understanding SQL through Iconic Interfaces. *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, (pp. 703 - 708).
- Baker, S. (1997). *CORBA Distributed Objects Using Orbix*. Addison Wesley Longman, Inc.
- Balen, H. (2000). *Distributed Object Architectures with CORBA*. Cambridge University Press.
- Barribeau, P. B. (2005). *Writing Guides: SurveyResearch*. Retrieved from Survey Research. Writing@CSU. Colorado State University Department of English. : <http://writing.colostate.edu/guides/research/survey/>
- Becker, B. D. (2005). *Case Studies. Writing@CSU. Colorado State University Department of English*. Retrieved 2008, from Colorado State University: <http://writing.colostate.edu/guides/research/casestudy/>
- Ben-Ari, M. (2001). Interactive Execution of Distributed Algorithms. *ACM Journal of Educational Resources in Computing* , 1 (2), 8 pages.
- Benns, S. B. (1999). What's in the middle? *BT Technol Journal* , 17 (2), 32 - 52.
- Berglund, A. D. (2006). Qualitative research projects in Computing Education Research: An Overview. *Proceedings of the 8th Australian Conference on Computing Education* (pp. 25 - 33). Hobart, Australia: Australian Computer Society, Inc.
- Booch, G. J. (1999). *The Unified Software Development Process* .
- Brown, C. (1999). Retrieved from The key to Universal Connectivity – MIDDLEWARE.: <http://www.sybase.com/sybmag/quarter4..ewafre/middlewarefeature/middlewarefeature.html>
- Brusilovsky, P. (2004). KnowledgeTree: a distributed architecture for adaptive e-learning. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters* (pp. 104 - 113). New York, NY, USA : ACM.
- Budd, T. (1997). *An Introduction to Object-Oriented Programming*. Addison Wesley Longman, Inc.
- Burger, C. R. (2001). A Framework to Support Teaching in Distributed Systems. *ACM Journal of Educational Resources in Computing* , 1 (1), 13 pages.
- Calloni, B. B. (1994). ICONIC programming in BACCII vs. textual programming: which is a better learning environment? *Proceedings of the twenty-fifth SIGCSE symposium on Computer science education* (pp. 188 - 192). Phoenix, Arizona, United States: ACM.
- Capetti, G. L. (2001). ORESPICS: a friendly environment to learn cluster programming. *Proceedings of the 1st International Symposium on Cluster Programming and the Grid. IEEE*.

- Capretti, G. L. (2001). ORESPICS: a friendly environment to learn cluster programming. *Proceedings of the first Internatinla Symposium on Cluster Programming and the Grid*. IEEE.
- Chen, S. M. (2005). Iconic programming for flowcharts, java, turing, etc. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (pp. 104 - 107). New York, NY, USA: ACM.
- Chua, W. (1986). Radical developments in accounting thought. *The Accounting Review* , 61 (4), 601 - 632.
- Condi, S. (1999, April). Distributed Computing, tommorrow's panacea - An introduction to current technology. *BT Technol Journal* , 13 - 23.
- Conway, R. W. (1973). Design and implementation of a diagnostic compiler for PL/I. *Communications of the ACM* , 169-179.
- Coryn, C. L. (2006). The Fundamental Characteristics of Research. *Journal of MultiDisciplinary Evaluation* , 124 - 133.
- Eaglesfield, G. (1999, January). Platform middleware. *Enterprise Middleware* , pp. 31 - 35.
- El-Khouly, M. F. (2000). Expert tutoring system for teaching computer programming languages. *Expert Systems with Applicatins* (18), 27-32.
- Emmerich, W. (2000). *Engineering Distributed Objects*. New York: John Wiley.
- Fang, W. M. (2005, May). A STUDENT MODEL FOR OBJECT-ORIENTED DESIGN AND PROGRAMMING. *JCSC CCSC:Northeastern Conference* , 260-273.
- Freund, S. R. (1996). THETIS: AN ANSI C PROGRAMMING ENVIRONMENT. *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education* (pp. 300-304). Philadelphia, Pennsylvania, United States : ACM New York, NY, USA.
- Garson, G. (2008, 5 14). *Survey Research*. Retrieved from <http://faculty.chass.ncsu.edu/garson/PA765/survey.htm>
- Gottschalk, T. (n.d.). *University of Idaho Engineering Outreach*. Retrieved from <http://www.uidaho.edu/eo/dist1.html>
- Guba, E. L. (1994). Competing Paradigms in Qualitative research. In N. L. Denszin, *Handbook of Qualitative Research* (pp. 105 - 117). London: Sage Publisjers.
- Hartley, J. S. (1973). Towards More Intelligent Teaching Systems. *International Journal of Man-Machine Studies* , 5 (2), 215-236.
- Hasker, R. (2002). HiC: a C++ compiler for CS1. *Journal of Computing Sciences in Colleges* , 18 (1), 56 - 64.
- Hentea, M. S. (2003). A Perspective on Fulfilling the Expectations. *Proceedings of the 4th conference on Information technology curriculum* (pp. 160 - 167). Lafayette, Indiana, USA : ACM New York, NY, USA .
- Hevner, A. M. (2004). *Design Science in Information systems Reserach*. Retrieved from MIS.

IEEE-CS, A. A. (2005). *Computing Curricula 2005 The Overview Report*.

IONA Making software work together. (2008). Retrieved 2008, from IONA Making software work together: <http://www.iona.com/info/aboutus/glance.htm>

Jones, D. (1996). Computing by Distance Education: problems and solutions. *Proceedings of the 1st conference on Integrating technology into computer science education* (pp. 139 - 146). Barcelona, Spain: ACM New York, NY, USA.

Kelleher, C. (2005, 37(2)). Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys* , pp. 83-137.

Kitchenham, B. P. (2002). Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Software Engineering Notes Volume 27 Issue 5* , 17-20.

Kuhn, T. (1996). *The structure of the Scientific Revolutions*. Chicago: University of Chicago Press.

Lakatos, I. (1987). *The Methodology of Scientific research Programmes*. Cambridge: Cambridge University Press.

Lenarcic, J. (2003). Engineering Education for a Sustainable Future. *Proceedings of the 14th Annual Conference for Australasian Association for Engineering Education* (pp. 239-245). Melbourne: Australasian Association for Engineering Education.

Lohr, K.-P. V. (2003). JAN - Java animation for program understanding. *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments* (pp. 67 - 75). Washington, DC, USA : IEEE Computer Society .

Lopez, N. N. (2002). WHAT: Web-based Haskell adaptive tutor. *AIMSA 2002 : International conference on artificial intelligence : methodology, systems, and applications* , 71-80.

Lydian. (2005, May 26). Retrieved from <http://www.cs.chalmers.se/~lydian/index.html>

March, S. S. (1995). Design and Natural Science Research on Information Technology. *Decision Support Systems 15* , 251 - 266.

Matsuda, H. S. (2001). Effect of using Computer Graphics Animation in Programming Education. *IEEE International Conference on Advanced Learning Technologies, 2001. Proceedings.*, (pp. 164 - 165).

McCarthy, J. (1980). Circumscription - A Form of Non-Monotonic Reasoning. *Artificial Intelligence* , 30, 27 - 39.

Mico is Corba. (2008). Retrieved 2008, from Mico is Corba: <http://www.mico.org/>

Moreno, A. M.-A. (2004). Program Animation in Jeliot 3. *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education* (pp. 265 - 265). Leeds, United Kingdom : ACM New York, NY, USA .

Moses, Y. P. (1998). Algorithm Visualization for Distributed Environments. *IEEE Symposium on Information Visualization* (pp. 71-78). Loas Alamitos, CA: IEEE Computer Society Press.

- Mowbray, T. u. (1997). *Inside CORBA Distributed object standards and applications* . London: Addison Wesley Longman.
- Myers, M. D. (1997, June). Qualitative Research in Information Systems. *MIS Quarterly* (21:2) , pp. 241-242.
- Nevanpaa, T. L. (2006). Pupil's ecological reasoning with help of modeling tool. *Proceedings of the 2006 conference on Interaction design and children* (pp. 41 - 44). New York, NY, USA : ACM.
- Nevison, C. (2001). Teaching Distributed and Parallel Computing with Java and CSP. *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*.
- Oates, B. (2006). *Researching Information Systems and Computing*. SAGE.
- Olivier, M. (2004). *Information Technology Research - A Practical Guide for Computer Science and Informatics* (second ed.). Pretoria: Van Schaik.
- Orfali, R. H. (1999). *Client/Server Survival Guide*. New York: John Wiley.
- Orfali, R. H. (1998). *Client/Server Programming with Java and CORBA*. New York: John Wiley.
- Orlikowski, W. a. (1991). Studying Information Technology in Organizations: Research Approaches and Assumptions. *Information Systems Research*, 2(1) , 1 - 28.
- Owen, C. (1998). Design research - Building the Knowledge Base. *Design Studies* , 19 (1), 9 - 20.
- Rocchetti, M. S. (2001). A web-based synchronized multimedia system for distance education. *Proceedings of the 2001 ACM symposium on Applied computing* (pp. 94 - 98). Las Vegas, Nevada, United States : ACM New York, NY, USA.
- Rock-Evans, R. (1999). Component Architectures. *Enterprise Middleware* , 7 - 20.
- Sadiq, W. C. (1998). *Developing Business Systems with CORBA* . Cambridge University Press.
- Schorsch, T. (1995). CAP: AN AUTOMATED SELF-ASSESSMENT TOOL TO CHECK PASCAL PROGRAMS FOR. *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education* (pp. 168-172). Nashville, Tennessee, United States: ACM New York, NY, USA.
- Schreiner, W. (2002). A Java Toolkit for teaching Distributed Algorithms. *ItiCSE'02*.
- Shang, Y. S.-S. (2001). An intelligent distributed environment for active learning. *Journal on Educational Resources in Computing (JERIC)* , 1 (2es), Article No. 4 .
- Siegel, J. (1996). *CORBA Fundamentals and Programming*. John Wiley & Sons. Inc. .
- SourceForge.net. (2008). *omniORB Free High Performance ORB*. Retrieved 2008, from omniORB Free High Performance ORB: <http://omniorb.sourceforge.net/>
- Sugita, K. M. (2000). Integrated visualisation-based environment for computer science education. *IFIP-16/ICEUT2000* , (pp. 225-229).
- TechTarget. (2008). *Whatis.com Target search*. Retrieved from http://whatis.techtarget.com/definition/0,,sid9_gci760724,00.html

- Trochim, W. (2006, 10 20). *Research Methods Knowledge Base*. Retrieved from Survey Research : <http://www.socialresearchmethods.net/kb/survey.php>
- Tucker, A. (2003). *Final Report of the ACM K-12 Task Force Curriculum Committee*. New York: ACM.
- Vaishnavi, V. a. (2004/5). Retrieved from Design Research in Information Systems: <http://www.isworld.org/Researchdesign/drisISworld.htm>
- Van Roy, P. H. (2002). Teaching Programming with the Kernel language approach. *International Workshop on Functional and Declarative Programming in Education (FDPE 2002)* . Pittsburg.
- Wataba, K. M. (1995). Expert tutoring system for teaching computer programming languages. *Computers in education* , 24 (3), 141-155.
- Watt, D. (1988). The Computer as Microworld and Microworld Maker. *Special issue on preservice education in educational computing* , 20 (1), 81-89.
- Webopedia. (2008). *Webopedia*. Retrieved from <http://www.webopedia.com/>
- Webster Dictionary and Theasaurus*. (1992).
- Welsh, T. (1999). Putting Middleware in context. Enterprise Middleware. February 1999. . *Enterprise Middleware* . .
- Xinoglos, S. (2002). An Integrated Programming Environment for Teaching the Object-Oriented Programming Paradigm. *EurAsia-ICT, LNCS 2510 EurAsia-ICT, LNCS 2510*, (pp. 544-551).
- Yin, R. (2003). *Case Study Research Design and Methods*. USA: Sage Publications.
- Zhang, G. C. (2002). Design of an Effective Learning Method SQ3R Based Distance Learning System. *First International Symposium on Cyber Worlds (CW'02)* (p. 318). Washington, DC, USA: IEEE Computer Society.

Appendix A – Software Environment selection

A.1 Introduction

This Appendix describes the process of establishing a development environment for the students. An extensive search for an appropriate environment for the teaching has been undertaken. Three development environments was identified and researched. A short description of each will follow, as well as the motivation of the choice made.

A.2 Three development environments

A.2.1 OmniORB

omniORB3 is a robust, high-performance CORBA ORB, developed by AT&T Laboratories Cambridge. It is one of only three ORBs to be awarded the Open Group's Open Brand for CORBA. This means that *omniORB* has been tested and certified CORBA 2.6 compliant (SourceForge.net, 2008). AT&T Laboratories Cambridge closed down in 2002 as part of AT&T's global research cuts. With the lab's closure, *omniORB* became an independent project, but several of the original developers continue their involvement. *omniORB3* is freely available.

In their lifetime, AT&T was advocates of good Distributed Systems practices, and wholeheartedly endorsed the efforts of the OMG in developing open standards for object oriented systems. The OMG vision of a common architectural framework enabling a world of connected objects was something the developers of *omniORB* actively shared. Standards that endure are always pragmatic, and the membership and composition of the OMG ensures that specifications are always practical; respond to genuine domains of interest and most importantly provide real solutions.

AT&T's particular interest in CORBA derived from their pioneering work in mobile and personalised computer and communications systems, such as the Active Badge system and Teleporting. An architecture such as CORBA plays an essential role in the control and coordination of heterogeneous environments spanning a wide variety of operating system and hardware platforms. In their laboratory, the inclusion of their own in-house platforms was one of the motivating factors in creating from scratch a CORBA 2 compliant ORB which is called *omniORB*.

The developers were extremely keen to promote the spread and use of CORBA. To this end, they had made *omniORB3* freely available in source form and have so contributed a significant component of software engineering technology to the community. They believe the simplicity and efficiency of their implementation is complementary to commercial offerings, and that it will appeal particularly to the research and academic communities. They encourage the use of it, to port it, and to fix bugs in it, extend and generally improve it. It is simply required that this is done under the terms and conditions of the GNU General Public License and GNU Library.

A.2.2 Mico

The acronym MICO expands to MICO Is CORBA. The intention of this project is to provide a freely available and fully compliant implementation of the CORBA standard. MICO has become quite popular as an OpenSource project and is widely used for different purposes. As a major milestone, MICO has been branded as a CORBA compliant by the OpenGroup, thus demonstrating that OpenSource can indeed produce industrial strength software. The goal is to keep MICO compliant to the latest CORBA standard. The sources of MICO are placed under the GNU-copyright notice. The following design principles guided the initial implementation of MICO:

1. Start from scratch: only use what standard UNIX API has to offer; don't rely on proprietary or specialized libraries.
2. Use C++ for the implementation.
3. Only make use of widely available, non-proprietary tools.
4. Omit "bells and whistles": only implement what is required for a CORBA compliant implementation.
5. Clear design even for implementation internals to ensure extensibility.

MICO is fully interoperable with other CORBA 2.3 implementations, such as Orbix from Iona or VisiBroker from Inprise. The authors have worked very hard to make MICO a usable and free CORBA 2.3 compliant implementation. Their initial request was to contribute to the development of MICO by implementing those parts of the CORBA standard, which were still missing in MICO. Although MICO is fully CORBA 2.3 compliant, there were some parts of the standard (like the CORBAServices) which were not mandatory and which were not implemented. They hoped that their decision to place the complete sources of MICO under the GNU public license will encourage other people to contribute their code.

A.2.3 Orbix from IONA

IONA was founded in 1991 in Dublin, Ireland and has a history of providing distributed, standards-based solutions to IT organizations with complex, heterogeneous computing environments and challenging integration problems. Since the early 1990s, IONA has built its integration products around two very significant open industry standards, initially CORBA and more recently Web services, and a unifying approach to designing and implementing large-scale systems referred to as service-oriented architecture, or SOA.

IONA's story began in 1983 in the computer science Department of Trinity College in Dublin. Researchers at the Department spent much of the decade researching the ability to make computers, and the software that runs them, work together collaboratively. In the years that followed, they continued the distributed computing research that would eventually become patented technology. In 1991, IONA was officially incorporated as a campus company in Trinity (the name is taken from a small island between Ireland and Scotland, which was home to monks who transcribed early Christian literature in the ninth century). In 1993, IONA shipped its *Orbix* product, which enables distributed computers and software systems to work together collaboratively. IONA left the Trinity College campus and began opening offices around the world.

Orbix is a software environment that allows you to build and integrate distributed applications. *Orbix* is a full implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture

(CORBA) specification. *Orbix* is engineered to support the integration of CORBA with other technologies, especially J2EE and Web services. *Orbix* has been used to encourage the reuse of existing systems by replacing their proprietary application interfaces with standards-based interfaces; to build new, service-oriented systems in Java or C++; to extend the value of mainframe assets by exposing them as services; and to non-intrusively augment existing systems with new functionality. *Orbix* meets the needs of enterprise IT organizations by providing a set of enterprise qualities of service for security, asynchronous messaging, management, transactions, load balancing, and fault tolerance.

A.3 Criteria for selection of a suitable development environment

The issues considered for a suitable development environment for COS3114, were:

- The developer's reputation.
- Availability.
- Windows support.
- C++ language support.
- Makes use of a free compiler.

Mico was chosen based on the criteria mentioned above.

Appendix B – Additional Resources

B.1 Introduction

A number of additional documents are included on the CDROM which students may find useful. In particular, the following were included.

B.2 The OMG CORBA specification.

This document can be found in `docs/CORBA/OMG/OMGCorbaSpec.pdf`. It is 1190 pages long and you are certainly not required to read it all! It is included primarily as a reference document. If you are unsure about a concept and require clarity or further explanation, you should be able to get the authoritative account by looking up in the index or table of contents of this document.

B.3 The OMG C++ mapping specification.

CORBA IDL can be mapped onto a number of different languages, including C++. This document describes in detail the C++ language mapping. It is useful as reference document.

B.4 The CORBA NameService specification.

You will be required to make use of the CORBA Nameservice in assignments 2 and 3. Whereas you will mostly be able to learn by example, this document is the definitive reference.

B.5 The Mico manual.

This is available in postscript format under `mico/docs` of your installation. To view postscript files under MSWindows you will have to install `ghostview` from the `utils` directory of your CDrom. We have also provided a converted pdf version under the `docs/CORBA/mico` directory of the CDrom, but the on-screen font rendering is very poor (it will print fine). The manual is not very comprehensive and it does not address the specific compiler environment which you are using. Nevertheless, there is some useful background information. Chapter's 3 and 4 are worthwhile reading, though please note that the compiler and linker wrappers (`mico-c++` and `mico-ld`) referred to only exist in a UNIX installation.

B.5 Orbacus training guide.

IONA Technologies (<http://www.iona.com>) is probably the most significant commercial CORBA vendor in the world. They produce a number of CORBA products, including a commercial CORBA environment called Orbix, as well as a stripped down version known as Orbacus. We have been given permission to distribute Orbacus to our students (it is available in source code form under `/install/orbacus` on your CDrom). The tutorial which accompanies Orbacus is available in `docs/corba/training-1.0.7.zip`. The student workbook is an extremely valuable source of information for programming CORBA applications in C++. Most of this material is based on the C++ CORBA language spec, so is readily portable to Mico applications as well.

Appendix C - Software Installation

C.1 Introduction

This Appendix describes the installation of the required software for COS311 as described in the practical guide (a tutorial letter).

Essentially the software consists of a C++ compiler and a CORBA environment. There are a number of permutations of compilers, CORBA ORBS and operating systems which could be used to achieve the objectives of the practical part of this course. We have tried to struck a balance between

- allowing students to work in their preferred environment,
- providing a set of freely distributable tools and
- providing detailed guidance and support for students and their environment.

C.2 The CDROM

On the CDROM, the minGW port of the `gcc` compiler for windows as well as a binary distribution of the Mico ORB, compiled with the same compiler, have been provided. Both of these packages are freely available and redistributable under the terms of the GNU General Public License. Currently this is the recommended environment for most students - it is sufficient to do all of the practical assignments, we have some experience in using it and its use is fully described in this practical guide.

Permission has also been granted from IONA to distribute their Orbacus environment. Orbacus is a minimal CORBA implementation (i.e. not IONA's flagship ORBIX product). It is made available on the CDROM in source form. Students wishing to use Orbacus will have to compile it for themselves. (This process was not described in the practical guide, but was discussed on the discussion forum for COS311¹⁰).

The Mico ORB (v2.3.11) is also available in source form on the CDrom. This is so that students can compile and use it with Linux or FreeBSD if they so choose. Students who use Visual C++ can also compile and use Mico on Windows with this compiler. Again this option was not fully supported in the practical guide, but was be supported via the discussion forum. The combinations of supported environments are summarized in Table C-1 below. The remainder of the practical guide was devoted to giving step-by-step guidance to install minGW and Mico on Windows.

¹⁰ It is important to note that Orbacus will not currently compile using the minGW compiler on Windows. If you are working with Orbacus on Windows you will need to have access to a Microsoft Visual C++ compiler.

C.3 Environments

Table C-1. Supported Environments

Compiler	ORB	OS	Support	Comment
mingw gcc 3.2.3	Mico	Win98/2000/XP	Full	Recommended for the busy
VC++ >= 6.0	Mico	Win98/2000/XP	Forum	Compile Mico from source
VC++ >= 6.0	Orbacus	Win2000/XP	Forum	Compile Orbacus from source
gcc >=2.95	Mico	Linux	Forum	Compile Mico from source
gcc >=2.95	Orbacus	Linux	Forum	Compile Orbacus from source
gcc >=2.95	Mico	FreeBSD	Forum	Compile Mico from source

C.4 Step 1: Installing the mingGW compiler:

Simply run the self installing executable by clicking on the icon in

```
<cdrom drive>:/install/mingw/mingw_gcc3.2.3.exe
```

The only prompt you will have to contend with asks you the location to install the compiler into. The default option will be fine for most purposes. Avoid using a path with spaces in the name (e.g. My Programs) as this can cause some difficulties. In this guide we will assume that you have installed into `c:/unisa/mingw`.

C.5 Step 2: Testing the compiler

Before using the compiler you must set up your executable search path correctly. On windows systems this involves adding the `mingw/bin` directory to your `PATH` environment variable. The details of how your `PATH` variable is set will vary from system to system. For example, on Windows 2000 and Windows XP it is set via the control panel. We are assuming you know how to set it on your own system. If you have difficulties please use the discussion forum.

Having set the `PATH` you should now open a console window (DOS box). Over time you will be opening a number of these so it may be a good idea to create a convenient shortcut on your desktop. From the command prompt invoke the C++ compiler by typing:

```
g++
```

Your system should respond with

```
g++: no input files
```

If not, go back and verify your path setting. If you got the correct response, create a simple 'hello world' application in your favorite programmer's editor and compile it as follows:

```
g++ hello.cc -o hello
```

This should result in an executable called `hello.exe` (assuming there were no errors in your source code!). So far so good. We will go through some more things to do with your compiler in section B. But first we will describe the installation of the mico development libraries and binaries.

C.6 Step 3: Installing the Mico binaries

There is a file on your CDrom called `/install/mico/mico2311-win32bin.zip`. Unzip this file into the location of your choice. Again avoid directories with spaces in the name. In this guide we will assume that you have unzipped into `c:/unisa/mico`. This should create a directory tree containing Mico with subdirectories `include`, `libs`, `demo`, `test`, `doc` and `bin`.¹¹ Now you need to add the `bin` directory to your system `PATH` so that your programs will be able to find the Mico executables and `dll`'s.

Errata - The `Makevars` file in the `mico/demo` directory indicates on the first line that it expects to find mico installed in `c:/mico`. This is contrary to your installation instructions, which recommend you install into `c:/unisa/mico`. Change the first line of `mico/demo/Makevars` to reflect the exact location you installed into - most likely `c:/unisa/mico`.

C.7 Step 4: Testing Mico

A quick verification that your `PATH` is set correctly is all that is needed here. Open a command console and type

```
idl
```

to launch the Mico `idl` compiler. If you get an error message complaining that `idl.exe` cannot be found review your path setting. Otherwise the `idl` compiler will execute and output a message about the proper way to call it.

Now that your `PATH` is properly setup, the best test of both your mico installation and the compiler installation is to try and compile something. The examples in `c:/unisa/mico/demo/poa` should compile “out of the box”.¹² Open a command console. Type: `cd \ unisa\ mico\ demo\ poa\ account-1` followed by

```
gmake
```

This should begin a lengthy process of building a simple demo application. Don't worry for the moment about the content of the code. We are just verifying that the environment is in order. You will learn more about the `gmake` command below.

We are assuming if you are reading this that you have completed the installation and verification steps above. At this stage you have a compiler and some library code installed and you want to start using them. We have called this section Software Mechanics because it addresses the mechanics of compiling and linking source code to

¹¹ The binaries and libraries were compiled using the provided minGW gcc compiler. You will not be able to use them with another C++ compiler. If you are, for example, using Visual C++ you will have to extract the Mico source code from the CDROM, follow the instructions and recompile.

¹² We have modified most of the Makefiles in these examples to work with the mingw port of gcc. Some other examples will not build “as is” without fixing the Makefiles. More about Makefiles in Appendix [B.4](#)

create executable programs. To do this effectively you need to understand the way that C++ source files are organized, the ways of invoking your compiler to compile them and how to link the resulting object files plus any external object code (such as the CORBA libraries) to create your executable program. This section is necessarily compiler specific. We describe using the `gcc` compiler.

Appendix D – Software Mechanics

D.1 Introduction

Appendix D contains the tutorial letter received by the students after they have successfully completed the first Assignment, which basically required the successful installation of the software. Since all good programming manuals begin with a `HelloWorld` example, this will be no exception. This tutorial letter gives a detailed description of writing the first “*Hello World*” program.

We are assuming if you are reading this that you have completed the installation and verification steps as described in the practical guide. At this stage you have a compiler and some library code installed and you want to start using them. We have called this section Software Mechanics because it addresses the mechanics of compiling and linking source code to create executable programs. To do this effectively you need to understand the way that C++ source files are organized, the ways of invoking your compiler to compile them and how to link the resulting object files plus any external object code (such as the CORBA libraries) to create your executable program. This section is necessarily compiler specific. We describe using the `gcc` compiler.

We begin by taking some simple code and compiling it manually by invoking the compiler from the command prompt. You will soon see that, except for very trivial cases, this is a tiresome and error prone activity. Nevertheless it is useful to understand the various compiler options. In section B.4 we will show you how to automate this build process using GNU `make`.

We begin by taking some simple code and compiling it manually by invoking the compiler from the command prompt. You will soon see that, except for very trivial cases, this is a tiresome and error prone activity. Nevertheless it is useful to understand the various compiler options. In section B.4 we will show you how to automate this build process using GNU `make`.

D.2 First steps - Hello World

All good programming manuals begin with a `HelloWorld` example so this will be no exception.

```
// file: hello.cc
#include <iostream>
int main(int argc, char* argv[])
{
    int y;
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

We have already seen in the previous section how to compile this and create `hello.exe`, but let us try and break down the steps.

D.2.1 The preprocessor

The second line in the program is a directive to the preprocessor. All lines which begin with # are preprocessor directives. This particular line is telling the preprocessor to include the contents of the file `iostream` into this one. The preprocessor is not as complicated a program as a C++ compiler and typically it is used to do pretty mundane things. In this case we are asking it to do a massive copy and paste operation on our behalf before compiling. We need to do this because our program is making reference to `std::cout` and `std::endl` which are both declared in the `iostream` file. Just for fun we can force `g++` to invoke the preprocessor and stop there without doing any further compiling:

```
g++ -E hello.cc -o processed.cc
```

You should be able to see in a text editor that the file `processed.cc` is the same `hello.cc` with the contents of `iostream` pasted into place as requested. It is worthwhile to ponder how the preprocessor knows where to find the `iostream` file. The answer is that it has a hardwired set of directories which it searches for files to be included. These are in the `include` subdirectory of your `mingw` installation. If you want to include files from other places there are ways to tell the preprocessor to look in user defined places. This will be described in more detail below.

There are also other uses for the preprocessor which we will come across later. Including the contents of other files is an important one. Others include conditional compilation, defining constants and macros and aiding with debugging.

D.2.2 The compiler

Preprocessing C++ files is not generally an end in itself so we wouldn't normally use the `-E` compiler switch. We are usually more interested in taking C++ code and compiling it to object code (or machine code). This is normally done using the `-c` switch as below:

```
g++ -c hello.cc
```

The `-c` switch tells the compiler to compile the `hello.cc` file and generate an object file called `hello.o`. You should be able to see this file in the same directory as your source file, but you won't make much sense of it if you try to open it with a text editor.

There is a utility for examining object files called `nm`. If you type in the following

```
nm -C hello.o
```

you will see a dump of the symbolic information contained in `hello.o`. Its not particularly human readable but this is all essential information used by the linker in the next step to create the executable. Before looking at the linker, we should make the comment that the previous compiler step was actually two steps. The `gcc` compiler actually compiles the C++ code down to assembly language first, and then assembles this to create the object

code.¹³ To see an example of assembly language we can instruct our compiler to stop before the assembly stage:

```
g++ -S hello.cc
```

will produce a file called `hello.s` which you can look at in a text editor. Unless you are mad keen on assembly language, you wouldn't normally want to do this. In case you do, you can still take the resulting assembly code and produce the object code by typing:

```
g++ -c hello.s
```

In the normal course of events we can remain blissfully unaware that the preprocessor and assembler are working on our behalf in the background, and we simply compile the `.cc` file.

D.2.3 The linker

Linking is the final step towards producing an executable. The linker takes our object code (plus standard C++ library code and any other libraries we want to link against) and glues it all together to create an executable file. So, for example, we can take the `hello.o` file we produced above and create `hello.exe` by typing

```
g++ hello.o -o hello.exe
```

D.2.4 Warnings

One more useful option to mention before we move on is the `-Wall` option. It tells the compiler not to just check for syntax errors, but also provide warning messages about things which may cause problems in the future. To get an idea compile the `hello.cc` program one last time:

```
g++ -c -Wall hello.cc
```

It will compile as before, but this time we are warned about the unused variable `y`. It is always a good idea to compile using the `-Wall` flag, as frequently the warnings give a clue to trouble that may lie ahead. You should take heed of these warnings and take corrective action to eliminate them.

D.3 Summary

If we ignore most of the detailed explanation above we are left with two important steps in creating an executable program: compiling and linking. Both are done on the command line by invoking `g++`.

To compile we use the `-c` switch and pass `hello.cc` as an argument. We then invoke `g++` again with the `.o` file and specify the name of the output file with the `-o` switch. `g++` “knows” to invoke the linker.

D.4 Second steps - multiple source files

¹³ This feature of `gcc` allows it to be act as a front end to many languages such as C++, C, Pascal, Fortran etc whilst reusing the assembler backend in each case.

The “HelloWorld” example we have use above is deliberately trivial. Building it by invoking g++ from a command shell is tedious but manageable. Larger projects have to be treated differently. It is generally not practical to have the whole program in one source file. Software projects may contain 100’s of 1000’s of lines of code. Such projects typically take a long time to build. For example the mico distribution took several hours to compile on my lowly 300MHz Pentium. If this were all in one file the iterative process of writing, compiling and fixing code would be intolerable.

So we break up the code into logical chunks and compile each chunk separately. Each chunk of source code translates into a chunk of object code and we use the linker to glue all our object code together. Now when we need to edit a particular chunk of source code we only need to recompile the source code which has changed. Sometimes figuring out which pieces have to be recompiled in response to changes is a non-trivial problem. Fortunately there are build tools available to simplify and automate this process. One such tool, GNU make, will be discussed in the next section. In this section we will continue going through the process of compiling manually, but you can rest assured that help is on the way.

Consider a program consisting of a main() function and two classes, A and B. We can structure such a program as follows:

```
=====
// File: A.h
#ifndef A_H
#define A_H
class A
{
public:
    A();
    ~A();
};
#endif
=====
// File: A.cc
#include <iostream>
#include ``A.h``
A::A()    { std::cout << ``A constructor`` << std::endl; }
A::~A()   { std::cout << ``A destructor`` << std::endl; }
=====
// File: B.h
#ifndef B_H
#define B_H
class B
{
public:
    B();
    ~B();
};
#endif
=====
// File: B.cc
#include <iostream>
#include ``B.h``
```

```
B::B()    { std::cout << ``B constructor`` << std::endl; }
B::~~B()  { std::cout << ``B destructor`` << std::endl; }
```

Assume we have another file `main.cc`, which makes use of the classes A and B. To build our executable we would have to go through the following steps:

Compile `A.cc`:

```
g++ -c A.cc
```

Compile `B.c`:

```
g++ -c B.cc
```

Compile `main.cc`:

```
g++ -c main.cc
```

Link the object files:

```
g++ A.o B.o main.o -o myprog.exe
```

It is worth noting a few points about this process. Firstly, it is a lot of typing. We would like to have a way to automate this process. Creating a simple batch file would help, but we can do better. It should not generally be necessary to repeat all of the steps each time we rebuild. For example, if I make a change to `A.cc` I shouldn't need to recompile `B.cc` and `main.cc`. I would just want to recompile `A.cc` and run the link step again. Life is made slightly more complicated by the include files. If I make a change to `A.h`, what needs to be recompiled? We have to recompile all `.cc` files which include `A.h`. In this case, `A.cc` and `main.cc`. `A.h` is said to be a dependency of both `main.o` and `A.o`. In other words if `A.h` changes, `main.o` and `A.o` will have to be rebuilt.

What the above discussion illustrates is the need for a tool which we can tell both how to build our programs, and also when. Such a tool is known as a build tool and is indispensable to managing the build process. Many IDE's provide a graphical interface to configuring the build process (the how and the when descriptions) by creating a project of some sort. We won't enter the debate over the merits of proprietary project formats here. In order to remain independent we will use a separate build tool.

There are a number of build tools available. Microsoft uses something called `nmake`. The Java community is fond of a thing called Ant (which uses XML to describe the build process and dependencies). Borland used to use a tool called `tmake`, but more recently just call it `make`. This caused many problems for students in the past, because we also use a tool called `make` which is incompatible with the Borland `make`. So if you had Delphi or C++ Builder installed, you would get strange errors when you invoked `make`, because the wrong one would be called. To avoid such confusion, from 2004 we have renamed our `make` tool to `gmake`.

D.5 The structure of a Makefile

Your Makefile will normally be called Makefile. It will generally consist of a section where you define some variables followed by descriptions of the targets and sub targets to be built. These descriptions are known as rules and have a simple format (usually two lines):

```
<target>: <list of files target depends on>
<tab><command required to build the target>
```

So, for example, we can describe the rule for linking our final executable in the sample session above as follows. Note the <tab> is actually a tab character.

```
myprog.exe: main.o A.o B.o
<tab>g++ main.o A.o B.o -o myprog.exe
```

This would indicate that myprog.exe is dependent upon main.o, A.o and B.o. If any of these changes, myprog.exe has to be recreated. The way to recreate it is to run “g++ main.o A.o B.o -o myprog.exe”. Done. Note that the space on the second line preceding g++ is a tab character.

To complete the Makefile we must fill in the rules for making main.o, A.o and B.o. Fortunately again, we can make use of some intelligence(!) built into gmake. The rules to build A.o and B.o would both take the form of:

```
A.o: A.cc A.h
<tab>g++ A.cc -o A.o
```

This is such a common pattern that we don’t even need to specify it. gmake has a built in (or implicit) rule which will suffice for both A.o and B.o. We do need to add something for main.o. Outside of the implicit rule gmake already understands about building a.o file from a.cc file, we need to tell it that it really has to rebuild main.o if there has been any changes to A.h or B.h. We can tell this to gmake by adding the following:

```
main.o: A.h B.h
```

Notice that we don’t need to specify how to build main.o. The implicit rule already takes care of that. We only need to specify when to build it. The complete (slightly enhanced) Makefile would like the following:

```
# Simple Makefile - note those spaces before the rule description
# are actually tabs
# specify to use g++ as the linker
LD = g++
# specify to use g++ as the compiler
CXX = g++
# pass some flags to the compiler
CPPFLAGS = -Wall

myprog.exe: main.o A.o B.o
    $(LD) main.o A.o B.o -o myprog.exe

main.o: A.h B.h
```

```
# a special target to clean up after we're done
clean:
    rm main.o A.o B.o myprog.exe *~
```

You will have noticed the enhancements. I specified a couple of variables at the top, specifically what C++ compiler and linker to use and some flags to pass to the compiler. The variable `CPPFLAGS` is a special variable that `gmake` understands it has to pass on to the C++ compiler. There are a few others we will meet in the next section. Defining these things as variables is generally a good idea. More about the `clean:` target below.

When you invoke `gmake` it first of all looks for a file in the current directory called `Makefile` and starts to process the rules it finds in there.¹⁴ It starts at the top and looks for the first rule and processes that. So in this case when we type `gmake`, `gmake` will find the rule for `myprog.exe` and start to build `myprog.exe` according to the rules we have created. Often we will want to build more than one program. This is typical of the exercises you will be doing where you have to build both `server.exe` and `client.exe`. You can tell `gmake` to this by specifying the first target in your `Makefile` similar to the following:

```
all: server.exe client.exe
```

Now `gmake` will find `all` as the first target, see that it depends on `server.exe` and `client.exe` and go on to search through the `Makefile` for further rules to build `server.exe` and `client.exe`. We will see examples of this in the following section.

You can also direct `gmake` not to build the first target it finds, but rather one which you specify on the command line. Our `clean:` target is a good example of this. When we want to clean up, i.e. remove all the old `.o` files and `.exe` files, we can just invoke `gmake` as follows:

```
gmake clean
```

`gmake` will scan the `Makefile` for a target called `clean`, and perform the associated actions. The actions in this case are simply to delete the rebuild-able files. We have used the `rm` command, rather than the DOS `del` command because it accepts a list of arguments. Persuading `del` to do the same thing is difficult. `rm.exe` is included in your `mingw/bin` directory so you can use these examples as is. The `*~` refers to the backup files my editor leaves littered around the working directory (e.g. `main.cc~`) etc. My editor is NTemacs. If your editor makes a similar mess, you can modify the `clean` target to help clean it up.

That's about all you need to know about `gmake` for now. There are a large number of additional features which we have conveniently ignored here. If you really want to become a `gmake` wizard you should download the manual from <http://www.gnu.org/>. The first two sections cover the basics.

¹⁴ It is possible to have a `Makefile` called something other than `Makefile`. If you do, you have to tell `gmake` which `Makefile` it is to use by invoking it like this: `gmake -f myMakefile`.

It is very rare that you will write a program which doesn't make use of some other precompiled library code. CORBA is no exception. We don't have to write our own ORB. We use one like Mico (or Orbacus). To do this we have to have a way to link against the precompiled code that is (typically) in the shared libraries (or DLLs).

To start off with, let us consider a simple example which uses the CORBA implementation in `mico2311.dll`. The example below actually illustrates the use of CORBA string functions and the `String_var` variable. Understanding memory allocation and deallocation is very important. This particular topic is very well covered in both the mico manual as well as the training workbook from Orbacus. We won't try and repeat an explanation here but suggest you refer to one or both of those documents. Our purpose here is to figure out how to compile the thing . . .

```
#!/ types.cc
/*
 * A demo program illustrating use of
 * some elements defined in the mico dll.
 */
#include <CORBA.h>
#include <iostream>
int main()
{
    // allocate space to hold a string
    char *str = CORBA::string_alloc(6);
    strcpy(str, "Hello");
    std::cout << "The string: " << str << std::endl;

    // deallocate the space again
    CORBA::string_free(str);
    char* str2 = CORBA::string_dup("Hello");
    std::cout << "The string: " << str2 << std::endl;

    // deallocate the space again
    CORBA::string_free(str2);

    // allocate and copy in one step into a String_var
    CORBA::String_var str3 = CORBA::string_dup("Hello");
    std::cout << "The string: " << str3 << std::endl;

    // don't have to worry about string_free. All CORBA _var
    // types free themselves in the destructor.
    return 0;
}
```

The first point to note is that we included the file `CORBA.h`. So the first thing we will have to tell the compiler (actually the preprocessor but we won't get pedantic) is how to find this file in order to include it. The relevant option for `gcc` is `-I`. This option is used to tell the preprocessor which directories to add to its standard search path when it encounters a `#include <nnn.h>`.

So we might compile with

```
g++ -c -Wall -Ic:/ unisa/ mico/ include types.cc
```

15 Unfortunately, it is quite common in University assignments. Students are often left with the feeling that C++ is not very useful because it is so difficult to actually do anything with C++ beyond "hello world". In reality it is extremely powerful because we can link against all sorts of libraries to easily tap into extra functionality.

This will produce for us the required object file, `types.o`. In order to build our executable we need to invoke the linker. As an experiment you can try to just run:

```
g++ types.o -o types.exe
```

You will be greeted with a list of linker errors, including:

```
types.o(.text+0x28a):types.cc: undefined reference to
`CORBA::string_alloc(unsigned long)'  
types.o(.text+0x2e3):types.cc: undefined reference to
`CORBA::string_free(char*)'  
types.o(.text+0x2f3):types.cc: undefined reference to
`CORBA::string_dup(char const*)'
```

Now all this should have been expected. Our program makes use of `string_alloc` et al but the linker has no idea where to find them. Fortunately, `gcc` makes this quite easy. All we have to do is include the `dll` in the list of object files to be linked.¹⁶

```
g++ types.o c:/ unisa/ mico/ win32-bin/ mico2310.dll -o types.exe
```

This should now produce the required executable. Not too painful, but quite a lot of boring (and error-prone) typing. Imagine that you kept getting compiler errors, which you then fixed and then had to recompile. Typing those two lines over and over would be intolerable. Worse still, imagine if `types.cc` was part of an assignment that you were sending to the lecturer. The lecturer sees the file and (if he is lucky) another file which reads: Dear Lecturer. If you are reading this it means that you have received my assignment. In order to compile please type the following . . . One unhappy lecturer!

This is where our Makefile comes into its own. What we really want to hear is: to compile the following please just type `gmake`. To finish off this section we present a sample Makefile to compile the `types.cc` example.

```
# set the root of your MICO installation here  
MICO=c:/unisa/mico  
# some compiler and linker flags - generally you can leave these as  
# is.  
LD=g++  
CXXFLAGS = -D_REENTRANT -Wall -Wno-deprecated -mthreads -  
I$(MICO)/include  
LDFLAGS = -mthreads  
LDLIBS = $(MICO)/win32-bin/mico2310.dll  
  
# Application specific stuff follows ... we want to build types.exe  
all: types.exe  
# types.exe is dependent upon types.o
```

¹⁶ This is a relatively new feature of the `gcc` linker. Previously one would have to link against something called an export library which contains info on the symbols exported from the `dll`. Now we simply have to specify the `dll`.

```
types.exe: types.o
    $(LD) $(LDFLAGS) types.o $(LDLIBS) -o types.exe

# useful to have a way to clean up with "gmake clean"
clean:
    rm -f types.o types.exe *~
```

The REENTRANT flags and the `-mthreads` switch were used to compile the `mico.dll` to support multiple threads. It is a good idea to pass the same flags to the compiler when you link your own code against the `dll`. This particular example would have been OK without them, but it is a good habit to get into now, to avoid unpredictable runtime errors in future projects.

Appendix E - Running a CORBA client and server

E.1 Introduction

This Appendix contains the tutorial letter, *Running a CORBA client and server*. After successful completion of Exercise 01, the students were encouraged to work through this tutorial before attempting Exercise 02.

This exercise involves running a server and a client program and observing the interaction between them. You are not required at this stage to code your own client and server programs. The client and server programs are one part of the set of demo programs which is distributed with Mico. The programs are to be found in the `demo/poa/account-1` directory under the directory where you have installed the Mico environment. We have already compiled the executables for you.

The first step is to open the IDL interface file, `account.idl`, in a text editor. You will see that it specifies two CORBA interfaces, a bank and account interface respectively. The server program is going to host objects which implement these interfaces and the client will connect to them and exercise their methods.

E.2 Listening sockets

To do this exercise you need to open three console windows (DOS boxes). Your current working directory in each case should be the directory of the demo program indicated above. An easy way to achieve this under Windows 2000/XP is to open one console window, change to the appropriate directory, and then open the other two windows by typing

```
start cmd
```

twice.

The new console windows inherit the entire environment from the parent window (including the current working directory). For convenience we will refer to these windows as the client, server and observation windows. In your observation window type:

```
netstat -a -p TCP
```

This command will show you the status of all your system's TCP sockets¹⁷. You are interested in the sockets indicated as being in the LISTENING state. When you start up the server according to the instructions in the next section, running `netstat` should show a new listening TCP socket.

E.3 Running the server

In your server window type: **server** If you get an error indicating that the application can't start because it is missing DLL files, then you have not set up your PATH environment variable properly. The `mico/bin` directory must be in your PATH in order for the system to find all the files it needs. Otherwise, if all went well,

17 Note that the `netstat` program is not available on Windows 98 or ME systems

you will just see the text "Running" displayed. You can kill the running server by pressing `Ctrl-C`. Restart it by typing `server` again.

E.4 The IOR file

Using the `netstat` command above, verify that your server has opened a new TCP listening socket and make a note of the port number it is listening on.

In order for the client to be able to connect to the server it will have to make a connection to this socket. If you start and stop your server a few times you will see that it listens on a new port each time. How will the client know where to connect to?

One of the things which your server did when it started up was to create a text file called "Bank.ref" (you can verify this by looking through the source code of `server.cc`). This file contains the IOR (see p146 of your text) of the bank object. If you open the IOR file in a text editor you will see that it is simply a long, cryptic, string of characters. Mico provides a convenient utility called `iordump` which interprets the IOR. To run it on the `Bank.ref` file type the following in your observation window:

```
iordump < Bank.ref
```

An important part of the information contained within the IOR is the internet address of the listening socket. You should be able to verify that this address is the same as the one indicated by `netstat`.

You can force the server to listen using a different port number by providing an additional parameter on the command line, for example:

```
server -ORBIIOPAddr inet:127.0.0.1:54321
```

Try this and check the resulting output of `netstat` and the contents of the IOR. You should see that the server is now listening on port 54321 of the local interface.

The `-ORBIIOPAddr` startup option is one of a number of options which can be passed to the CORBA orb on initialization. If you open the `server.cc` file you can see where this takes place on line 97:

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
```

The orb is initialized with the same parameters which are passed to the `main()` function on program entry. The `ORB_init()` function is designed to extract and process any command line options which were passed to the program on start-up. These usually begin with `-ORB.` and we will be encountering a few of them. The Mico documentation lists the full list of options recognized.

E.5 Summary

When we start the server it starts listening for client connections on a TCP listening socket. The internet address of that socket is encoded within the IOR file. We can specify the port to listen on by providing an extra command line parameter on start-up. If the client can open the IOR file it should be able to make a connection to the Bank object hosted on the server.

E.6 The client connects..

Make sure that a server is running in the server window. In your client window, type

```
client
```

The client should display the following output:

```
C:\unisa\mico\demo\poa\account-1>client
Balance is 250
Press return key to exit
```

Before exiting the client, take a look at the client source code in `client.cc`. At this stage you can ignore the intimidating details and try and understand the general picture. As predicted, the client gets a reference to the bank object on the server by reading in the IOR file. Having got this reference it can call the `create()` method on the remote bank object to create a new account. It then calls the `deposit()`, `withdraw()` and `balance()` methods on the remote account object.

While the client is still "connected" it is interesting to run `netstat` again in the observation window. This time you should see, in addition to the open LISTENING socket, a pair of TCP sockets in the ESTABLISHED state. This pair represents the connection between the client and server - one end is open on the server and the other end on the client. Once you close the client program you should see that these two sockets are no longer open and ESTABLISHED. Note that `netstat` on Windows XP simply shows the connection to have disappeared. Those of you familiar with TCP states will realize that the socket on one end of the connection does not close immediately, but actually lingers in a `TIME_WAIT` state for a period first. For our purposes it is sufficient to believe that the connection is gone at this point.

E.7 IIOP messages

What we have seen above is that the orb on the server opens a TCP listening port. In order for a client program to "talk" to a remote server object it establishes a TCP connection to the server orb. The discussion on pp 134-138 describes how the higher level GIOP/IIOP protocols define the messages which are actually passed backwards and forwards across the TCP connection. By using another startup parameter we can watch a trace of these messages.

To conduct this experiment, restart the server program like this:

```
server -ORBDebug=IIOP,Transport
```

Now when you run the client program you will see a trace of the messages on the server. The `-ORBDebug` option can be used on the client as well as the server. Tracing IIOP and Transport level communication is extremely most useful for debugging your own programs, so remember this option.

E.8 Testing the build environment

The programs in this assignment were prebuilt for you. In subsequent assignments you will have to write and compile your own. The last part of this assignment is for you to test that your build environment is properly set up. Open a console window and change directory to

```
c:\unisa\mico\demo\poa\account-1.
```

From here type `gmake clean` in order to remove the current executables. Now type `gmake` in order to compile the example code again. If you have a problem with this the chances are that the path to your compiler is probably not set correctly¹⁸.

We have modified some, but not all, of the makefiles which accompany the Mico demo examples. In particular you should be able to build the following:

- all the POA examples
- the services\naming example
- the services\naming-lb example
- the services\events example
- most of the test\idl examples (1 to 18)
- the test\poa examples

E.9 Wrapping up

That concludes your investigation of running a small CORBA application. We hope you found it interesting and look forward to your feedback via the discussion forum. However this module is about advanced programming so warm your compiler up, because your next assignment will involve you writing and compiling some actual code.

18 Also check the errata in Appendix A.2 of this guide

Appendix F - CORBA with Mico

F.1 Introduction

In Exercise 04, the NamingService is to be used. Before attempting this exercise, the students were encouraged to work through the tutorial *CORBA with Mico*. In the exercise, the currency convertor was refined so that the server registered the convertor object with a running nameserver, and the client resolved a reference to the convertor, by looking it up on the nameserver. Exceptions had to be caught where appropriate.

This section assumes that you have completed the set of practical exercises in the previous section. Having completed that assignment you will know that:

1. A CORBA application uses IDL to define the interface to a distributed application.
2. We can run a server program which provides an implementation of such an object (the concrete implementation is commonly known as the servant).
3. A client program can invoke methods on the server implementation of the object.
4. The object is uniquely identified by a stringified IOR which can be stored in a file by the server, and read by the client.
5. The underlying communication is achieved by exchanging IIOP messages over a TCP socket connection.
6. The building of the complete application (client and server) is facilitated using a build tool called `gmake`.

What we will do here is to examine the code from the Account example to see what is happening at each stage in the client and the server. We will also look at how `gmake` is used to build the application. Having worked through this section you should be in a position to write your own client and server, and build and test them.

F.2 Defining the interface

Before we can “distribute” an object we need to define an interface for it. This example presents an interface to Bank and Account objects. The interface is defined in CORBA idl as follows:

```
// File: account.idl
interface Account
{
    void deposit( in unsigned long amount );
    void withdraw( in unsigned long amount );
    long balance();
};

interface Bank
{
    Account create ();
};
```

The Bank interface represents an interface to a Bank object which has a method, `create()`, which should create a new Account object and return a reference to it. We are going to implement the client and server using

C++ and Mico so we need to generate the stub and skeleton classes using the Mico idl compiler. The command to do this is:

```
idl account.idl
```

This creates two files, `account.h` and `account.cc`. These files contain the client and server stub code that we need to implement the client and server parts of our application. Note that some implementations, such as Orbacus, produce client stubs and server skeletons in separate files. Mico, by default, just generates the `.cc` and `.h` files which are used by both sides¹⁹. If you look inside the `.h` file you will see that there are a number of class declarations which have been generated off the `idl`. The following is a selection of the most useful classes generated off the `Account` interface. There are equivalent classes declared for the `Bank` interface.

F.2.1 Account

This is the abstract base class for the stub and skeleton classes. You can see that it is abstract because of the pure virtual `Account::deposit()`, `Account::withdraw()` and `Account::balance()` methods. Because it is abstract, we cannot instantiate objects of type `Account`.

F.2.2 Account_stub

This is the stub class which is used on the client side to act as a proxy to the remote object. It is not an abstract class (it has no pure virtual methods) and implementations are provided for the pure virtual methods of the `Account` class. These implementations are concerned with marshalling and demarshalling messages to and from the remote object.

F.2.3 Account_ptr and Account_var

These are known as object reference types. All dealings with `Account` object references are done using one of these types. The `Account_var` type is a smart pointer²⁰ around the `Account_ptr` type. It is provided as a convenience to avoid error prone dealings with C++ pointers where possible. The `Account_ptr` type is typically implemented as a simple C++ pointer. You will find good descriptions of the `_ptr` and `_var` types in Unit 6 of the Orbacus tutorial (Client-Side C++ Mapping) as well as section 1.3 (Mapping for Interfaces) of the OMG C++ Language Mapping Specification.

19 You can ask the Mico idl compiler to generate separate server skeleton files by specifying `-c++skel` as commandline option to the `idl` command. The effect of this is to make the client code slightly smaller, which is not a major concern of ours here.

20 A smart pointer is a class which encapsulates a pointer, and uses operator overloading as well as constructor and destructor semantics to manage the memory allocation for the object being pointed at.

F2.4 POA_Account

The server skeleton class. Again this class is an abstract class. This is the class your interface implementation must inherit from.

F.3 Implementing the server side

Now we have the interface to our objects sorted out we need to consider implementing them. The example in the Mico distribution puts all the server implementation into one file, but we are going to break it into three, reflecting the different parts of our server code. First there is the implementation of the two interfaces, which we will put in `Bank_impl.cc` and `Bank_impl.h` and `Account_impl.h` and `Account_impl.cc`. In CORBA parlance, these classes are used to instantiate servants to which the POA can dispatch requests from clients. Then we will create a separate file, `server.cc`, which contains the `main()` function of the program.

Partitioning your application into separate files like this is a much more scalable way of building programs, rather than trying to implement everything in one file. Compiling and linking all the separate parts is a little more complicated, but the use of the build tool, `gmake`, together with a Makefile simplifies the task. We will look at a Makefile suitable for building our application when we finish walking through the code below.

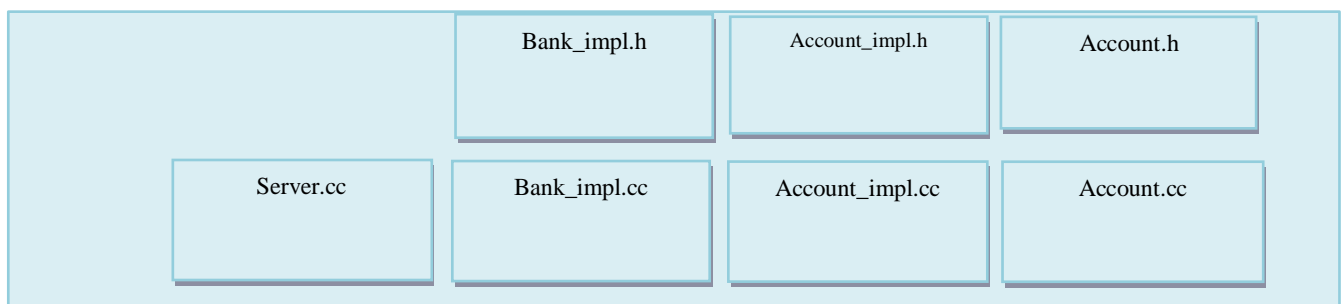


Figure F-1. C++ files used to compile the server

F.3.1 Implementing the interfaces - the servants

The `Account_impl.h` and `Bank_impl.h` files are shown below. Notice how they simply define classes which fill in the virtual methods required for the interface. First we show the `Account_impl` class:

```
// File: Account_impl.h
#ifndef ACCOUNT_IMPL_H
#define ACCOUNT_IMPL_H
#include "account.h"
class Account_impl : virtual public POA_Account
{
public:
    Account_impl ();
    void deposit (CORBA::ULong);
    void withdraw (CORBA::ULong);
    CORBA::Long balance ();

private:
```

```

CORBA::Long bal;
};
#endif

```

There is not much exciting here. Notice how `Account_impl` derives from the `POA_Account` class which was produced by the idl compiler. This allows objects of type `Account_impl` to be activated and served up by a CORBA Portable Object Adaptor which we will see shortly.

The bank declaration is even simpler:

```

// File: Bank_impl.h
#ifndef BANK_IMPL_H
#define BANK_IMPL_H
class Bank_impl : virtual public POA_Bank
{
public:
    Account_ptr create ();
};
#endif

```

We just have one method to implement, which will instantiate an object of type `Account` and register it with the POA. All that is required of the `Account` implementation is for it to provide the logic for implementing the methods. Our implementation is shown below:

```

#include "Account_impl.h"
Account_impl::Account_impl ()
{
    bal = 0;
}
void Account_impl::deposit (CORBA::ULong amount)
{
    bal += amount;
}
void Account_impl::withdraw (CORBA::ULong amount)
{
    bal -= amount;
}
CORBA::Long Account_impl::balance ()
{
    return bal;
}

```

The implementation of the `Bank_impl` class is shown below.

```

// File Bank_impl.cc
#include "Bank_impl.h"

Account_ptr
Bank_impl::create ()
{
    // Create a new account (which is never deleted)
    Account_impl * ai = new Account_impl;

    // Obtain a reference using _this. This implicitly
    // activates the account servant (the RootPOA,

```

```
        // which is the object's _default_POA, has the
        // IMPLICIT_ACTIVATION policy)
        Account_ptr aref = ai->_this ();
        assert (!CORBA::is_nil (aref));

        // Return the reference
        return aref;
    }
}
```

This is exactly as is implemented in the Mico example code. There are a few interesting points to note.

The creation of the new `Account_impl` is what we would expect. This will instantiate a new `Account` for which we must return a pointer reference to.

Reference to the `Account's _this()` method is a slightly sneaky way to activate it. Because the `Root POA` (see below) has `IMPLICIT_ACTIVATION` policy, the object will get activated simply by calling one of its methods. It has the advantage that we don't need to have a reference to the `POA` in order to activate.

F.3.2 Serving it all up with the POA

The last part of the server we need to consider is the main function which actually uses the `POA` to serve up the `Bank` object. The following is taken from the Mico example:

```
#include "account.h"
#include "Bank_impl.h"
#include <fstream>
using namespace std;

int main (int argc, char *argv[])
{
    // Initialize the ORB
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Obtain a reference to the RootPOA and its Manager
    CORBA::Object_var poaobj = orb->resolve_initial_references
        ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow
        (poaobj);
    PortableServer::POAManager_var mgr = poa->the_POAManager();

    // Create a Bank
    Bank_impl * micocash = new Bank_impl;

    // Activate the Bank
    PortableServer::ObjectId_var oid = poa->activate_object
        (micocash);

    // Write reference to file
    ofstream of ("Bank.ref");
    CORBA::Object_var ref = poa->id_to_reference (oid.in());
    CORBA::String_var str = orb->object_to_string (ref.in());
    of << str.in() << endl;
    of.close ();

    // Activate the POA and start serving requests
}
```

```
cout << "Running." << endl;
mgr->activate ();
orb->run();

// Shutdown (never reached)
poa->destroy (TRUE, TRUE);
delete micocash;
return 0;
}
```

There are lots of interesting things going on here, which we will not explain in detail. We will however, make a brief comment on some parts of the code. First of all note the call to `CORBA::ORB_init(argc, argv)`. This is necessary for all programs (client or server) which are going to use the orb. Note also how it passes the parameters from the main function (ie. the parameters passed on the command line when the program was invoked). This is a frequently used mechanism for sending initial references and parameters to the orb at start up time. We will not be using this feature here, but you will see how it used later to pass information such as the location of a CORBA NameService.

Note also how the `Bank_impl` object is instantiated and explicitly activated in contrast to the implicit activation of the `Account_impl` object above. The code which resolves a reference to the `RootPOA` and activates it is explained in a little more detail later. For the moment you can simply think of it as “boilerplate” code.

Having instantiated a `Bank_impl` object the server writes out its stringified reference to a file, `Bank.ref`. The content of the file (from a run on my desktop machine) looks like the following:

```
IOR:010000000d00000049444c3a42616e6b3a312e300000000020000000000000
03b000000010100001700000070632d323038303039352e756e6973612e61632e7a
610000cd080000130000002f333737322f313038303635323831362f5f300001000
00024000000010000000100000001000000140000001000000010001000000000
0901010000000000
```

A string of cryptically ordered characters. This string is simply a convenient form of object reference which can be stored in a file, sent by email, posted on a web page, memorized and smuggled across borders etc. There is a program called `iordump` which we can use to see what the file represents: `iordump < Bank.ref`

This should dump some more meaningful information to the console on what is inside the IOR string. On my machine, for example, I get:

```
Repo Id: IDL:Bank:1.0
IIOP Profile
Version: 1.0
Address: inet:pc-2080095.unisa.ac.za:2253
Location: corbaloc::pc-2080095.unisa.ac.za:2253//3772/1080652816/%5f0
Key: 2f 33 37 37 32 2f 31 30 38 30 36 35 32 38 31 36 /3772/1080652816
2f 5f 30 /_0
Multiple Components Profile
Components: Native Codesets:
normal: ISO 8859-1:1987; Latin Alphabet No. 1
```

```
wide: ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit
form
Key: (empty)
```

F.4 The client

The last step is to create a simple client program which reads the object reference from a file connects to the server object and invokes methods on it. The following example from the Mico distribution does exactly that. It makes use of a useful feature of the Mico orb which allows us to pass a reference in the form of a URI. In other words, instead of going through the steps of the process to open the file `Bank.ref`, read the IOR string out of it, close the file and then call `string_to_object()` on the IOR string, we can simply call `string_to_object()` on the file URI.

We have enhanced the basic example here to show how the client can catch the system exceptions which might be thrown when things go wrong.

```
#include "account.h"
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#ifdef _WIN32
#include <direct.h>
#endif
using namespace std;
int main (int argc, char *argv[])
{
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    //IOR is in Bank.ref in the local directory
    char pwd[256], uri[300];
    sprintf (uri, "file://%s/Bank.ref", getcwd(pwd, 256));

    // Locate the Bank
    Bank_var bank;
    try
    {
        CORBA::Object_var obj = orb->string_to_object (uri);
        bank = Bank::_narrow (obj);
    }
    catch (CORBA::Exception &ex)
    {
        cerr << "Problem locating bank: " << ex << endl;
        cerr << "Perhaps the file doesn't exist?" << endl;
        exit (1);
    }
    try
    {
        // Open an account
        Account_var account = bank->create ();
        if (CORBA::is_nil (account))
            { cout << "oops: account is nil" << endl; exit (1); }

        // Deposit and withdraw some money
        account->deposit (700); account->withdraw (450);
        cout << "Balance is " << account->balance() << endl;
    }
}
```

```
    catch (CORBA::Exception &ex)
    {
        cerr << "Problem invoking object methods: " << ex << endl;
        exit (1);
    }

    return 0;
}
```

For more information about CORBA exceptions, you should consult sections 6.29 to 6.32 and sections 9.10 and 9.11 of the Orbacus training guide. You will also find useful information in section 4.12 of the OMG CORBA Specification and section 1.19 of the OMGC++ Language Mapping Specification.

F.4 Final word - the Makefile

We still need to compile all the sources to produce the client and server executables. There are quite a lot of steps involved, which is why we make use of a build tool (gmake) to automate the process. gmake reads its instructions from a file (usually called Makefile), which you can think of as a recipe of instructions on how, when and what to build. The example below is a Makefile suitable for building the example we have discussed in this section:

```
# Makefile for account example
# Adjust the next line to match your installation
MICO=c:/unisa/mico

# Set some variables. You shouldn't need to change these.
IDL=$(MICO)/bin/idl
LD=g++
CXXFLAGS = -D_REENTRANT -DHAVE_ANSI_CPLUSPLUS_HEADERS -Wall -
mthreads -I$(MICO)/include
LDFLAGS = -mthreads
LDLIBS = $(MICO)/bin/mico2311.dll -lpthread -lwsock32

# these are the object files needed for the server and client
SERVEROBS = server.o account.o Bank_impl.o Account_impl.o
CLIENTOBS = client.o account.o
# Tell gmake we want to build server.exe and client.exe
all: server.exe client.exe

# Rules for making the server and client
server.exe: $(SERVEROBS)
    $(LD) $(LDFLAGS) $(SERVEROBS) $(LDLIBS) -o server.exe

client.exe: $(CLIENTOBS)
    $(LD) $(LDFLAGS) $(CLIENTOBS) $(LDLIBS) -o client.exe

# List of dependencies
# eg. recompile account.o if either account.cc or account.h
changes
#
# Hint: these rules are generated from running ``g++ -w -MM *.cc``
account.o: account.cc account.h
Account_impl.o: Account_impl.cc Account_impl.h account.h
Bank_impl.o: Bank_impl.cc Account_impl.h account.h Bank_impl.h
client.o: client.cc account.h
server.o: server.cc account.h Bank_impl.h
```

```
# rule for processing idl file
account.h account.cc : account.idl
    $(IDL) account.idl

clean:
    rm -f account.cc account.h Bank.ref *.o *.exe *~
```

We will not try and explain all the details of the Makefile here, but offer it as a template you can use for your own programs. If you are adapting this file to build your own programs:

- You will not need to change anything in the first section of the Makefile;
- The SERVEROBS and CLIENTOBS may well be different from one program to another;
- The list of dependencies will vary, but you can generate them automatically by running gcc in the same directory as your source files, with the options shown in the Makefile.
- The rule for processing the idl file will simply change to reflect your file names.
- Important note: all the indented lines MUST be indented with a tab, not spaces. gmake specifically looks for the tab character to process the rules.

More information on gmake and the mechanics of building programs is available in Appendix C and D. The complete manual for gmake is also on your CDrom.

Appendix G - More CORBA with Mico

G.1 Introduction

In this section we revisit the simple account example and examine the components of the client and server parts of that application in more detail. The aim is not to be exhaustively thorough, but rather to cover sufficient detail to assist you to get started writing programs. We have not attempted to describe the IDL to C++ language bindings in this guide. We suggest you consult Chapter 5 of the Mico manual for a brief overview and Chapter 5 of the Orbacus training workbook for a more complete view. Note that these language bindings are not particular to Mico or Orbacus. For a definitive reference you should refer to the OMG standard.

Be sure to read Chapter 3 of your text book (especially section 3.4) to establish a basic conceptual framework. Programming CORBA clients and servers requires some knowledge of the orb and object adaptor interfaces. We discuss the essential aspects of these interfaces in this guide. Whereas in the introductory example we passed an object reference from the server to the client by placing the stringified object reference into a file, we will now look at more general solutions, culminating in a description of the CORBA Naming Service.

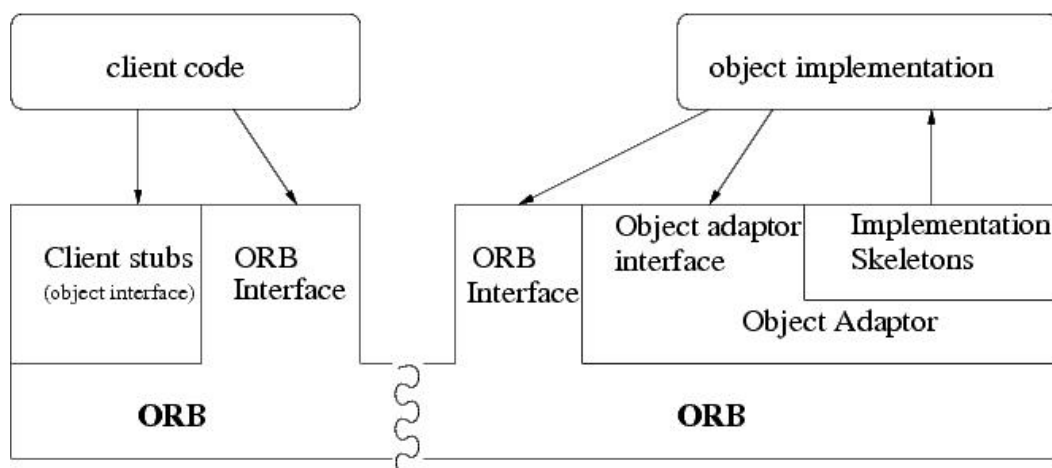


Figure G-1. Architecture of a simple application

Figure G-1 shows a simplified architectural view of a CORBA application using compiled client stubs and server skeletons. From the diagram, it should be apparent that if your process is to be able to act as a client to a CORBA object it will need to have access to the ORB interface as well as to the client stub of the object. Notice that, although it is common to see the ORB depicted as a single object, spanning both client and server, from an implementation point of view client and server processes access their orb implementations independently. It is more useful to think of “the ORB” as a single communications layer. Different programs accessing this layer see it through the interface exposed by their local orb implementation. Some commonly used aspects of this interface are described below.

The server code also makes use of some direct calls to the orb interface. But, from the programmer’s perspective, the major task is to provide an implementation of the object interface i.e. provide code which implements the

methods which the client can invoke and which are defined in the IDL description. Objects which implement such interfaces are commonly known as servants. An important component of a CORBA server is an Object Adaptor. The object adaptor provides the bridge between the ORB and the servant implementations. The CORBA standard envisages that there could be a variety of different types of object adaptor, but the minimum specification states that all implementations should support the POA (Portable Object Adaptor). Earlier versions of the standard specified the use of a simpler adaptor known as a BOA (or Basic Object Adaptor), but its use is now deprecated.

G.2 The ORB interface

There are essentially three kinds of operations you need to carry out using the orb interface. You need to know how to initialize it, how to use its services for converting between object references and stringified forms and you need to be able to tell it to process events i.e. run an event loop for server objects. These three categories are discussed briefly below.

G.3 ORB initialisation

The first encounter with the orb is usually made when the `CORBA::ORB_var` variable is declared and initialised. You will recall that both the client and server code from our account example initialize the orb early on in the `main()` function. This is done with the line:

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
```

Most ORBs support a variety of standard as well as implementation specific initialization parameters. All implementations allow parameters to be set at the command line (hence the passing of `argc` and `argv` to `ORB_init()`) as well providing implementation specific means, such as initialization files or registry keys. Because the ORB is responsible for making the transport layer (usually TCP) transparent to the client and server code as well as providing location transparency, many of the ORB initialization parameters relate to configuring the underlying TCP sockets layer and priming the orb with initial references. The following list should be sufficient for most purposes:

G.4 -ORBIIOPAddr

This is a Mico specific option which specifies which server address the orb should bind to when listening for incoming requests. This option takes a single argument in the form of `inet:hostname:port`. In our account example when we started the server program we didn't care which port it would listen on. The orb simply chose the next free available port. If we needed to specify that it listen on port 5555, for example, we could launch the program with:

```
server.exe -ORBIIOPAddr=inet:starship:5555  
or  
server.exe -ORBIIOPAddr=inet:192.168.1.1:5555
```

This parameter is passed into your `main()` function as part of the `argv[]` array which is then passed on to the orb when your program calls `ORB_init()`. Note that `starship` is the name of my computer at home. It has

an Ethernet card which is configured with IP address 192.168.1.1. If your computer is networked you can find out the IP address with ipconfig (on Windows 2000/XP). If you are not on a network you can use the loopback interface address of 127.0.0.1.

This option is only really useful for server code where the orb is required to listen for incoming connections on a particular well known port. In general it is better not to have to specify the listening address like this. Wherever possible it is better to use CORBA services (such as the NameService discussed later in this document) to locate objects than to manually configure TCP/IP socket addresses, but there are cases where you need to. One case might be if your firewall was configured to allow IIOP traffic through on this address only. Another common use is to force a CORBA name server to listen on a fixed, well known address. It is all very well to depend on the NameService to locate objects, but ultimately we still need a way to locate the NameService itself.

G.5 -ORBInitRef

This is a standard option which should be supported by all orb implementations. It is most commonly used to provide the orb with an initial reference to a CORBA NameService. Even though Mico has an implementation specific option for passing the location of the NameService (-ORBNamingAddr), it is preferable to use -ORBInitRef. An example of its use is as follows:

```
myprog.exe -ORBInitRef NameService=<string>
```

where <string> is any string form which is recognized by the orb string_to_object() method described in section 5.1.3 below.

G.6 -ORBDebug

Another Mico specific option which causes the orb to dump trace information on the standard error stream. It is very instructive to do this and watch the IIOP messages being passed between clients and servers. There are a number of debug levels. Use -ORBDebug=All for the most verbose output, or -ORBDebug=IIOP if you just want to see the IIOP messages.

G.7 Resolving initial references

Even though a good part of the orb's functionality is geared towards providing location transparency, it still needs to bootstrap itself with some initial references. There are a number of useful initial references which can be configured, but the following two are almost essential to any application:

G.7.1 NameService

If there is a CORBA NameService running, all of our client and server programs will need to be able to locate it. We can provide the orb with the initial reference in the manner described above. The user program can then call

```
orb->resolve_initial_reference('NameService');
```

in order to get a CORBA reference with which to interact with the NameService. The example in section 6 below illustrates the use of the NameService.

G.7.2 RootPOA

We have seen that a server needs an object adaptor to be able to dispatch incoming requests from the orb to the appropriate servants. Orb implementations must minimally provide a default POA, called the RootPOA. We can access this POA by calling

```
orb->resolve_initial_reference('`RootPOA`');
```

on the orb. Having got a reference to the RootPOA we can create child POAs using the methods in the RootPOA's interface. The POA interface is discussed further below.

G.8 Converting between object references and strings

The ORB keeps track of CORBA objects by making use of an object reference. We saw in the account example of part 1 of the practical guide that the orb provides a method, `object_to_string()`, which can be used to create an ASCII string representation of the object reference.

So for example, the following snippet of server code creates a servant of type `Account_impl`, which is activated and converted to a string:

```
// instantiate servant
Account_impl acc_servant = new Account_impl;
// activate with current POA and return reference
Account_ptr acc_ref = acc_servant->_this();
// convert the object reference to a string
CORBA::String_var acc_strref = orb->object_to_string(acc_ref);
```

We saw earlier that this string is of the form "IOR:4545435...". It is in a convenient form to be saved in a text file and passed to the client program.

Clients in turn make use of the orb's `string_to_object()` method to extract the object reference from the string. Unfortunately, besides being in a portable ASCII text format, the IOR string is far from convenient from a human perspective. Fortunately, recent implementations of the CORBA standard support a number of enhancements to the `string_to_object()` method which are much more convenient. Given that the IOR string is likely going to be posted into a file, it is sufficient to be able to specify the URL of that file rather than the file itself. The Mico orb understands URLs which have any of the following prefixes:

```
"file://..", "http://..", "ftp://..", "corbaloc:..." and
"corbaname:...".
```

Thus if your server program saved a stringified object reference in a file called "C:/iors/account.ref", a client program can get an object reference with:

```
CORBA::Object obj = orb->string_to_object
    (`file://C:/iors/account.ref`);
Account_var acc = Account::_narrow (obj);
```

Accessing a file: // URL is of limited use in a Distributed System (though the file could be stored on a central network share). Using the http form of the URL, on the other hand, is quite a convenient way of sharing stringified object references. The corbaloc and corbaname URLs provide a way of accessing known CORBA services without the use of the stringified object reference.

We will look briefly at the use of `corbaloc` and `corbaname` in the section on the `NameService` below.

G.9 Processing incoming events

This last section in our discussion of the orb interface deals with the set of functions required to make our orb process incoming requests. An important of the functionality of any middleware is concerned with hiding the details of the underlying transport API and protocols. Without a middleware platform, such as CORBA, we would have to implement our clients and servers using TCP/IP sockets. Writing robust and scalable sockets based servers from scratch is not a trivial exercise, so we make use of the orb services to abstract the details.

As far as client code calling methods on distributed objects is concerned there is no more to be done. Our client code simply invokes the method on the stub (transparently) and the orb takes care of the rest.

On the server side we need to tell our orb to open up listening sockets to wait for incoming connections and to process those events accordingly. The easiest way to do this is to simply call the `orb->run()` method. This is what you see in `server.cc` of the account example in part 1. An important consideration is that the orb takes over program execution from here. The orb will now process all incoming requests and route messages to objects via the object adaptor. For the simple exercises in this course this is all your server code need do.

It is possible to exert finer control over the orb's functionality than this. One common example would be to call the orb's `run()` method in a separate thread. This would leave the main thread free to perform other tasks such as process user interface events, shutdown and restart the orb etc. There are also methods in the orb interface which allows the orb event processing to be integrated into other event loops. Some examples are given in the Mico demos, but they are mainly UNIX based which is foreign to many of you, so we won't go there now From the perspective of COS311, it is sufficient for you to write your server code and then call `orb->run()` and let the orb take care of things from here.

One last point which should be made is that we have tended to draw a sharp distinction between clients and servers. In reality things are rarely this simple. Server code may well have to act as client to other code, and, more significantly, client code may also have to process incoming events. This is particularly the case when processing asynchronous requests as described on page 182 of your textbook. If your client also has to process incoming requests it has also to make use of the orb's event processing using one of the strategies outlined in the paragraph above.

That concludes our quick survey of the orb interface. We have found ourselves talking of processing incoming events and routing through the object adaptor, so now as an appropriate place to describe the object adaptor interface.

G.10 The POA interface

The ORB takes care of low level communication for us; servants provide the implementation code of our distributed server objects. What lies in between is the object adaptor. The portable object adaptor (or POA) is the most commonly deployed object adaptor so we will concentrate on the POA interface. A default POA, called the RootPOA, is associated with the orb. The implication of this is that we do not need to create the RootPOA - we just call `resolve_initial_references()` to get a reference to it. It is helpful to think of the POA as a sort of router, providing a mapping between the object references (which the orb uses) and the servants which are provided by the application programmer. To make things a little more complicated, there are actually two levels of mapping. The POA uses internally something called an object ID.

The POA manages the mapping between object references and object id's, as well as the mapping between object id's and servants. In the simplest case an object reference will map directly to a POA object ID which in turn will map to a servant. Having this two stage mapping seems overly complicated, but it does allow maximum flexibility in terms of scalability and resource management. When an object ID is mapped to a servant, the object is said to be activated. The servant is said to incarnate (provide a body for) the object. For a server managing many thousands of objects it might not make sense to have them all activated simultaneously, using memory and other resources. By making the object ID separate from the servant implementation, it is then possible to activate and deactivate servants on demand thus making optimal use of the available resources.

This relationship between references, id's and servants is illustrated in Figure G-2. The important (if over-simplified) points to remember are that:

1. Object references are used within the orb to identify objects. Clients need to acquire such an object reference in order to invoke methods on it via the orb.
2. The POA maintains an object ID which is mapped to an object reference. Clients are not aware of the existence of an object ID. It is purely internal to the POA functioning.
3. When the object is activated within the POA, the object ID is associated with the servant.

The topic of object lifecycle and implicit vs. explicit activation of objects is discussed in Chapter 9 of your text book. We limit ourselves here to the briefest description to be able to use the POA interface. Pages 50 to 55 of the Mico manual provide a fuller description. To maintain a practical perspective, let's take another look at the section of the server code from `server.cc` earlier which deals with activating the Bank object:

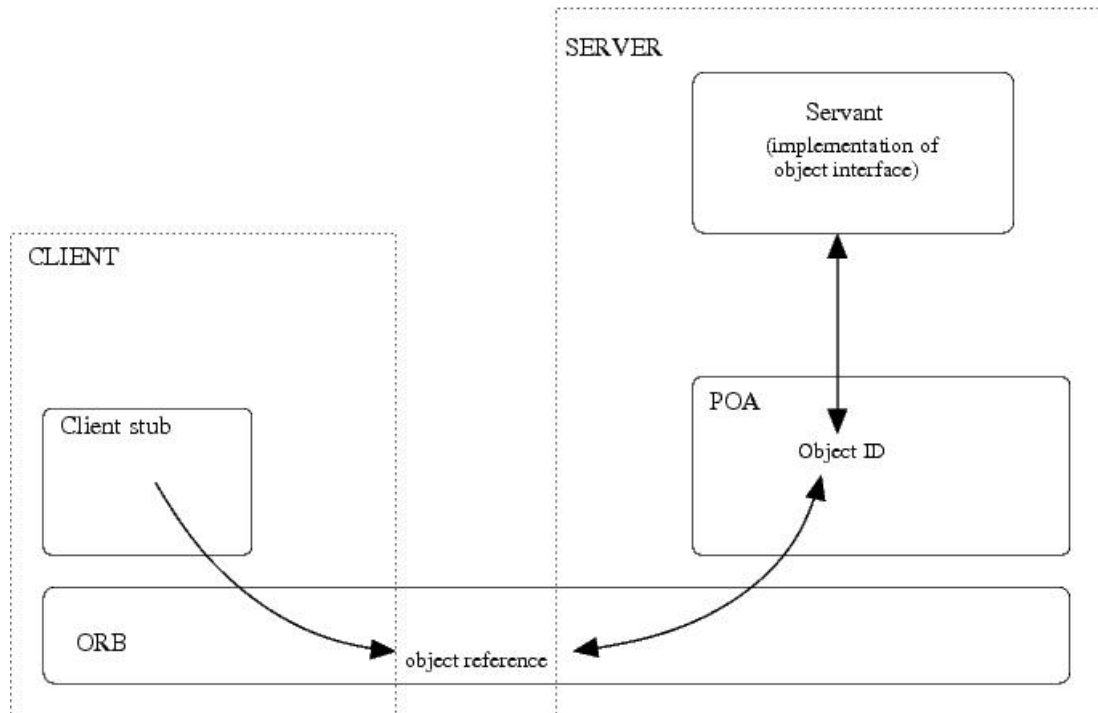


Figure G-2. Mapping of object reference to servant

```

// instantiate servant
Bank_impl * micocash = new Bank_impl;
// activate and return object id
PortableServer::ObjectId_var oid = poa->activate_object
(micocash);

// Write reference to file
ofstream of ("Bank.ref");
// the poa can tell us which object ref is mapped to which id
CORBA::Object_var ref = poa->id_to_reference (oid);
// the orb can do the conversion for us ...
CORBA::String_var str = orb->object_to_string (ref);
of << str.in() << endl;
of.close ();

```

This code takes a bit of a long road from the point of instantiating a new `Bank_impl` object and acquiring an object reference. We saw in the case of `Account_impl` that we can call the object's `_this()` method to implicitly activate the object and return a reference.

Alternatively, instead of calling the POA's `activate_object()` followed by `id_to_reference()`, an application developer can just call the POA's `servant_to_reference()` method. This also has the effect of implicitly activating the servant.

Using this approach, the above code would look like:

```

// instantiate servant
Bank_impl * micocash = new Bank_impl;
// implicitly activate the servant ..
CORBA::Object_var ref = poa->servant_to_reference (micocash);

```

```
// get stringified reference
bCORBA::String_var str = orb->object_to_string (ref);
```

In order to get our POA to actually start serving requests we have to make use of the POAManager which is associated with the POA. It is necessary to call activate() on the manager. The code which does this in our simple server application is shown below:

```
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate ();
```

G.11 Conclusion

That concludes the brief tour of the POA interface. The bare minimum to write a simple POA based server was covered. Important issues such as POA policies, creating hierarchies of POAs, concurrency strategy and servant managers have not been discussed. The interested student is referred to the Mico manual section on the POA (a reference to the appropriate location of the Mico manual) for a slightly fuller description, and Unit 12 of the Orbacus training work book for a more comprehensive overview.

Appendix H - The CORBA NameService

H.1 Introduction

Chapter 8 of your text book addresses the universal problem of locating distributed objects. In particular it describes the CORBA naming service in some detail. Please ensure that you have read pp 209-213 of chapter 8 before proceeding further. Note that because the CORBA naming service is specified in IDL, it should not make any difference which vendor's implementation of a name server is used - whether you use Mico, Orbacus or even the Java jacob, the service must comply with the standard interface and will be interoperable.

You can also refer to the OMG Standard specification document for the CORBA NameService. The Orbacus tutorial also has a section describing the use of the NameService.

The explanation below is based on the name server distributed with Mico, known simply as `nsd`. Note that when you compile code which makes use of the CORBA NameService, you will have to change the following two parameters in your Makefile:

```
LDFLAGS = -mthreads -L$(MICO)/libs
LDLIBS = $(MICO)/bin/mico2311.dll -lmicocoss2.3.11 -lpthread -lwssock32
```

This is because the C++ client stubs for the NameService are not part of the CORBA core in `mico2311.dll`, but are in a separate library file, `libmicocoss2.3.11.a`. The above parameters will ensure that this library code will be located when your application is built.

H.2 Starting the Mico NameService

The nameserver which comes with Mico is called `nsd`. It is a simple POA bases server which provides access to a `NamingContext`. In fact the Mico name service also supports the extended interface known as `NamingContextExt`. `NamingContextExt` is derived from `NamingContext` and provides some additional helper methods to perform conversions between Names and URL strings. We will see below how these additional methods make the task of manipulating CORBA names considerably less cumbersome.

Whereas the NameService provides location transparency for our distributed objects, an administrator (that's you in this context!) would have to ensure that all clients and servers can find this initial `NamingContext`. There are a number of implementation specific variations on this theme, but the following is a simple recipe:

Open a new console window (DOS box) and start `nsd` with the `-ORBIIOPAddr` command line option, for example: `nsd -ORBIIOPAddr inet:192.168.1.1:5555`

Note that you will have to provide the IP address of your own machine. If your machine has a network interface of its own it will have an IP address associated with that interface. On Windows XP/2000 you can find that

address by running `ipconfig`. If your machine is not networked, you can specify the loopback address of 127.0.0.1.

You can select any unused port number greater than 1024 to listen on. In this example `nsd` will listen on port 5555. `nsd`, by default, will operate quietly. If you start `nsd` and don't see any output this is good, not bad! Sometimes, it can be reassuring to see some sort of output, so you might want to pass additional parameters to tell the orb to dump debug information. For example, invoking `nsd` with:

```
nsd -ORBIIOPAddr inet:192.168.1.1:5555 -ORBDebug= IIOP,Transport
```

provides some useful feedback of which messages are being transferred to and from the server.

H.3 Using the NameService

In the example at the end of part 1, we wrote a simple server which created a `Bank` object and wrote a stringified reference to a file, `Bank.ref`. The client got the reference by reading the stringified IOR from the file. In this section, we will look at a modified version of the client, which makes use of a name server instead. The following two assumptions are made:

1. Your `Bank` server has been started and the file has been copied to `C:/Bank.ref`.
2. A `NameServer` has been started with the parameters given above.

We will then use the simple `NameService` tool which is distributed with `Mico`, `nsadmin`, to bind the `Bank` object reference to a name.

We need to have a way of telling `nsadmin` where to find the `NameService`. There is a `Mico` specific parameter, `-ORBNamingAddr`, which could be used for this, but we will use the `CORBA` standard `-ORBInitRef` parameter (as described in section 5.1.2 above) instead:

```
nsadmin -ORBInitRef NameService= corbaloc::192.168.1.1:5555/NameService
```

Assuming your parameters are all correct, `nsadmin` should connect and return a prompt like:

```
CosNaming administration tool for MICO(tm)nsadmin>
```

If you started `nsd` with the suggested debug parameters, you should also have seen some output in the `nsd` window. If you don't see any of this behaviour, please check the parameters used to start `nsd` and `nsadmin` carefully and try again.

As stated above, `nsadmin` is a simple administrative tool. It understands a limited set of commands. For a list of these, type 'help' at the `nsadmin` prompt.

We are going to use the 'bind' command to bind the reference to the `Bank` object to the name "Bank". To do this we will make use of the fact that the stringified IOR of the `Bank` object is contained in the file. Now, at the `nsadmin` prompt, we type:

```
nsadmin> bind Bank file:/// C:/ bank.ref
```

This should create a new object binding in the NamingContext. Having done this, you can try out some of the other commands understood by nsadmin: ls, cat, iordump, url etc.

H.4 Resolving a name in the NameService

A NamingContext maintains a list of bindings between object names and object references. The fundamental operations are `bind()` and `resolve()`. A client will contact a NameService and resolve a name within a given context to get an object reference.

The code to achieve this is relatively simple, though rendered a little clumsy by the fact that a name is a compound structure, composed as a sequence NameComponents, each having an id and kind part. The following snippet is taken from the Mico examples at `services/naming/client.cc21`.

```
// Acquire a reference to the Naming Service
CORBA::Object_var nsobj =
    orb->resolve_initial_references ("NameService");
CosNaming::NamingContext_var nc =
    CosNaming::NamingContext::_narrow (nsobj);
if (CORBA::is_nil (nc))
{
    cerr << "oops, I cannot access the Naming Service!" << endl;
    exit (1);
}

// Construct Naming Service name for our Bank
CosNaming::Name name;
name.length (1);
name[0].id = CORBA::string_dup ("Bank");
name[0].kind = CORBA::string_dup ("");

// try to find that node in the Naming Service tree
CORBA::Object_var obj;
cout << "Looking up Bank ... " << flush;
try
{
    obj = nc->resolve (name);
}
catch (CosNaming::NamingContext::NotFound &exc)
{
    cout << "NotFound exception." << endl;
    exit (1);
}
catch (CosNaming::NamingContext::CannotProceed &exc)
{
    cout << "CannotProceed exception." << endl;
    exit (1);
}
catch (CosNaming::NamingContext::InvalidName &exc)
```

21 Note that in order to build this code you will have to modify the makefile. You can probably just substitute the one used for the account-1 example.

```
{
    cout << "InvalidName exception." << endl;
    exit (1);
}
cout << "done." << endl;

// The Naming Service returns a generic reference as a
// CORBA::Object We need to narrow this to the desired type
Bank_var bank = Bank::_narrow (obj);
```

Note the use of an initial reference to the NameService. You have to supply this initial reference! As we did above with nsadmin, you start the client with:

```
client.exe -ORBInitRef NameService= corbaloc::192.168.1.1:5555/
NameService
```

Now you might argue that this is a lot of code to do a simple thing, and in a distance-based education environment it is. Through the use of the newer URLs understood by the orb it is possible to compress all of the above into:

```
try
{
    CORBA::Object_var obj = orb
        ->string_to_object("corbaname:rir:#Bank");
    Bank_var bank = Bank::_narrow (obj);
}
catch (CORBA::Exception &ex)
{
    cerr << ``Failed to resolve Bank`` << endl;
    exit(1);
}
```

The meaning of the corbaname url is as follows: corbaname indicates that the string is a corbaname url; rir indicates to use the nameservice that would be returned by resolve_initial_references(); and what follows the # symbol is the compound name represented as a simple string.

H.5 Binding an object to a name

We saw at the beginning of this discussion, how we can use nsadmin to bind an object to a name (or a name to an object?). Often we want the server to automatically register its object(s) with a nameservice. An example of how to do this is illustrated in the server code of the same Mico example referred to in the section above. This time we don't show the code used to resolve the initial reference to the NameService:

```
// Construct Naming Service name for our Bank
CosNaming::Name name;
name.length (1);
name[0].id = CORBA::string_dup ("Bank");
name[0].kind = CORBA::string_dup ("");

/* Store a reference to our Bank in the Naming Service. We use
'rebind' here instead of 'bind', because rebind does not
complain if the desired name "Bank" is already registered,
but silently overwrites it (the existing reference is
probably from an old incarnation of this server).
*/
cout << "Binding Bank in the Naming Service ... " << flush;
nc->rebind (name, ref);
```

```
cout << "done." << endl;
```

Note the use of `rebind()`. This is to avoid the `AlreadyBound` exception being thrown by the `NameServer` every time we have to restart the bank server. For a list of other exceptions as well as the definitive description of the `NamingContext` interface, you should look at the `CosNaming.idl` file under your `mico/include/coss` directory.

Again, we assume that the initial reference to the `NameService` is provided on the command line.

Appendix I - A Simple Application

I.1 Introduction

To clarify the process followed in section 5 for developing with middleware, a simple example of defining an IDL interface, implementing it in C++ and then writing the client program will be introduced. No unnecessary detail will be addressed.

I.2 A simple application

The example is based on the software required by a cinema to manage ticket sales. A simple interface to the front office object of this system is defined, an outline of implementation is given and a simple client is shown.

The interface to the front office is defined in the CORBA Interface Definition Language (IDL), and this interface is kept simple by defining only sufficient operations for finding the price of a given seat and to book a seat.

The interface to the front office object implemented in C++ and installed in a server. The server and client can be run on different machines in the Distributed System, or in different address spaces in the one machine, or in distance-based education environment within the same address space.

I.3 The Programming steps

The programming steps correspond to the process discussed in section 4 and are the typical steps required to implement a distributed client-server system in CORBA and C++.

- *Define the interfaces using the standard IDL.*
- *Implement these interfaces with C++ classes.*
- *Write a server main function that creates instances of these classes, and then inform the CORBA when initialization has been completed and the server is ready to accept requests.*
- *Register the server.*
- *Write a client main function to connect to the server and to use the server's objects.*

These steps will be illustrated in the remainder of this section.

I.4 The IDL specification

The first step in writing a CORBA program is to define the IDL interfaces to the application's objects. The interface to the front office can be defined in IDL as follows:

```
// IDL for front office. In file front.idl
// Start with some type definitions
typedef float Price
struct Place
{
    char row;
    unsigned long seat;
```

```
};

// Then the IDL interface
Interface FrontOffice
{
    readonly attribute string name;
    readonly attribute unsigned long numberOfSeats;

    Price getPrice (in Place chosenPlace);
    Boolean bookSingleSeat (in Place chosenPlace,
                          in string CreditCard);
};
```

The interface provides two attributes: `name` and `numberOfSeats`

- `name` gives the cinema's name as a string
- `numberOfSeats` gives the overall number of seats in the cinema

Both are labeled `readonly`, so that they cannot be directly changed by a client.

There are also two operations:

- `getPrice()` : this returns the price of a place. The place is specified as a structure, which gives the row and seat number. (The structure can be replaced by a class).
- `bookSingleSeat()` : this books a chosen seat for a client. A credit card number is passed (as a string) to that the client can be charged for the booking.

The parameters to the operations are labeled as `in`, which means that they are being passed from the client to the server. In other operations, parameters may be labeled as `out` (passed from the sever to the client) or `inout` (passed in both directions).

1.5 Compiling the IDL interface

The IDL specification must be compiled, both to check the specification and to map it into C++ so that it can be implemented and used. The IDL compiler produces three C++ files (the files names differ for various operating systems and compilers):

- `front.h` : A header file to be included into the implementation of the front office and into all clients. This defines the C++ view of the IDL definitions.
- `front.cpp` : A source file to be compiled and included into the clients of the front office. It includes the code required to make remote requests on a `FrontOffice` object. This is referred to as the **stub** code, and its role is to transmit requests, for the interfaces defined in `front.idl`, from a client to a server.
- `frontS.cpp` : A source file to be compiled and included into the implementation of the front office. It includes the code required to accept remote requests on a `FrontOffice` object. This is referred to as the skeleton code, and its role is to accept requests, for the interfaces defined in `front.idl`, from the client.

If the client and target object are in the same address space, no extra code is required to communicate between them. If they are in different address spaces, on the same or on different hosts, the extra code is required in the client to send the request to the server side, and extra code is required at the server side to accept the request and pass it to the target object.

I.6 The C++ produced

The `front.h` file produced by the IDL compiler contains the following code.

```
// C++
// Automatically produced (in front.h)
#include <CORBA.h>
typedef CORBA::Float Price
struct Place
{
    CORBA::Char row;
    CORBA::ULong seat;
};
// Some other definitions not relevant here are omitted.
class FrontOffice : public virtual CORBA::Object
{
public:
    virtual char * name()
        throw (CORBA::SystemException);
    virtual CORBA::ULong numberOfSeats ()
        throw (CORBA::SystemException);
    virtual Price getPrice
        (const Place & chosenPlace)
        throw (CORBA::SystemException);
    virtual CORBA::Boolean bookSingleSeat
        (const Place & chosenPlace,
         const char * creditCard)
        throw (CORBA::SystemException);
};
```

A number of other members for the class `FrontOffice` are generated, as well as other classes, but these will suffice. Each declaration will now be discussed.

Consider the operations, `getPrice()` and `bookSingleSeat()`. The mapping to C++ is very straightforward: each IDL operation is mapped to a C++ function. These C++ functions are labeled as virtual so that C++ will use dynamic binding. The parameter types are the natural translations from IDL (structs are passed by reference for efficiency). Each of these functions is also declared to throw an exception of type `CORBA::SystemException`.

The IDL basic types `char` and `unsigned long` are mapped to the C++ types `CORBA::Char` and `CORBA::ULong`. The header file `CORBA.h` provides appropriate typedefs to the equivalent basic types.

The two `readonly` attributes have been replaced with functions of the same names. Since the server and ent might not be in the same address space, there would be no point in replacing attributes with public member variables.

I.7 Implementing the interface

When C++ is used to implement the `FrontOffice` interface, the programmer must write a C++ class, which provides the functions, listed in the previous section. As mentioned in section 5.2.5 there are two approaches for indicating that this C++ class implements the IDL interface. The approach followed in this example is inheritance.

The implementation class will be called the `FrontOffice_i` class (in files `front_i.h` and `front_i.cpp`) and defining it as follows will indicate the fact that it implements the interface `FrontOffice`:

```
// C++
#include "front.h"
class FrontOffice_i :
    public virtual FrontOfficeBOAImpl
{
    // details shown later
};
```

The inheritance above is from `FrontOfficeBOAImpl`, which is automatically produced by the IDL compiler in the `front.h` header file. The name of this class is the interface name with the string 'BOAImpl' appended (BOA stands for Basic Object Adapter).

Class `FrontOffice_i` should be written to redefine each of the functions `name()`, `numberOfSeats()`, `getPrice()`, and `bookSingleSeat()`, and to add some member data, a constructor, and a destructor.

```
// C++. In file FrontOffice_i.h
class FrontOffice_i public virtual FrontOfficeBOAImpl
{
    char * m_name;
    CORBA::ULong m_numberOfSeats;
    CORBA::Char m_divide; // divide btw cheap and expensive
    Price m_highPrice;
    Price m_lowPrice;

    // The seat availability matrix
    const unsigned short m_rows;
    const unsigned long m_seatsPerRow;
    unsigned char ** m_avail; // Availability matrix

public:
    FrontOffice_i (
        const char *theName,
        const CORBA::ULong theNumberOfRows,
        const CORBA::ULong theNumberOfSeatsPerRows,
        const CORBA::Char theDivide,
        const Price theHighPrice,
        const Price theLowPrice );
    virtual ~FrontOffice_i();
```



```
        // Functions corresponding to the 2 IDL attributes
virtual char * name()
    throw (CORBA::SystemException);
virtual CORBA::ULong numberOfSeats ()
    throw (CORBA::SystemException);

        // Functions corresponding to the 2 IDL operations
virtual Price getPrice (const Place & chosenPlace)
    throw (CORBA::SystemException);
virtual CORBA::Boolean bookSingleSeat
    (const Place & chosenPlace,
     const char * creditCard)
    throw (CORBA::SystemException);
};
```

Class `FrontOffice_i` could also add further functions and constructors. The functions that have been defined can be implemented as follows:

```
// C++. In file FrontOffice_i.cpp
FrontOffice_i::FrontOffice_i(
    const char *theName,
    const CORBA::ULong theNumberOfRows,
    const CORBA::ULong theNumberOfSeatsPerRows,
    const CORBA::Char theDivide,
    const Price theHighPrice,
    const Price theLowPrice      ) :
    // Consistency checks are not shown here
    m_rows(theNumberOfRows),
    m_seatsPerRow(theNumberOfSeatsPerRows0,
    m_divide(theDivide),
    m_highPrice(theHighPrice),
    m_lowPric(theLowPrice)
{
    m_name = new char [strlen(theName) + 1];
    strncpy(m_name, theName);
    // Now allocate space to the availability matrix
    for (int i = 0; i < m_rows; i++)
    {
        m_avail[i] = new unsigned char [m_seatsPerRow];
        for (int j = 0; j < m_seatsPerRow; j++)
            m_avail[i][j] = 1;        //Initially available
    }
}

FrontOffice_i::~~FrontOffice_i()
{
    delete [] m_name;
    // Free the space of the availability matrix
    for (int i = 0; i < m_rows; i++)
        delete m_avail[i];
    delete [] m_avail;
}

char * FrontOffice_i::name()
    throw (CORBA::SystemException)
{
    // Settings that are passed out through a CORBA interface,
    // must be allocated using CORBA functions, such as
```

```

// string_dup()
    return CORBA::string_dup(m_name);
}
CORBA::ULong FrontOffice_i::numberOfSeats ()
    throw (CORBA::SystemException)
{
    return m_numberOfSeats;
}
Price FrontOffice_i::getPrice (const Place & chosenPlace)
    throw (CORBA::SystemException)
{
    if chosenPlace.row < m_divide)
        return m_lowPrice;
    else
        return m_highPrice;
}
CORBA::Boolean FrontOffice_i::bookSingleSeat
    (const Place & chosenPlace,
     const char * creditCard)
    throw (CORBA::SystemException)
{
    // Bonds testing not shown
    unsigned long rowIndex = chosenPlace.row - 'A';
    if (m_avail[rowIndex][chosenPlace.seat])
    {
        // Book that place
        m_avail[rowIndex][chosenPlace.seat] = 0;
        // Charge the price to te credit card
        // the code is not shown hete
        return 1;
    }
    else return 0;
}

```

Rather than having `FrontOffice_i` inherit from `FrontOfficeBOAImpl`, a programmer can implement `FrontOffice_i` as a standalone class and **TIE** this class to the `FrontOffice_i` IDL interface.

1.8 Providing a server

The next step in the process is to write a server program in which the `FrontOffice` object will run. In this example, the client and server are in separate address spaces.

```

// C++. In file srv_main.cpp.
#include "front.h"
#include <iostream.h>
int main()
{
    // Any number of objects can be created
    // In this example only one object is created
    FrontOffice_i myFrontOffice("Royal",25,40,'G',5.00, 3.00);
    // Now the server must be initialized.
    // This varies according to the is CORBA
    // implementation **
    CORBA::-----.impl_is_ready("FrontOfficeSrv");
    cout << "Server terminating" << endl;
}

```

(** There is a CORBA standard way of communicating with the ORB, but for simplicity this function call has been used instead.)

This server program creates a `FrontOffice_i` object. It then indicates that the server's initialization has been completed, by calling `impl_is_ready()`. The parameter to `impl_is_ready()` is the name of the sever as registered in the implementation Repository.

I.9 Registering the server

The server can be registered so that it will be run automatically when a client uses the `FrontOffice` object. The system maintains a database of servers known as the Implementation Repository, which maintains a mapping from a server's name to the name of the executable file that implements that server.

The registration step is not strictly required, but carrying it out does have the advantage that a server will be launched if it is not running when one of its objects is invoked by a client.

I.10 Writing a client

A client can be written as follows. Note that no error handling is included for simplicity.

```
// C++. In file client.cpp.
#include "front.h"
#include <iostream.h>
int main()
{
    FrontOffice_var foVar;
    foVar = FrontOffice::_bind(":FrontOfficeSrv")
    CORBA::String_var itsName = foVar->name();
    cout << "Name is " << itsName << endl;
    cout << "Number of seats is " << foVar->numberOfSeats();
    Place p = {'B', 27};
    cout << "Price of seat B27 is " << foVar->getPrice(p);
    if (foVar->bookSingleSeat(p, "4531 3458 9489 8212"))
        cout << "Seat B27 booked ok." << endl;
    else
        cout << "Seat B27 cannot be booked. " << endl;
}
```

Note that the variable `itsName` could have been defined to be a `char *`, but the type `CORBA::String_var` ensures that the space for the string will be freed automatically when `itsName` goes out of scope.

The client defines a variable of `FrontOffice_var`. This type is a C++ helper class automatically generated by the IDL compiler from the `FrontOffice` interface. As an instance of `FrontOffice_var` holds a reference to a C++ proxy object, which is a C++ object that acts as a representative or stand-in for a remote object. Thus, the client will have a proxy for the remote `FrontOffice` object that it uses and its `foVar` variable will hold a pointer to that proxy.

This concludes the discussion of this simple example. It is clear that its purpose was just introductory, as more detail is needed to gain a clear understanding.