1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Distributed Systems Prototyping

by

Abdelsalam Shanneb

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science

**School of Computer Science
Carleton University
Ottawa, Ontario
August 10, 1994
© copyright
1994, Abdelsalam Shanneb**

**National Library
of Canada**

**Acquisitions and
Bibliographic Services Branch**

**395 Wellington Street
Ottawa, Ontario
K1A 0N4**

**Bibliothèque nationale
du Canada**

**Direction des acquisitions et
des services bibliographiques**

**395, rue Wellington
Ottawa (Ontario)
K1A 0N4**

Your file Votre référence

Our file Notre référence

**THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.**

**L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.**

**THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.**

**L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.**

ISBN 0-315-98565-8

Name ABDELSALAM SHANNER

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science

SUBJECT TERM

0984

U-M-I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0316
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0310
Marketing 0509
Canadian Studies 0510
Economics 0511
General 0358
Agricultural 0366
Commerce-Business 0351
Finance 0578
History 0579
Labor 0578
Theory 0579
Folklore 0578
Geography 0578
Gerontology 0578
History 0578
General

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
General 0938
Criminology and Penology 0331
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family Studies 0630
Industrial and Labor Relations 0700
Public and Social Welfare 0344
Social Structure and Development 0709
Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology 0306
Biology 0287
General 0308
Anatomy 0309
Biostatistics 0379
Botany 0329
Cell 0353
Ecology 0369
Entomology 0793
Genetics 0410
Limnology 0307
Microbiology 0317
Molecular 0416
Neuroscience 0433
Oceanography 0821
Physiology 0778
Radiation 0472
Veterinary Science 0786
Zoology 0760
Biophysics 0760
General 0425
Medical 0996

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0494
Pharmaceutical 0495
Physical 0754
Polymer 0405
Radiation 0605
Mathematics 0986
Physics 0606
General 0608
Acoustics 0748
Astronomy and Astrophysics 0607
Atmospheric Science 0798
Atomic 0759
Electronics and Electricity 0609
Elementary Particles and High Energy 0610
Fluid and Plasma 0752
Molecular 0756
Nuclear 0611
Optics 0463
Radiation 0346
Solid State 0984
Statistics

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal 0790
System Science 0428
Geotechnology 0796
Operations Research 0795
Plastics Technology 0994
Textile Technology

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,

Distributed Systems Prototyping

submitted by

Abdelsalam Shanneb, B.C.S.,

in partial fulfillment of the requirements
for the degree of
Master of Computer Science

Thesis Supervisor

Director, School of Computer Science

Carleton University

September 20 , 1994

Abstract

Distributed operating systems have become increasingly influential with the development of cost-effective microprocessors and with the invention of high-speed local area networks (LANs). However, distributed applications need radically different software than do centralized system applications, furthermore, much software produced using established development methods cannot easily accommodate changes, forcing portions to be rewritten or discarded after installation. The prototyping development paradigm alleviates these problems by using an iterative process involving experimentation, thus tools must be provided which minimize the cost of experimentation.

This thesis proposes a design and implementation of a set of prototyping tools that can be used in building prototypes of distributed systems applications. This set of tools is realized by an object-oriented layer that sits on top of the existing PARASOL kernel. This layer, PARASOL Object-Oriented Layer (POOL), provides an object-oriented environment for distributed systems functional prototyping and performance prediction.

Acknowledgments

This thesis could not have been completed without the support and guidance of my supervisor, Dr. John Neilson. Without his patience and advice, the POOL project would not be a reality. I owe him a lot.

I wish to thank my wife, Samia, for her support and patience, and also Alla, Mohamed and Hajer, who have endured the hardships of being the children of a student.

Lastly, I thank the Natural Sciences and Engineering Research Council (NSERC), as well as Dr. Murray Woodside, from the Department of Systems and Computer Engineering, for their much appreciated financial support of this research project.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	ix
List Of Figures	x
1 Introduction	1
1.1 Background.....	1
1.1.1 The Life Cycle Model.....	3
1.1.2 What Is Software Prototyping?.....	4
1.1.3 Approaches to Prototyping.....	6
Throw-away Approach.....	6
Evolutionary Prototyping.....	7
1.1.4 Computer-aided Prototyping.....	9
1.1.5 Prototyping Techniques.	10
Executable Specification.....	10
Very High-Level Languages.....	11
Tool-Set Approach.....	12
Object-Oriented Languages.....	12
1.1.6 Prototyping and Performance.....	13
1.2 Thesis Objectives.....	15
1.3 Thesis Organization	16

2 PARASOL.....	18
2.1 Introduction	18
2.2 PARASOL "Hardware" Components	19
2.3 PARASOL "Software" Entities.....	20
2.4 PARASOL Task States.....	23
2.5 PARASOL & Prototyping	24
 3 POOL Architecture & Design.....	26
3.1 PARASOL Object-Oriented Layer (POOL)	26
3.2 POOL Architecture	27
3.2.1 POOL & Object-Oriented Design.....	27
3.2.2 The Design Approach.....	30
3.3 POOL Classes	32
 4 Implementation Issues.....	37
4.1 Interfacing PARASOL with POOL	37
4.1.1 Justification	37
4.1.2 POOL & PARASOL Integration Method.....	38
4.1.3 POOL's Wrappers.....	40
4.2 The Carrier Class.....	43
4.2.1 Carrier.h.....	43
4.2.2 The Carrier Constructor.	45
4.2.3 The Transmitter Task	46
4.2.4 The Carrier Task	47
4.2.5 The Carrier_Send Command	48
4.2.6 The Receiver Task	49

4.3	The Disk Class.....	49
4.3.1	Characteristics of a Disk.....	50
4.3.2	Positioning Times	51
4.3.3	The Disk Constructor	51
4.3.4	The Disk Server Task.....	53
4.3.5	The Disk Manager Task	54
4.4	The Directory Server Class.....	55
4.4.1	Name Server V/s. Directory Server.....	57
4.4.2	Dir_Server.h.....	58
4.4.3	Internal Behavior	59
4.5	The File_Server Class	60
4.5.1	File_Server.h	60
4.5.2	The File_Server Constructor.....	62
4.5.3	The File_Server Thread(s)	63
4.6	The Pool_User Class	64
5	Network File System Prototype.....	67
5.1	Background	67
5.2	POOL NFS Class	70
5.2.1	NFS Constructor	76
5.2.2	System_Calls Task.....	77
5.2.3	VFS Layer	78
5.2.4	Client Task	80
5.2.5	Server Task	82
6	Conclusions	84
6.1	Summary	84

6.2 Performance	85
6.3 Future Work	87
Appendix A: Pool_User Class Users' Manual	89
A.1 The Carrier Class	89
A.2 The Disk Class	90
A.3 The Dir_Server Class	91
A.4 The File_Server Class	91
A.5 The NFS Class	94
Appendix B: Pool_User Example	96
References	103

List of Tables

Table 6.1 Comparing Bus & Carrier Total Execution Times (Seconds).....	86
--	----

List of Figures

Figure 1.1 Throw-away Prototyping	7
Figure 1.2 Evolutionary Prototyping	8
Figure 1.3 Prototyping Environment	16
Figure 2.1 PARASOL Hardware Entities	20
Figure 2.2 Tasks Creation Tree	21
Figure 2.3 PARASOL Ports	22
Figure 2.4 A Simplified PARASOL Task Transition Diagram.....	24
Figure 3.1 POOL Layers	27
Figure 3.2 Hybrid Approach.....	29
Figure 3.3 POOL Prototyping Environment	30
Figure 3.4 POOL Extensions	31
Figure 3.5 User-Defined Extension Layer	32
Figure 3.6 Pool_User Class	34
Figure 3.7 A "flexible" File Services	35
Figure 3.8 POOL Classes	36
Figure 4.1 Global Indexing Array.....	41
Figure 4.2 POOL Meets PARASOL	42
Figure 4.3 The Carrier Constructor	46
Figure 4.4 The Carrier Task	47
Figure 4.5 Packets Traveling Through the Carrier	48

Figure 4.6 The Mechanical Components of a Disk.....	50
Figure 4.7 A Directory Hierarchy	57
Figure 4.8 Fs_task Behavior.....	64
Figure 5.1 Mounting operation	68
Figure 5.2 NFS layers	69
Figure 5.3 POOL NFS Layer	70
Figure 5.4 The System calls layer.....	77
Figure 5.5 VFS Layer	79
Figure 5.6 The Client Layer.....	81
Figure 5.7 NFS Server	83
Figure 6.1 Microkernel Prototyping Environment	87
Figure B.1 The Disk_Tester	97

Chapter 1

Introduction

This chapter sets the scene for this thesis report by first presenting a background on the topic of prototyping and its related issues, then by giving thesis objectives and thesis organization.

1.1 Background

The last decade has seen the creation and rapid expansion of the field of distributed computing systems. This has been driven by two major forces namely communication and computation [Mullender 89], [Zedan 90]. Long haul, relatively slow communication paths between computers have existed for a long time, but only recently has the technology for fast, cheap and reliable local area network (LAN) emerged. These LAN's allow the connection of large numbers of computing elements with a high degree of information sharing. These have become relatively cheap and plentiful with the advances of

microelectronics and microprocessor technology. In response, the wide area networks (WAN) are becoming faster and more reliable.

Distributed systems have a number of potential selling points. They can offer good price/performance ratios, can be made highly reliable, and can be gradually increased in size as the workload grows [Tanenbaum 85,88,92]. However, they also have some disadvantages, such as having more complex software, potential communication bottlenecks, and weak security. Nevertheless, many computer systems tend to adopt distributed architecture in order to more fully utilize system resources.

Many distributed systems demand good performance. Researchers have noted that the behavior of such distributed systems become so complex that it is very difficult to verify the specification of the systems [Chang 87]. However, it is highly desirable to determine if it is feasible to implement a system according to its specification. For such a feasibility study, prototyping has been shown to be a promising way to accomplish the goal [Taylor and Standish 82], [Cieslak 88]. Prototyping is particularly suitable in the early phase of the development of distributed systems due to the ever increasing complexity and the inevitable high development cost of such systems [Luqi 88]. Prototyping has proven to be a promising way of accomplishing a feasibility study. It cuts costs and reduces the complexity of the development of distributed systems.

Prototyping has been advocated as a method of obtaining information about the potential implementation of a system under design, before the production system has been developed and has been used to verify the specification and behavior of a system's design before implementation.

1.1.1 The Life Cycle Model

During the last thirty years, software system development has been based on a model, commonly referred to as the software life cycle model. Though characterized differently by different authors, its overall theme is well understood and universally acknowledged. The life cycle model leads to a software development strategy that is usually called the phase-oriented, the linear or the traditional strategy. The life cycle model essentially advocates that software projects should consist of a number of distinct stages. These are: requirements analysis, requirements specification, design, implementation, validation, verification, operation and maintenance. The life cycle model was originally derived from the hardware production model of requirements, fabrication, testing, operation and maintenance [Blum 82]. It primarily reflects management concerns in production, such as planning, control, budget expenditure and resource allocation. Central to the model is its linear structure, with exception of validation and verification, all other stages are carried out linearly, i.e., each stage begins only when the previous stage has been completed.

The life cycle model is strongly based on the assumption that a complete concise and consistent specification of a proposed system can be produced prior to design and implementation. The validity of this assumption has been challenged and refuted by a number of authors [Swartout and Balzer 82, McCracken and Jackson 82, Shaw 85]. In many cases a complete specification cannot be produced, simply because the user does not really know what he wants. Lack of experience in projects where it is almost impossible to construct a precise specification leads to the situation where the customer requirements can be established only when a complete software system has been built and when the system

can be examined in a fully concrete form. For this reason many systems end up being written at least twice.

It has been suggested, therefore, that the life cycle model has many shortcomings which may have adverse effects on software projects. This is, of course, not to say that this model should be totally rejected. To the contrary, in certain areas, such as embedded software and real time control systems, it is the most rational approach and indeed the best way of controlling the complexity of such projects. However, for the majority of other applications, especially those related to commercial data processing, the life cycle model is inappropriate and has many deficiencies that are too serious to be ignored. The prototyping solution is a technique used during the process of software development, and is considered one of the solutions to the problems associated with the conventional software development method.

1.1.2 What Is Software Prototyping?

A software prototype is a preliminary version or a model of all or part of a system before full commitment is made to develop it. Prototyping is the process of creating the prototype. Prototyping involves producing early work (prototypes) of the future application systems and experimenting with them. Prototyping addresses two major problems of the essence: the difficulties involved in requirements validation, and the need for the representations of requirements that are understandable by a number of different readers. It involves developing an executable version of the functional and interfaces aspects of the system, usually in some language that will allow the prototype to be developed easily but will not have the performance characteristics of the target implementation.

Software prototyping is an approach to software construction based on experiment and experience. It provides a communication basis for discussions among all the groups involved in the development process, especially between actual end users and the developers. In reality, customers are often unaware of exactly what they want, they may be aware that a particular problem needs an automated solution, or that a system with various objectives must be developed, but the details of the system requirements may be unclear. By presenting customers with a prototype, they can clarify their ideas, and identify exactly what it is that they need. Furthermore, if there are visible gaps or deficiencies in the prototype, then customers may be prompted to volunteer more information. The prototype may thus be used to aid in ensuring the completeness of the requirements.

Since a prototype is an implementation of the functional aspects of a computer system, then to implement the prototype, a certain amount of software design and coding must take place. If the system is large and complex, this is likely to take a great deal of effort. There are even systems for which the production of a prototype is completely infeasible or at least exceedingly difficult, for example, embedded systems, these do not have user interfaces through which the functionality of the system may be tested, and they often involve control system.

Another disadvantage is that the prototype may implement only part of the requirements, then clearly those aspects of the requirements that have not been included in the prototype cannot be validated through its use, consequently, validating the prototype will validate part of the requirements. It is clear that although there is a number of significant advantages to the development of system prototypes, great care must be exercised during the production of the prototype and the system to ensure that they are

consistent. Ideally, a simple rule by which validating of the prototype implies of the requirements is required

1.1.3 Approaches to Prototyping

There are mainly two different approaches to software prototyping. They are: the throw-away prototyping approach and the incremental or evolutionary prototyping approach.

Throw-Away Approach

This approach usually corresponds to the popular view of prototyping in which the prototype is built in a 'quick and dirty' manner, simply to identify basic requirements and illustrate system potential, it is then discarded, and the system is built using more formal methods. Strictly speaking, no part of the throw-away prototype should be reused in any form. Since the prototype is to be used for a limited period, quality factors such as efficiency, structure, maintainability, error handling and documentation are of little relevance. The prototype may even be implemented on hardware or within an environment other than the one required for the target system. What is important about throw-away prototyping is the process itself and not the product. The major part of the effort, therefore, should go into the critical evaluation of the prototype rather than its design.

Throw-away prototypes are sometimes perceived as a waste of effort, and there is some justification for this point of view. The simple-minded, management argument is that developing code that will be thrown away is a waste of resources. It is true that prototype code is often too inefficient and insufficiently general to be directly incorporated into final product, but this argument ignores the fact that production-quality code often must be discarded because it is based on incorrect requirements. It is more cost-effective to correct

the requirements by evaluating and discarding a relatively inexpensive prototype instead of an expensive implementation. Figure 1.1 shows a schematic of throw-away prototyping in which prototypes are redeveloped or translated into another form before the target system is delivered

Figure 1.1 Throw-away Prototyping

Evolutionary Prototyping

In this approach, the prototype is built in a more structured manner, and is continually adjusted until it meets the full system requirements. At this point, the final version of the prototype becomes the first live version of the target system. With the

evolutionary approach, the aim is to manage risk by only defining and developing a small part of the system at a time. Thus, a 'core' system is first designed and implemented, then additional functionality is implemented in a series of future overlapping subprojects. Figure 1.2 shows a schematic of evolutionary prototyping in which prototypes themselves become the delivered target systems.

Figure 1.2 Evolutionary Prototyping

A great attraction of this approach is that at most times there is an executable, usable system and it is possible for users to give early feedback on, and gain confidence in the implemented system. However, some oppose this approach by arguing that there may be difficulties in scaling the methods up to very large systems. Until recently, both prototyping approaches seem to have been used equally extensively, and with equal success, however, most of the new methodologies of the last five or six years seem to have favored the evolutionary approach.

1.1.4 Computer-aided Prototyping

Prototyping is an attractive approach to systems development, but its practical usefulness depends on technologies for computer-aided design and analysis. Software tools facilitate and speed prototyping and help analysts formulate, understand, and communicate a proposed system's properties to users. A computer-aided prototyping environment should be integrated with tools for measuring, optimizing, and refining the prototype design into a production-quality product.

A computer-aided prototyping system reduces the designer's efforts by automating time consuming tasks in conventional prototyping, such as turning specifications into prototypes, modifying prototypes, and searching for available reusable components. Progress to date has shown that it is feasible to develop this technology [Luqi 88], but a great deal of research and development remain before this technology realized its full potential.

Prototypes may only be of value if they can be produced quickly, hence the common use of the term 'rapid prototyping.' It is above all in the US literature that this term is frequently encountered. Most authors use it to denote a concept that does not significantly differ from the one in the European literature. It acquires a totally different meaning, however, in cases where, given the efficient tools currently available for software construction, thorough requirements analyses and system design processes are abandoned in favor of an unsystematic, trial-and-error approach to software development. Rapid prototyping, in this sense, means that the prototype is the only valid document in the system development process.

Some see rapid prototyping as a seriously misguided emphasis. If staff is expected to produce software rapidly, at the expense of all other qualities, then the quality and maintainability of their work is bound to suffer. They argue that emphasis on prototyping should be less about speed of development and more about involving the users throughout the development.

1.1.5 Prototyping Techniques

Prototyping techniques and prototyping tools are the software engineering aids needed by a programmer in order to build an application system. There are scarcely any instances of prototype construction without special tools or toolkits. A prototyping tool is of reduced value if it cannot create the entire prototype within the scope of its own inherent facilities.

Executable Specification

The rationale behind such systems is simple. If it is possible to take a specification and execute it or animate it in some way, then an early version of the software system can be delivered during the requirements analysis phase. This is an attractive idea because the specification of a software system has to be produced, and no extra effort for prototype construction would therefore be needed. This is in contrast to other techniques, such as those using very high-level languages where, after a prototype has been built and agreed with the customer, the specification has to be constructed anew.

One of the earliest systems for execution specification was based on the Me-Too language developed at Stirling University [Henderson 86]. A much more ambitious system is the EPROS system [Hekmatpour and Ince 88]. This is based on UNIX, and offers a wide spectrum language EPROL as the prototyping medium. Functional languages are also

a good medium for specification execution prototyping. It is easy to write a mathematical specification using these languages, and execute it with little further work.

Very High-Level Languages

A Very High-Level Language (VHLL) is one in which it is possible to express compiled operations using a small amount of written code. The disadvantages of such languages are that they tend to be inefficient in terms of run time or memory usage. In general, VHLLs are interpretive and interactive, offer a rich set of data objects and corresponding operations, contain a language notation which is short, concise and very expressive, and are normally supported by very powerful programming support environments and debugging facilities. It is the expressive power, together with the ease with which modifications can be carried out, that make such languages ideal for prototyping.

Probably the earliest VHLL to be used for prototyping was APL [Tavo 84]. This is a language that was very popular in the early 70's. Another language that has a long history of being used for prototyping is Lisp. Normally this language is associated with AI software development. Because such development is exploratory and experimental, powerful versions of Lisp are available which probably have the best program support environments associated with a VHLL. A more recent programming language is Prolog [Dwig 83], [Bruy 84], [Lieb 84], its declarative style and implicit control mechanism make rapid development possible. Because they are executable and formal, prototypes written in Prolog should be verifiable and transformable. In addition, most Prolog environments contain good debugging facilities which reduces development time. Prolog is incorporated into MLog [Karam 89], which is designed to be a rapid prototyping language combining the object-oriented, logic programming, and concurrent system paradigms. No VHLL is

suitable for prototyping all applications. Normally, a choice has to be made, according to the application area.

Tool-Set Approach

This approach uses a software system that provides a series of processors and tools which can be joined together easily. The best system exemplifying the tool set approach is UNIX [Kern 84] which contains utilities such as YACC and LEX that are capable of processing tables of definitions and producing software very quickly.

Object-Oriented Languages

Object-oriented programming is appealing for prototyping because it promotes code reuse. In object-oriented languages the first-class entities are objects which usually match objects in the external world. Object-oriented languages have a long history, although it is now that they are coming into prominence. The earliest language with object-oriented properties was Simula 67. However, the most well-known object-oriented languages are Smalltalk [Gold 84] and C++ [Stroustrup 90]. The importance of object-oriented languages for prototyping lies in a property known as inheritance. This allows the programmer to define a base object, say a plane, and operations on that object, and then with little effort to define new objects based on that object, for example, fighter planes and transport planes. Object-oriented languages can be used by the developer to build up a library of objects and operations which are required in the application area and then, for each application to be prototyped, to develop the objects in the applications by using the inheritance facilities of the language. This normally requires a little programming effort. In essence, prototyping is achieved by a very sophisticated software re-use technology.

1.1.6 Prototyping and Performance

A prototype is typically used as a demonstration model or to verify functionality aspects of a particular model, however, using a prototype to predict performance is a distinctly different form of usage. A prototype can be a model of the target software which would be used with simulated execution environment to predict performance.

Viewpoints relating prototyping and performance prediction have varied. It has been suggested that prototyping and performance prediction are orthogonal, and that generating a prototype for performance evaluation is "costly and inefficient" [Nichols 90]. However, [Zave 86] suggests that there are existing examples where performance information can be provided by a prototype, and prototypes can be more effectively used for performance predictions by enhancing the prototype, and extending the prototype process, extraction of performance information from a prototype early in the design process is possible.

However, for a prototype to provide explicit and useful performance information, it will require technology advancements and a paradigm shift. For a prototype to provide useful performance measures it needs detailed behavior and resource usage information. A prototype must be equipped with probes and measuring tools to enable developers to identify system bottlenecks and potential problems in resource utilization.

Related system used for simulation and prototyping distributed and concurrent environment have different characteristics and behavior. TimeBench [Buhr 90a] allows concurrent system designs to be entered graphically using Buhr's machine charts notation [Buhr 90b], and automatically generates the code in a special machine charts language called MCL, which specifies how the concurrent components interact with each other. The

remainder, containing algorithmic details, is specified by developers using MCL. MCL contains many features for easing the coding burden on developers, such as lists which automatically expand and contract as needed. An MCL interpreter executes the resulting program and provide visual feedback by animating the machine charts entered by the developer.

PROTEUS [Brewer 91] is a high-performance simulator for MIMD multiprocessors. It can reproduce results from real multiprocessors, and it is easily configured to simulate a wide range of architectures. PROTEUS provides a modular structure that simplifies customization and independent replacement of parts of architecture. This modular structure allows users to tradeoff accuracy and performance: users pay for accuracy only when and where they need it. The structure also allows easy customization of the architecture.

PARASOL is a task-oriented, discrete-event simulator, specially designed for simulating or emulating distributed/parallel computer systems [NEILSON 91a]. Tasks, distinct from the process concept used by other simulation languages, are user definable and execute in a simulated distributed environment. PARASOL simulates through emulation by executing these software tasks which can also be scheduled, and run independently in a multitask operating environment similar to UNIX. Their definitions are actual functions written in C or C++. This approach in defining tasks gives the maximal flexibility, as well as the similarity to the real world system. System topology in PARASOL is described with computing nodes, communication buses and links.

1.2 Thesis Objectives

Having demonstrated the aspects of prototyping and using prototypes in the process of developing system software and with the increase importance of distributed systems, there is a well acknowledged need for convenient prototype development environments for distributed system software. Distributed systems designers would be very interested in a toolkit that enables them to prototype a system's functionality to determine its feasibility before they commit themselves with the development of the target system. It is even more beneficial if the same toolkit was designed not only for functionality prototyping but also for performance prototyping.

The primary objective of this thesis is to provide an object-oriented environment for distributed systems performance and functional prototyping. Under this banner, more specific objectives are listed as follow:

- To develop a toolkit that can be used in prototyping distributed and/or parallel systems and make use of PARASOL to provide the basic underpinnings such as user-defined hardware execution environment, task management, and inter-task communication. Figure 1.3 shows the abstraction layering of the prototyping environment.
- To construct a first version of the toolkit (POOL) as an object-oriented layer which is basically a C++ class library containing entities particularly useful to system prototypers.
- To demonstrate the usage of the class library and to show how this first version layer can grow and evolve.
- To demonstrate the efficiency of this toolkit by providing some measurement of its throughput.

Figure 1.3 Prototyping Environment

1.3 Thesis Organization

Chapter 2 of this thesis deals with PARASOL, the predecessor and foundation of POOL. A high level overview of PARASOL is presented. The presentation includes a description of PARASOL system components and their relationships. Task scheduling and inter-process communication issues are also given.

Chapter 3 presents the architectural and design issues of POOL. It shows the methodology and approach used in choosing and designing each class of POOL. This chapter is considered as a high level view of POOL without going into the implementation details.

Chapter 4 is dedicated to the implementation details and issues of POOL. It first demonstrates how POOL (the object-oriented paradigm) and PARASOL (the procedural

approach) were integrated together, then it describes the tiny details and internal workings of each class in POOL

Chapter 5 deals with application aspects of POOL by presenting a tool built using POOL's abstractions. The chapter shows how a Network File Server (NFS) prototype was constructed using the distributed system entities provided by POOL.

Chapter 6 is the conclusion section of this thesis where it deals with the final results and efficiency measurements testing and discusses some possible future directions.

Appendix A contains a users' manual that gives a brief description of each method in the class Pool_User.

Appendix B contains the code of an example showing the use of Pool_User class. This is a good example that demonstrates some of the concepts explained in this report.

Chapter 2

PARASOL

This chapter gives a very high level overview of PARASOL. At the beginning a brief introduction to PARASOL section is given, then followed by three sections dealing with the different PARASOL entities. The last section is devoted to PARASOL's role in prototyping. For more details on PARASOL readers are referred to [NEILSON 91a], [NEILSON 91b], and [NEILSON 91c].

2.1 Introduction

PARASOL is a software library package which is used to model computer systems, it is specifically suited for modeling distributed and parallel systems. In particular, PARASOL supports two distinct kinds of modeling activity: simulation and emulation. PARASOL is a simulator in situations where the primary goal is to conduct a performance

analysis of the modeled system. It is an emulator, on the other hand, in situations where the goal is to create a functional prototype of a computer system.

PARASOL is " a medium-to-fine-grained simulator/emulator for distributed and/or parallel computer systems" [NEILSON 91a]. Tasks in PARASOL are user-defined functions which execute on a user-configurable network. PARASOL runs in a single processor environment, however it emulates the user defined distributed or parallel environments through specialized task scheduling.

2.2 PARASOL "Hardware" Components

At first approximation, a distributed system consists of a group of processing nodes connected together by a common communication medium for the purpose of resource sharing or communication matters. PARASOL users are offered the means to abstract processing nodes that can be interconnected either with direct, one-way links and/or bus-type links. Figure 2.1 shows PARASOL hardware entities.

A Node may be defined as either a single processor or a shared memory multiprocessor. A multiprocessor node has a common ready-to-run queue of tasks shared by all processors. The discipline of processor scheduling may be First-Come-First-Serve (FCFS), Preemptive Resume (PR), or Non-Preemptive Priority (NP). In each of the queuing disciplines, a round-robin option is provided yielding to a total of six different queuing disciplines that cover most of the common strategies.

Other attributes of the node are its name and its speed factor which indicates the relative speed of processors inside the node. Two nodes can be connected by a one-way, asynchronous communication link where data sent over a link can be packetized in various

sizes. Pairs of links can be used when two-way data flows are required. A set of nodes may be connected using the bus abstraction which provides multiway communication among them. PARASOL link communication arbitrations are FCFS, whereas bus communication arbitrations may be FCFS or it can be random to provide a crude model of CSMA/CD protocols.

Figure 2.1 PARASOL Hardware Entities

2.3 PARASOL "Software" Entities

Applications software is modeled in PARASOL by coding tasks. The most remarkable feature of PARASOL is its task concept and usage. Being actual functions

written in C (or C++), PARASOL tasks give users a flexible tool to closely mimic the behavior of the modeled system. Tasks are the most active entities in a PARASOL system in which they are used to model everything but nodes, buses and links. When created, a task is assigned to run on a specific node and permitted to execute on any of the processors within the assigned node. Complexity of the tasks may vary depending on the degree of modeling detail required, this may be, at one extreme, writing a code for a real application in detail or, at the other, coding a simple entity in the model being simulated.

A special task called "ps_genesis" must be written in addition to writing other tasks. The task ps_genesis executed on node 0 which is a special node automatically built for every PARASOL model. Ps_genesis allows users to define the main elements of the execution environment which includes building the hardware topology and launching tasks to start the simulation process. It becomes the root of the user defined task creation tree (Figure 2.2).

Figure 2.2 Tasks Creation Tree

A finer grain view of a PARASOL model reveals details including ports and messages for inter-process communication, spin-locks for process synchronization, statistical counters, etc.. Inter-process communication is defined through ports and messages, communication links and bus queues contain messages to be sent to their destinations known as ports. To send a message, a task creates a message and directs it to a specific port that belongs to the target task. A message has four attributes: a system supplied time stamp and a user-defined type code, reply port and pointer to a message text. Port queues are strictly FCFS, a port is like a mailbox from which the owning task can retrieve messages. A standard port is assigned to each task at the time the task is created, however, a task may own an arbitrary number of other ports as needed (Figure 2.3). Within the private ports belonging to any given task, ports may be grouped into port sets to facilitate message handling. A port may also be shared by several tasks. As well, ports may be passed from one task to another along with any queued messages.

PARASOL provides arrays of spin-locks and counting semaphores for task synchronization purposes. Besides the synchronization mechanism provided by message passing, spin-locks and semaphores can be used directly for fine-grain mutual exclusion synchronization or as building blocks to construct more elaborate synchronization mechanisms.

Figure 2.3 PARASOL Ports

In the area of statistics capturing, PARASOL provides three different mechanisms . The first mechanism is utilized through the automatic capturing of hardware device utilization statistics by specifying the statistic flag to be true on the creation of the hardware entities. PARASOL provides two more mechanisms for capturing two types of statistical data. First there is the type used to capture device utilization statistics and it is referred to as "VARIABLE" since users are interested in tracking some system state variable over time and measuring its mean value, e.g., queue length, cpu utilization, etc.. The second type which is referred to as "SAMPLE" statistics is concerned with population statistics where users will collect population observations during the course of the simulation and report sample means values, e.g. mean response time, mean waiting time, etc..

2.4 PARASOL Task States

Scheduling of tasks in PARASOL is analogous to scheduling of processes in an operating system. Figure 2.4 shows the main features of the task state space and state transition space. Task states in PARASOL are divided into two groups. The first group is referred to as "executing" states group and the other one is referred to as "non-executing" states. For more detailed description of the internal workings of PARASOL, interested readers are referred to [NEILSON 91a, 91b].

Figure 2.4 A Simplified PARASOL Task Transition Diagram

2.5 PARASOL & Prototyping

The PARASOL kernel acts as a sort of a simplified distributed operating system by providing pseudo system calls to support task management, inter-task communication, and task synchronization. PARASOL was designed specifically for performance modeling and not for prototyping, so it has some weaknesses with respect to prototyping.

Although it provides an execution environment for user-defined tasks together with message passing, synchronization, statistics collection, etc. . PARASOL is lacking the "building blocks" for simulated disks, file systems, name servers, i/o streams, etc. . Besides the software entities in PARASOL, simulation of some hardware entities is also

provided, however, these models are crudely designed, for example, the bus abstraction in PARASOL does not support "true" packetizing of messages across the network.

PARASOL supplies a task-based paradigm for describing distributed systems which makes it a natural candidate for prototyping, furthermore, it supplies all of the necessary hooks for performance prototyping. So it is advantageous to recognize the current basic capabilities of PARASOL and build the lacking facilities using PARASOL's capabilities as underpinnings to provide tools that explicitly support distributed systems abstractions building.

Chapter 3

POOL Architecture & Design

This chapter presents a high level design concepts of POOL. It starts by defining POOL and stating the reasons for creating POOL. The subsequent section deals with the architectural design approach taken for creating POOL. The last section of this chapter deals with the classes of POOL where the idea behind selecting each class is presented.

3.1 PARASOL Object-Oriented Layer (POOL)

The main reason for creating POOL is to provide an array of abstractions in an object-oriented environment that can be used to model distributed systems applications. POOL is a layer built on top of the PARASOL kernel, this layer basically consists of a C++ class library, where each class represents an abstraction in a distributed system.

POOL was constructed as a "super layer" resting on top of PARASOL, that is, it is a layer that sits between the user and PARASOL (Figure 3.1).

Figure 3.1 POOL Layers

This architecture implies that users are not expected to need adjusting and accessing the underpinnings provided by PARASOL. What users do need access to are the "building blocks" software objects provided by POOL which were built from the stable and "frozen" PARASOL primitives. To summarize, POOL is an object-oriented library of software components that can be used directly or indirectly to prototype applications in a distributed system environment.

3.2 POOL Architecture

3.2.1 POOL & Object-Oriented Design

Just as we are surrounded by objects in real life, a distributed system can also be broken down into basic objects that encapsulates attributes (instances) and behavior

(methods) . Many of the concepts of Object-Oriented programming are currently finding widespread applications in distributed systems. Existing distributed object-oriented systems can be roughly divided into two trends: Systems that do provide a uniform object model and follow the "pure" object-oriented design approach such as Amber [Chase 89] and Emerald [Black 86,87,88], these systems generally provide fine-grain object management along with thread and object mobility. The other trend includes systems that do not provide a uniform object model, such as Argus [Liskov 84, 87, 88], Clouds V2 [Dasgupta 88], and SOS [Shapiro 86,89a,89b] where the model is not "purely" object-oriented and the very low-level operating system layers are implemented typically in a "non-pure" manner.

If the "pure" design approach had been followed in the process of designing POOL, then the approach should also include changing the design of PARASOL so that the whole environment would be a fully general object-oriented one. Although this "pure" approach would make PARASOL congruent with POOL by encapsulating the abstractions that are already in PARASOL, pursuing this approach would create several problems, one of which is that PARASOL is a stable product and making these major changes to it, has the effect of putting PARASOL back in its untested early stages once again.

Furthermore, these changes would go deeper than the encapsulation of PARASOL abstractions, they would involve changes to the internal behavior of the routines in each class. One more reason for not following this approach is that PARASOL is an established library and should remain intact, so any changes should be done in POOL to accommodate the interface with PARASOL.

However, while construction of the run-time support and network communication mechanism in an object-oriented style may well prove to be a suitable choice for POOL, the "object-orientedness" of the final system is not a prime concern in POOL. So, what is

important is that the approach used should encourage the reuse of software and should be sufficiently efficient to cope with the changes in a distributed system design process. The motivation for adopting an object-oriented approach in POOL, therefore, is to bring the reuse advantages of object-oriented programming to the prototyping of distributed application software.

POOL was designed with a different approach, an approach that sees an object-oriented interface to facilitate object-oriented application development built on a distributed operating system. POOL was built on a "conventional" foundation. By choosing PARASOL for this foundation POOL acquired a flexible, simulated, distributed execution environment, and a set of operating system-like primitives for task management, synchronization, and message passing.

This approach suggest a hybrid system in which POOL is developed in an object-oriented manner whereas its underpinnings are developed in a non-object environment. This approach is a common one such as COOL [Habert 90] where object-oriented libraries are making use of existing stable non object systems. Figure 3.2 depicts this common approach.

Figure 3.2 Hybrid Approach

The hybrid approach in figure 3.2 although shows the architectural design of some object-oriented distributed operating systems, a slightly different view of that is relevant to the prototyping environment (POOL) looks like the one in figure 3.3 where this hybrid approach gives a flexible platform in which operating system server building blocks can be developed using PARASOL as underpinnings.

Figure 3.3 POOL Prototyping Environment

So, this architecture reuses the conventional PARASOL code but creates a class library (POOL) which can be used by users in two ways: by either reusing POOL generic classes "as is" to construct object-oriented prototypes, or by deriving sub classes of POOL which more accurately reflect the reality of the prototype target environment. Thus, POOL should be considered a typical object-oriented class library i.e. it should evolve over time as its concepts and ideas mature. On the other hand, its foundation (PARASOL) can be "frozen" to provide stable underpinnings.

3.2.2 The Design Approach

The architectural approach taken for designing POOL was towards constructing a forest of rooted hierarchies library. This organization is a combination of two approaches; a rooted hierarchy approach where all classes are descendants of an abstract root class, and a forest of objects approach where objects are separate entities with no common superclass. POOL's forest of hierarchies approach was adapted since there was no good reason for

grouping the main classes in POOL under one common root, and that is due to the fact that these abstract classes do not support a common responsibility, hence there was no need for that common class.

However, this does not mean that the concept of a hierarchy is obsolete in POOL. Indeed one of the goals in building POOL is to provide an extendible and flexible platform to be used for further enhancements and refinements and this is realized by the fact that all abstractions are generically designed in order to give the prototyper more freedom to design more specialized routines. In fact, POOL was designed as a generic layer in which the concept of expansion is very fundamental. Figure 3.4 shows POOL "extensions". The first extension can be accomplished by adding more generic classes to the current classes of POOL, that is, by using PARASOL primitives, a user can create more abstractions to augment POOL. Indeed POOL does not encompass all building blocks for distributed system applications and sooner or later more abstractions are needed.

Figure 3.4 POOL Extensions

The second extension can be done by creating user defined specialized abstractions based on POOL's generic classes. It is in this area users are expected to derive more abstractions and that is due to the basic functionality that exists in each POOL entity giving a user a simple foundation to build and adapt from.

A different kind of extension can be obtained by creating a layer on top of POOL (Figure 3.5). A user-defined super layer can be built using POOL primitives in the same way POOL is using PARASOL as underpinnings. The NFS example (presented in chapter 5) is a demonstration of such a layer. In summary, the design approach chosen for POOL was no accident since the aim is to allow POOL to evolve and to grow, and the present design format will lead POOL towards that goal.

Figure 3.5 User-Defined Extension Layer

3.3 POOL Classes

An implementation with many classes means that each class encapsulates relatively less of the overall system intelligence [Rumbaugh 91], [Rebecca 90]. Thus, more knowledge is required of anyone trying to understand the application. This is an indication of complexity. Fewer numbers of relatively more intelligent classes typically means a

simpler overall design. There are, however, tradeoffs here. The more intelligence a class encapsulates, the harder it is to reuse the class in other software. Ideally, then, classes should be exactly as intelligent as required to make them generally useful, and applications should have exactly as many classes at this intelligence level as is required for them to function meaningfully. Clearly, human judgment must be applied to make these hazy statements mean something in a specific context. The important point that should be considered when it comes to designing a class library is that prototypers would prefer to see a small class library of directly useful entities than a large library with a correspondingly high learning threshold.

A distributed system prototyper ideally should have all the tools that would facilitate building any prototype of a distributed system. Looking at PARASOL in its present state, we find it provides some basic abstractions to build a distributed system. PARASOL offers some building blocks for emulating some hardware aspects of a distributed system such as computational nodes and communication links.

The major role of PARASOL is in the area of processes management where it provides most of the services pertained to managing processes such as creating, scheduling, and synchronizing processes. However, since PARASOL was designed specifically for performance modeling and not for prototyping, additional distributed systems abstractions are needed at this stage. First, before identifying those needed abstractions, it is important to present the POOL user class `Pool_User`. It is one of POOL's classes which acts as an interface between POOL and user applications. It is viewed as a single-class layer (Figure 3.6) on top of POOL which provides a concise and less-exposed interface to the functions calls of POOL abstractions.

Figure 3.6 Pool_User Class

The **Pool_User** class is also considered as one of the avenues through which a user can adapt and derive from POOL. A user can derive from this class a specialized entity that contains a different set of interfaces. One set of methods a user needs in a prototyping toolkit is a set that deals with file operations such as: **create_file(path_name)**, **open_file(path_name)**, **close_file(fd)**, **delete_file(path_name)**, **read_file(fd,length)** and **write_file(fd,length)**. This implies the creation of a file server class. So the first class we present is the **File_Server** class which provides abstractions for flat file services. A user should find a tool for creating directories under which created files can be grouped. The two basic directory operations that should exist in such a tool are: **create_directory(dir_path)** and **delete_directory(dir_path)**. The **Dir_Server** class was the next class created to provide such abstractions.

The third set of methods that should exist in the toolkit are the methods pertaining to disk operations such as: **disk_read(block)** and **disk_write(block)**. The **Disk** class offers a user the means to prototype a disk service which is concerned with reading and writing disk blocks. The three classes; **File_Server**, **Dir_Server**, and **Disk** are considered as components of a file system abstractions. One important issue in a distributed file service

is how closely the three components are integrated. At one extreme, a file system can have distinct class for disk, file, and directory services. This approach is more flexible, because any user can make use of each entity independently especially if they are designed generically to allow the user to derive from them. At the other extreme, a single class can handle all three components. Clearly this class would be so condensed with many methods to perform many file management functions, this design will not only make it difficult to derive from this class, but also it would prevent a user from prototyping some aspects of distributed system design such as the client-server scheme or the microkernel approach. So it is important to separate these services from each other so that each one can be an independent component that can be located any where in the system. Figure 3.7 depicts a form of a flexible file system.

Figure 3.7 A "flexible" File Services

Having established the importance of having a design that enables each entity to be independently abstracted and used, the need for a communications medium between these entities rises. A prototyper is inclined to use a common communication link (bus) which closely mimics the behavior of a real one. PARASOL provides a simple abstraction for a common bus that connects several computational nodes (machines), however, a more realistic version of PARASOL's bus is desirable. The concept of packetizing messages in PARASOL is very crude, besides, packets are sent contiguously. POOL provides a class

called **Carrier** which offers the functionality that closely mimic the CSMA arbitration style with true message packetizing. Objects from the **Carrier** class would contain the capability to connect any specified number of nodes. In the next chapter, each class will be closely inspected and explained to reveal its user interface. Figure 3.8 depicts the classes that can be found in POOL in its present stage. Indeed many more candidate classes can be suggested and added to POOL. In the final chapter of this report, a discussion about the suggested candidates that can be included into POOL is presented.

Figure 3.8 POOL Classes

Chapter 4

Implementation Issues

The first section of this chapter presents the method adapted to interface POOL and PARASOL. The five subsequent sections will deal with classes of POOL, where each section describes in details the internals workings of each class.

4.1 Interfacing PARASOL with POOL

4.1.1 Justification

The choice for selecting an object-oriented language to develop POOL was really restricted to a very small set of languages. Given that PARASOL was used as the "base" for POOL and it was developed in C, the object-oriented language C++ was a natural choice since it is considered as an evolutionary enhancement to the C language. In fact,

C++ is a superset of the C language. Some may argue that Smalltalk could have been a better choice, however, besides the obvious greater difficulties in integrating a Smalltalk POOL with PARASOL, from a performance prediction point of view, the overheads would make simulation prohibitively expensive with that approach. Although C++ seems like a natural match with the C language, it turned out that the mix between the two languages was not as smooth as suspected.

4.1.2 POOL & PARASOL Integration Method

It is generally regarded as a bad practice to mix procedural and object-oriented code in the same program, and for this reason, hybrid languages such as C++ and Object Pascal have been targeted with some criticism [MASINI 91]. As is usual in the computing field, this criticism has both its merits and its weaknesses. If careful planning is used to mix the two paradigms however, then the criticism is not justified. Unfortunately, the mix of the two paradigms is very often not well planned. It is important at this stage to describe a situation which is relevant to PARASOL and POOL. Referring to PARASOL user's manual, the task management function:

```
SYSCALL    ps_create(name,node,cpu,class_code,priority)
char *name;
int  node;
int  cpu;
void      (*class_code)(void);
int  priority;
```

is used to construct a task on a node. The "class_code" parameter in this function defines a pointer to a parameterless function that returns no value. This parameterless function is a C function that represents the behavior of the task being created. Referring to POOL and

choosing one of its objects, we present the Disk object. A skeleton definition of the disk object that reveals the related items is as follows:

```
class      Disk {
    int    disk_man;      // disk manager task id.
    .
    .

public:
    virtual void disk_manager(void);
    .
    .
};
```

In this class, the method "disk_manager" defines the C++ code that represents the disk manager task. An uninitiated mixed language programmer is thus tempted to use the following code segment :

```
Disk      myDisk;

ps_create("my_disk",1,1,myDisk.disk_manager,1);
```

This approach is incorrect however, as it violates the C++ rule which states that a pointer to a member function in a class can be obtained by applying the address-of the operator (&) to a fully qualified class member name. That is, any pointer to any "true" member function should be created with respect to the object where that function is residing. However, PARASOL's ps_create function is designed to accept a direct pointer to a function and it is not compatible with referencing through objects. This incompatibility is the stumbling block for achieving the integration of PARASOL and POOL, so one of the first tasks before starting developing POOL was to overcome this hurdle and achieve the integration between the two paradigms.

4.1.3 POOL's Wrappers

The most important rule that should be followed when undertaking the integration was not to call true object methods directly with the PARASOL's `ps_create` function. An indirect mechanism was developed to call such methods. For each method in an object that represents a POOL task, a wrapper method is created. This wrapper is an object method, however, it is declared static which makes it lose its right of accessing any member in that object. That is, a static member function is not different from an ordinary C function. So after creating this wrapper function, a message is sent to it which contains an address of the wrapper's object. This address is essential since the wrapper function has no idea of the location of its object, so through a global mechanism, the wrapper would just call the desired method. The following is an example showing the use of a POOL wrapper.

Continuing from the previous example, the following is extracted from the declaration of the Disk class :

```
class      Disk {
    int     disk_man;        // disk manager task id.
    int     disk_ind;        // global disk index

public:
    virtual void disk_manager(void);
    static  void disk_run_manager(void);
};
```

(Notice the addition of the function `disk_run_manager` which is declared as **static**.)

The body of `disk_run_manager` is as follows:

```
void Disk :: disk_run_manager(void)
{
    int         ap;
    int         disk_ind;
    double      ts;
```

```
char      *tp
ps_receive(ps_my_std_port, NEVER, &disk_ind, &ts, &tp, &ap);
disks_list[disk_ind] -> disk_manager();
}
```

Disks_list is a global array of pointers to Disk classes (Figure 4.1), each time a Disk class is created a pointer in the array is set to point to that class and the index of that pointer in the array is stored in the class data section as **disk_ind**. This mechanism is needed for every set of classes in POOL since a global accessing method is necessary so that classes can be easily accessible. The **disk_ind** is obtained during the creation of the Disk class where it is determined by the Disk constructor.

Figure 4.1 Global Indexing Array

An index in array of pointers to classes is passed to the wrapper using the **type** field of PARASOL **ps_send** command.

The previous code segment for creating the function would look like this:

```
disk_man = ps_create("my_disk",1,1,disk_run_manager,1);
ps_resume(disk_man);
ps_send(ps_std_port(disk_man),disk_ind,"",0);
```

Figure 4.2 summarizes the interface between POOL and PARASOL. It shows how POOL calls PARASOL's **ps_create** function to create the wrapper task **disk-run-manager** and then use the function **ps_send** to send the index of the Disk object to the newly created wrapper. Using the index value, the wrapper method would access the Disk object through the global array **disks_list** and call the method **disk_manager**.

Figure 4.2 POOL Meets PARASOL

4.2 The Carrier Class

One of the main reasons distributed systems are becoming more popular is the invention of high-speed local area networks. One of the most widely used networks is the Ethernet local area network. It is used in many distributed systems for its efficiency, low cost and flexibility. The first tool that was introduced to POOL was a class which provided an abstraction similar to the Ethernet network. Although this abstraction does not behave exactly like an Ethernet, it greatly adheres to the Carrier-Sense Multiple Access (CSMA) protocol. This class was named Carrier and it maintains a closeness to the CSMA protocol by allowing random arbitration between contending sending nodes, that is, no centralized control is exercised to determine whose turn it is to access the network. Another characteristic of Carrier is that messages are broken down into packets, so a message logically arrives at its destination only when all of its packets have been transmitted

4.2.1 Carrier.h

The header file "carrier.h" contains the declaration of the Carrier class. The following is only a section of the file which reveals the declaration of the methods. The highlighted entities are the ones that are explained in detail.

```
class Carrier {    // Carrier class

private: // variables
    char      *name;    // Carrier name, although there is
                        // no current use of the Carrier
                        // name, it can be used later as an
                        // object identifier.

protected: // variables
    packet    *head;    // head of the packet
                        // queue
```

```

    int      nqueued;           // # packets queued.
    int      send_tasks[MAX_TASKS]; // this is an array
                                     // used by the task carrier_receive to store
                                     // the number of packets delivered by the
                                     // sender tasks.

public: // variables
    node_info nodes_table[MAX_CAR_NODES]; // each
    // entry in this table contains information about
    // each node on the carrier, information includes
    // carrier_receive id, carrier_transmit id, and node id.
    int      car_ind;           // index of this carrier in
                                     // the global array
                                     // carriers_list.

    int      c_task;           // carrier_task id
    int      packet_size;      // given packet size
    double    trans_rate;       // the delay duration for
                                     // modeling the transmission
                                     // of one byte of data.

    int      nodes_count;      // # nodes connected

private: // methods
    static    void run_car_task(void);
    static    void run_car_transmit(void);
    static    void run_car_receive(void);

protected: // methods
    virtual    void carrier_abort(char *message);
    virtual    SYSCALL car_cache_packet(car_info *clc,int
                                     pn,double ts,int ap);

    virtual    void      carrier_task(void);
    virtual    void      carrier_transmit(int
                                     to_car_ap);

    virtual    void      carrier_receive(void);

```

```

public: // methods
    Carrier(char *name,int node,int host,int nnodes,
             int nodes[],int packet_size,
             double trans_rate,int *cindex);

    //Description of parameters passed to the disk
    //constructor
    //name          :    carrier name
    //node          :    node id the carrier task
    //host          :    cpu to be assigned to carrier task
    //nnodes        :    number of nodes to be connected
    //nodes         :    array of nodes' ids to be connected
    //packet_size   :    carrier's packet size
    //trans_rate    :    transmission rate
    //cindex        :    returned carrier index
    virtual ~Carrier() {}

    virtual SYSCALL carrier_send(
        int port,           // id of the target port
        int snd_node,       // index of sender's node
        int snd_tag,        // a unique tag given by sender
                           // task
        int trgt_node,      // index of target's node
        int msize,         // message size
        int type,          // message type
        char *tp,          // pointer to message text
        int ap);           // acknowledgment port

};

```

4.2.2 The Carrier Constructor

The constructor of the Carrier class connects all the PARASOL nodes specified on the argument list by launching the two tasks; **carrier_transmit** and **carrier_receive** on each node, the two tasks together act like a transceiver on an Ethernet cable. It then

launches the main task of the Carrier class `carrier_task`. The rest of the constructor's job is to record the related data. Figure 4.3 shows how a network would look after creating a Carrier object.

Figure 4.3 The Carrier Constructor

4.2.3 The Transmitter Task

Any process on the node that wishes to send a message across the carrier has to do it through the transmitter task `carrier_transmit`. The transmitter task accepts a message, breaks it into packets, and sends each packet to the `carrier_task`. The protocol followed by `carrier_transmit` works as follows: `carrier_transmit` sends a packet to the `carrier_task`, waits to receive an acknowledgment of the delivery of the packet to the `carrier_receive` task, then it proceeds with the sending of another packet. The previous steps are followed until all packets are sent.

4.2.4 The Carrier Task

Carrier task accepts requests for packets to be transported through the carrier. It inserts each request into the packets' cache which is basically a queue of packets where the order of insertion into the queue is done randomly. The random insertion of the packets into this queue enforces a no order access policy to the carrier. When able to transmit, the carrier task picks the first packet in the queue, acknowledges its carrier_transmit task of its delivery, then sends it to the carrier_receive task on the target node. Figure 4.4 illustrates the behavior of **carrier_task**.

Figure 4.4 The Carrier Task

Although this generic Carrier class adheres to the CSMA protocol, it does not support **true** Carrier-Sense nor Collision Detection except, as mentioned before, through a "random" arbitration mechanism, i.e. while it may not model Ethernet arbitration accurately, it does model to some degree contention delays. The packet queue can be an important measuring tool. It can be used in a simple extension of the Carrier class as a

measuring tool to allow more accurate modeling, i.e. its length could be used to reflect increased processing requirements in `carrier_transmit` task especially if the extension considered `Carrier_Sense` and dealt with Collision Detection.

4.2.5 The Carrier_Send Command

Two methods of the `Carrier` class are available for a user disposal; the `Carrier` constructor (described earlier) which can be used to build a carrier, and `carrier_send` method used to send a message across the carrier. Figure 4.5 shows the complete route a packet travels to get to its destination.

Figure 4.5 Packets Traveling Through the Carrier

- 1) A process sends a message to the transmitter task.

- 2) **Carrier_transmit** breaks the message into packets and sends the packets to **carrier_task**.
- 3) **Carrier_task** receives packets and queue them (see figure 4.4 for carrier's internal behavior).
- 4) **Carrier_task** send packets to **carrier_receive**.
- 5) Packets get grouped into a message by the receiver task.
- 6) Message arrives to its destination.

4.2.6 The Receiver Task

The receiver task **carrier_receive** collects all the packets that are destined for a process on its node, when all the packets of a process are collected, they are grouped into a message and sent to that process. The receiver task is able to handle multiple messages simultaneously since more than one message may be incident on its node at any given time. Besides assembling incoming messages, **carrier_receive** must deliver these to target users using the universal PARASOL **ps_receive** call which is sufficient to access messages sent by **carrier_send** call. At this stage **carrier_receive** does not model the case when all packets of a message are not delivered and consequently the message is not delivered, however, a simple extension that involved adding a method to this generic class might provide handling of such failures.

4.3 The Disk Class

The Disk class provides a tool for prototyping a generic disk drive. A hard disk is an important component in a computer system, it provides a permanent storage area for

data, and it also provides a relatively fast and easy access to the stored data. The disk model is parameterized so that disks with various specifications can be easily prototyped.

4.3.1 Characteristics of a Disk

To prototype a disk drive, we must understand some of the characteristics of a typical disk. A disk is a collection of *platters*. The platters, all of equal size, are permanently mounted in the centers on a spindle (Figure 4.6). Each surface of a platter consists of a collection of concentric rings called *tracks*. The capacity of a track is defined by the number of bytes that can be stored on a track. Furthermore, tracks are divided into small sections called *sectors*. The set of all tracks with the same number is called a *cylinder*.

Figure 4.6 The Mechanical Components of a Disk

4.3.2 Positioning Times

Disk transfers always start at the beginning of a sector. If a multisector transfer crosses a track boundary within a cylinder, no time is lost because switching from one head to another is done electronically. However, if the transfer crosses a cylinder boundary, time will be lost while repositioning the heads to the next cylinder. If the head happens to be positioned over the wrong cylinder, it must first be moved to the cylinder where the requested block is residing. The time elapsed in this case is called *seek* time. Once the head is positioned properly, the controller must wait until the opposite sector has rotated under the head before beginning the transfer. The time wasted waiting for the right sector varies from 0 to the complete rotation time if it just missed. This waiting time is called rotational latency or *latency time*. The settling time is another delay that should be considered, and it is the time the disk controller waits after head movement before reliable operation can begin.

Note that a detailed model of a disk drive, specifically one that explicitly models seek delays by considering arm positioning is required to account for such performance influencing factors as disk arm scheduling, disk fragmentation, disk fragmentation, etc. . It is important to consider all these low-level details for accurate performance prediction.

4.3.3 The Disk Constructor

Part of the declaration of the Disk class looks as follows:

```
class Disk {  
private: // variables
```

```

    char *disk_name;           // disk name. Again, this
                                // variable can be used as an
                                // object identifier.

protected:    // variables

    io_req     *head;          // head of request queue
    io_req     *tail;          // tail of request queue

public:    // variables

    int  disk_ind;              // disk index value in the
                                // global array disks_list
    int  disk_man_port;         // disk manager port
    int  disk_serv_port;        // disk server port
    int  disk_man;              // disk manager id
    int  disk_serv;             // disk server id
    int  disk_msize;            // disk message size
    int  my_node_ind;           // disk node index in the
                                // global nodes array
    int  my_car_ind;            // disk carrier index
    int  heads;                 // number of heads/surfaces
    int  tracks;                // number of tracks/cylinders
    int  sectors;               // number of sectors per track
    int  head_position;         // current disk head position
    int  total_blocks;          // disk size
    int  server_status;         // busy/idle
    double  rotational_delay;    // rotational latency
    double  seek_delay;          // seek/step delay
    double  io_delay;            // transfer delay
    double  set_delay;           // settling delay

private: // methods

    static void disk_run_man(void);
    static void disk_run_serv(void);

protected:    // methods

    virtual SYSCALL  disk_queue_request(int block_id,
                                         int ap,int op,int trgt_node);
    virtual void      disk_abort(char *message);
    virtual void      disk_server(void);
    virtual void      disk_manager(void);

public:    // methods

    Disk(char *name,int node,int cpu,int nh,
          int ns,int nt,double rotd, double sekd,
          double iod, double setd,int car_ind,int
          node_ind,int ms,int *dindex);

```

```

//Description of parameters passed to the disk
//constructor

// name      :    disk name
// node      :    node id for creating disk tasks
// cpu       :    cpu to be assigned to disk tasks
// nh        :    number of heads or surfaces
// ns        :    number of sectors per track
// nt        :    number of tracks or cylinders
// rotd      :    delay value of one disk rotation
// sekd      :    delay value of one track seek
// iod       :    value of a block transfer delay
// setd      :    value of settling delay
// car_ind:    disk's carrier index
// node_ind:    disk's node index
// ms        :    disk message size
// dindex    :    returned disk index

virtual ~Disk() {}

};

```

The Disk constructor records all the disk specifications passed via its parameters and then launches the main tasks; the disk server task `disk_server` and the disk manager task `disk_manager`. The capacity of a disk is calculated using the passed values: number of heads/surfaces (nh), number of sectors per track (ns) and number of tracks (nt) where the maximum capacity of the modeled disk = $nh * ns * nt$.

4.3.4 The Disk Server Task

`Disk_server` simulates the service delay for a block transfer. It only receives messages from `disk_manager`. The request coming from the `disk_manager` specifies the block number. The block number together with the current head position of the disk are sufficient to determine the service delay. The `disk_server` determines the service delay considering seek delay, settling delay, rotational delay, and the delay for the actual data transfer. As mentioned earlier, these low-level delays are important to keep a high degree of

realism for obtaining accurate performance predictions. The service delay is simulated by a call to PARASOL `ps_sleep` routine which causes `disk_server` to sleep for the desired service time, then `disk_server` ends its operation by sending a service completion message to `disk_manager`.

4.3.5 The Disk Manager Task

The task of `disk_manager` is to model the disk interrupt service routine. `Disk_manager` should have a high priority, that is, the priority parameter of PARASOL's `ps_create` function should be set to `MAX_PRIORITY` or a degree lesser. The node where the `disk_manager` task is running should ideally have a preemptive resume scheduling discipline, however, changing the node's scheduling discipline at this stage is difficult without some changes to PARASOL. Hopefully one of the next releases of PARASOL will allow users to change a node's scheduling policy at any stage of the simulation. `Disk_manager` has a bigger job than `disk_server` since it models the interrupt service routine which involves handling current service requests. Other services carried out by the disk manager includes cache simulation where caches in the `Disk` class and in other classes are designed to reflect system level (as opposed to application level) caching. This approach allows either or both (system level and application level caching) but defines only system level caching since applications level is very application dependent.

The following is a simplified version of an algorithm (written in pseudo C++ code) which describes the behavior of `disk_manager` :

```
While (TRUE) {           // while simulation duration
    get_a_request (if any);
    switch(request_type) {
        DISK_SERVER:      // an old i/o request coming from disk_server
            if ( operation was READ )
```

```
        simulate i/o buffer copy
    release the blocked caller;
    if ( a "fresh" request is queued )
        dequeue it;
        start a new i/o request for the disk_server;
    else
        server_status = IDLE;
    break;
READ:          // a user's request for a block read
    if ( server_status == IDLE )
        server_status = BUSY;
        start a new i/o request for the disk_server;
    else        // server_status == BUSY
        queue_request;
    break;
WRITE:         // a user's request for a block write
    if ( server_status == IDLE )
        server_status = BUSY;
        simulate i/o buffer copy
        release the blocked caller;
        start a new i/o request for the disk_server;
    else        // server_status == BUSY
        queue_request;
        release the blocked caller;
        start a new i/o request for the disk_server;
    break;
} // switch
} // while
```

4.4 The Directory Server Class

The directory server class (`Dir_Server`) provides a mechanism for naming files so that they can be stored and accessed conveniently. The `Dir_Server` class provides a

directory structure where each structure contains a number of entries, one entry per file. A directory server is an important component in a distributed system. The primary function of a directory server is to provide a mapping from human-readable names to information about the names. The information may include capabilities, size, location, address, etc. . A directory server should provide some basic operations on directory objects such as lookup, add, and remove. The first operation looks up a directory object name and returns its information, the other two operations add and remove objects from directories. A directory server should allow directories to contain subdirectories, to make it possible for users to group related files together. Subdirectories can contain their own subdirectories, and so on, leading to a tree of directories, often called a directory hierarchy or a hierarchical file system. (Figure 4.7)

Figure 4.7 A Directory Hierarchy

POOL's directory server class `Dir_Server` is designed to provide an abstraction for a directory server in a distributed system. The `Dir_Server` class offers the basic operations of a directory server, and as mentioned in chapter 3, it is more "flexible" having the directory server as a separate entity and not integrating it with the file server. Before

explaining in details the internal workings of the `Dir_Server` class, the next section presents the relationship between a directory server and a name server which may justify why the directory server is constructed as a separate entity.

4.4.1 Name Server Vs. Directory Server

A name server can be a critical entity in a distributed system. It provides a basic task of keeping a database of names, where each name is associated with a location or an address. In one scheme, every time a client runs, on the first attempt to use a server, the client sends a query message to the name server, asking it for the machine address where the server is currently located. Once this address has been obtained, the client can directly send a service request to that server. The basic operations provided by a name server is a kind of a registry service where in a distributed system, each server upon joining the system can send its name and address to the name server to be inserted into the database of the name server. The other important service of the name server is the lookup operation where it looks up a given name and returns the address or location associated with that name.

Referring to POOL, it is clear to see the similarity between the operations of the directory server `Dir_Server` and the operation of a name server. Name and location pair insertion into the name server database is not different from the addition of a directory object to a directory tree of a directory server. The same comparison applies to the lookup operation in both servers. Although POOL does not offer a name server abstraction, since it is not needed in the current stage, it is very easy to derive such abstraction from POOL's `Dir_Server` class.

4.4.2 Dir_Server.h

The header file of the Dir_Server class contains the declaration of the Dir_Server class and other related data structures and data types. The following listing is the declaration of the Dir_Server class.

```

class Dir_Server {      // Directory server class

private: // variables
    char *ds_name;      // directory server name is used as
                        // the name of the file where the
                        // directory tree is saved by the
                        // maintenance function
                        // ds_save_tree.

protected: // variables
    dir      *root;     // pointer to the root directory

public: // variables
    int      my_node_ind; // dir. server node index
                        // in the global nodes
                        // table
    int      my_car_ind;  // dir. server carrier
                        // index in the global
                        // carriers_list array
    int      ds_message_size; // dir. server message
                        // size
    int      ds_id;       // dir. server task id.
    int      ds_index;    // dir. server index
                        // value in the global
                        // table dir_servers_list
    int      ds_port;     // dir. server task port
    double   ds_delay;    // dir. server functional
                        // delay

private: // methods
    static void ds_run_task(void);

protected: // methods
    virtual void ds_task(void);
    virtual entry* ds_find_entry(dir *dp,
                                char *entry_name);
    virtual void ds_get_top(char *path,char
                            *rest,char* dname);
    virtual void ds_abort(char *message);
    virtual void ds_save_tree(long sp,dir *dp,FILE
                                *fp);

```

```

        virtual    dir*        ds_build_tree(long sp, FILE *fp);
public:    // methods
        Dir_Server(char *name, int node, int cpu, int car_ind,
                    int node_ind, int ms, int *index);
        virtual    ~Dir_Server(void)
        virtual    SYSCALL    ds_add_entry(char *dpath,
                                           char *ename,
                                           int type, int fd);

        virtual    SYSCALL    ds_remove_entry(char *dpath,
                                           char *ename);

        virtual    SYSCALL    ds_get_fd(char *path, char *name);
        virtual    SYSCALL    ds_backup_tree(void);
        virtual    SYSCALL    ds_restore_tree(void);
        virtual    dir*        ds_find_dir(char *dname);
        virtual    entry*      ds_get_entry(dir *dp);

};

```

4.4.3 Internal Behavior

The constructor of the `Dir_Server` class performs one main task and that is launching the directory server task `ds_task`, the rest is only recording of information passed through its parameters. `Ds_task` accepts requests sent by users and accordingly calls the appropriate function. If the request is for creating a new entry in the directory tree, `ds_task` would call the method `ds_add_entry` supplying it with an entry name and a directory path. The other two possible requests are for deleting an entry where the method `ds_remove_entry` is called, and for retrieving a file descriptor of an entry, in this case

the method `ds_get_fd` is called. Two other useful methods exist in the `Dir_Server` class; `ds_backup_tree` and `ds_restore_tree` which are used as maintenance operations for backing up a prototype of a directory and for restoring a backup of a directory. These operations can save a good deal of effort in initializing a prototyping environment.

4.5 The File_Server Class

The `File_Server` Class is the class that provides the flat file services by offering its clients with an abstraction consisting of files, each of which is a linear sequence of blocks. The methods of the `File_Server` class provide operations that include creating or deleting files and reading or writing files. The `File_Server` class depends on the other two classes, the `Dir_Server` class and the `Disk` class, for the completion of its operations and that is because the physical file is stored on a disk and a file or a directory is stored under some directory in the directory tree.

4.5.1 File_Server.h

The following is part of the `File_Server` class declaration:

```
class File_Server {    // File server class

private:
    char        *fs_name;        // file server name is used to
                                // name the file where the
                                // inode_table of the file
                                // system is stored

protected:
    cache        fs_cache[MAX_CACHE]; // fs cache table
    fs_req       *head;              // head of fs requests
    fs_req       *tail;              // tail of fs requests
    block        *block_fl;          // blocks free list
```

```

public:
    int      fs_index;          // file server global index
    int      fs_port;          // file server shared port
    int      files_created;    // # of curr. created files
    double   fs_cache_hit_delay;
    double   fs_functional_delay;
    int      current_disk_size;
    int      current_cache_index;
    int      disk_size;        // for each disk
    int      my_node_ind;      // fs node index
    int      my_car_ind;       // carrier index
    int      disk_man_port;    // disk manager port
    int      disk_node;        // disk node
    int      dir_ser_port;     // dir. server port
    int      dir_ser_node;     // dir. server node
    int      net_mess_size;    // network message size
    int      dir_ser_ind;      // dir. server index
    inode_tab fs_inode_table[MAX_FILES];
    // The fs_inode_table is used to store information about
    // each created file in the file system, these
    // information include file type, file size, file mode
    // and most important a pointer to the file's data
    // blocks.

private:    // methods
    static void fs_run_task(void);

protected: // methods
    virtual ~File_Server() {}
    virtual SYSCALL fs_queue_request(cl_to_fs *cfs,
                                     int type, int ap);

    virtual void fs_task(void);
    virtual void fs_get_name(char *path, char *rest,
                             char * name);
    virtual int fs_cache_exist(int disk_id,
                              int block_id);
    virtual block* fs_get_block(void);
    virtual SYSCALL fs_cache_insert(int disk_id,
                                    int block_id, double ts);
    virtual void fs_abort(char *message);
    virtual SYSCALL fs_backup_inode_table(void);
    virtual SYSCALL fs_restore_inode_table(void);

public:    // methods
    File_Server(char *name, int node, int cpu, int nthreads,
               int car_ind, int node_ind, int disk_ind,
               int ds_ind, int ms, double chd, int *index);
    virtual SYSCALL fs_file_open(char *name, int mode);
    virtual SYSCALL fs_file_close(int fd);

```

```

virtual    SYSCALL    fs_file_flush(int fd);
virtual    SYSCALL    fs_file_create(char *path,int mode);
virtual    SYSCALL    fs_file_delete(char *name);
virtual    SYSCALL    fs_file_read(int fd,int length,
                                int fp);
virtual    SYSCALL    fs_file_write(int fd,int length,
                                int fp);
virtual    SYSCALL    fs_file_seek(int fd,int offset,
                                int relative_to,int fp);
virtual    SYSCALL    fs_dir_create(char *path);
virtual    SYSCALL    fs_dir_delete(char *path);
virtual    SYSCALL    fs_disk_read(int disk_id,int
                                block_index);
virtual    SYSCALL    fs_disk_write(int disk_id,
                                int block_index);
virtual    void        fs_restore(void);
virtual    void        fs_backup(void);

};

```

4.5.2 The File_Server Constructor

It is very important in this section to describe in a little detail the role of the `file_server` constructor as it is the method a user would call first to create the file server. The most important steps a user should carry out before constructing a file server are to construct the directory server and the disk class, since the `File_Server` constructor is designed to expect the addresses of these servers, so they should exist before the invocation of the file server constructor.

The following list defines the parameters of the `File_Server` constructor:

```

name       :    name of the file server
node       :    node id where fs_task would run
cpu        :    cpu that should be assigned to fs_task
nthreads   :    number of file server threads
car_ind    :    index of file server's carrier
node_ind   :    index of file server's node
disk_ind   :    index of disk attached to file server

```

<code>ds_ind</code>	:	index of directory server
<code>ms</code>	:	file server message size
<code>chd</code>	:	cache hit delay value
<code>index</code>	:	file server index to be returned

Nthreads is the parameter that specifies the number of file server threads, that is, the file server task can be in one or more instances and this is done by the cloning of the `fs_task`.

4.5.3 The File_Server Thread(s)

The file server task is multithreaded to improve its efficiency. Any of the file server threads works by receiving a request from a user and after appending the request to the queue of requests `fs_task` picks a request from the head of the queue and checks its type and accordingly it calls the appropriate method. Most of the methods in the `File_Server` class are designed to be used by `fs_task(s)`, however, the methods in `Pool_User` class (will be described in the next section) are the ones that users are supposed to invoke in order to get the desired services. Figure 4.8 depicts the behavior of the file server task(s) `fs_task`.

Figure 4.8 Fs_task Behavior

- 1) Receives a request from a user.
- 2) Appends the request to the queue of requests.
- 3) Picks a request from the head of the queue.
- 4) Invokes the appropriate method.
- 5) Ends the service by returning the result to the user.

4.6 The Pool_User Class

Users of POOL should start with the Pool_User class in order to use any of the operations provided by POOL. The Pool_User class contains a method for each operation that a POOL server provide, these methods introduce the user to POOL operations by offering a less-exposed system function calls that resemble the UNIX system operations. In this section, a listing of the class declaration is presented, and since this class is the user

interface class, appendix A is designated as a users' manual describing each of the method in this class.

The Pool_User class is a template for user applications tasks embodying all of the user interfaces to other POOL objects. In appendix B, a listing of the code that shows one usage of the Pool_User class. Appendix B presents the Disk_Tester class which is a subclass of the Pool_User class and contains a task that accesses the Disk object and tests it by reading and writing some data blocks.

```
class Pool_User{
public:
char      *p_name;           // POOL user name
int       user_ind;          // POOL user global index
int       my_node_ind;       // fs node index
int       car_ind;           // carrier index
int       net_mess_size;     // network message size
int       disk_ind;          // user disk index
int       ds_ind;            // user dir. server index
int       fs_ind;            // user file server index
int       nfs_ind;           // user NFS index

open_file p_open_files_table[MAX_OPEN_FILES];

    Pool_User(char *name,int car_ind,int node_ind,
               int fs_ind,int ds_ind,int disk_ind,
               int nfs_ind,int ms,int *index);

    virtual ~Pool_User() {}
    virtual void pool_abort(char *mess);
    virtual void pool_get_name(char *path,char *rpath,
                               char *name);

    virtual SYSCALL pool_carrier_send(int port,
                                       int snd_node,
                                       int snd_tag,
                                       int trgt_node,
                                       int msize,
                                       int type,
                                       char *tp,
                                       int ap);

    virtual SYSCALL pool_disk_read(int disk,int block);
```



```
virtual SYSCALL pool_disk_write(int disk,int block);

virtual SYSCALL pool_dir_create(char *path);
virtual SYSCALL pool_dir_delete(char *path);

virtual SYSCALL pool_fs_file_open(char *path,
                                   int mode);
virtual SYSCALL pool_fs_file_close(int fd);
virtual SYSCALL pool_fs_file_create(char *path,
                                   int mode);
virtual SYSCALL pool_fs_file_delete(char *path);
virtual SYSCALL pool_fs_file_read(int fd,
                                   int length);
virtual SYSCALL pool_fs_file_write(int fd,
                                   int length);
virtual SYSCALL pool_fs_file_seek(int fd,int offset,
                                   int rel_to);
virtual SYSCALL pool_fs_dir_create(char *path);
virtual SYSCALL pool_fs_dir_delete(char *path);

virtual SYSCALL pool_nfs_mount(char *rdir,
                               char *ldir);
virtual SYSCALL pool_nfs_file_open(char *path,
                                   int mode);
virtual SYSCALL pool_nfs_file_close(int fd);
virtual SYSCALL pool_nfs_file_write(int fd,
                                   int length);
virtual SYSCALL pool_nfs_file_read(int fd,
                                   int length);
virtual SYSCALL pool_nfs_file_seek(int fd,int offset,
                                   int rel_to);
};
```

Chapter 5

Network File System Prototype

This chapter presents an example of using POOL in building a prototype of Sun's Network File System (NFS). The first section includes a description of NFS's general characteristics. Subsequent sections are devoted to POOL's NFS class where the contents of the class declaration are listed first, the `nfs_mount` function is described next, then each subsection that follows describes the behavior of the main tasks, namely `system_calls` task, `client` task and `server` task. The main emphasis of this chapter is to show how existing POOL classes are used to realize this prototype. Regarding this issue, a comment about POOL usage will be mentioned wherever appropriate.

5.1 Background

One of the most widely used network file systems is Sun Microsystem's Network File System, universally known as NFS. It was originally designed and implemented by

Sun Microsystems for use on its UNIX-based workstations. Other manufacturers now support as well, for both UNIX and other operating systems. NFS supports heterogeneous systems, for example, MS-DOS clients making use of UNIX servers. NFS was chosen to demonstrate the use of POOL abstractions since each one is needed in the process of building the NFS prototype. The basic idea behind NFS is to provide a facility for sharing files in a heterogeneous environment of networks and computer systems. One of the basic characteristics of NFS is providing a mechanism to mount remote directories. NFS clients should get access to exported directories by mounting them in their local hierarchies.

Figure 5.1 Mounting Operation

When a client mounts a remote directory it becomes part of its directory hierarchy. The mount operation shown in Figure 5.1 depicts how directory `a` in machine A is mounted at mount point (directory) `b` in machine B. Thus the path `/b/a/c` in machine B refers to the same file as `/a/c` when referenced in machine A. Mounting remote directories

on top of local directories results in a file system that is partly local and partly remote. Another basic feature provided by NFS is directory and file access. Local clients can send messages to remote servers to manipulate directories and to read and write files.

NFS allows every machine to be both a client and a server at the same time. It is relevant in this section to take a quick look at the details of Sun's implementation of NFS since POOL's NFS class was developed in a much similar way. Figure 5.2 shows the architectural design of Sun NFS, it consists of three layers.

Figure 5.2 NFS Layers

The top layer is the system calls layer which handles users calls for file operations like open, read and write. After receiving the call and checking the parameters, it invokes the second layer, the Virtual File System (VFS) layer. VFS maintains a table of entries with one entry for each mounted file, the table is referred to as vnode (virtual inode) table, the main use of the vnode table is to tell whether a file is local or remote. Vnode entries are constructed for a remote directory after it has been mounted. The third layer is the client

layer where the client code maintains another table called rnode table which consists of rnode (remote inode) entries. Each entry in the rnode table holds a file handle of a remote file. Ultimately, each entry in the vnode table should contain a pointer either to an rnode entry in the NFS client code or to an inode entry in the local file system. NFS server is a stateless server, that is, no table of entries is maintained on its side. However, the NFS server is prepared to provide file information (file handle) upon request. It also uses file handles sent to it for accessing files on its local file system.

Caching is one of the techniques used to improve performance in computer systems. An NFS client maintains a cache of remote files data. When a file block is needed, a check is first made to see if the request can be satisfied out of the client's cache, if so, network traffic can be avoided then. POOL's prototype of NFS reflects all the details previously explained which next sections deal with.

5.2 POOL NFS Class

The idea behind creating an NFS prototype is to show how POOL's abstractions are used to prototype such an application. The NFS model is built using the primitives of POOL, it can be considered as a layer built on top of POOL and using POOL as underpinnings (Figure 5.3). The NFS class offers users some methods to mimic the behavior of an NFS such as mounting remote directories and remote file access. A caching technique is also implemented in the client's code.

Figure 5.3 POOL NFS Layer

The following section shows the declaration of the NFS class.

```
// -----  
// This is a definition of the constants used by the various  
// methods in the NFS class.  
  
#define MAX_CLIENT_CACHE 100 // Max. client cache entries  
#define NFS_BLOCK_SIZE 8192 // NFS block == 8K  
#define NFS_MOUNT 1 // mount operation constant  
#define NFS_OPEN_FILE 2 // open_file constant  
#define NFS_CLOSE_FILE 3 // close_file constant  
#define NFS_READ_FILE 4 // read_file constant  
#define NFS_WRITE_FILE 5 // write_file constant  
#define NFS_SEEK_FILE 6 // seek_file constant  
#define NFS_REMOTE 0 // a constant for remote files  
#define NFS_LOCAL 1 // a constant for local files  
#define NFS_CLIENT 1 // client task constant  
#define NFS_SERVER 2 // server task constant  
#define NFS_USER 3 // NFS user constant  
#define MAX_VNODES 100 // Max. entries in vnode table  
#define MAX_RNODES 100 // max. entries in rnode table  
  
// -----  
// This structure is used to represent an entry in the rnode  
// table nfs_rnode_table  
  
typedef struct rnode {  
    int type; // for marking entry availability  
    int rfd; // remote file descriptor  
} rnode;
```

```
// -----
// This struct represents an entry in the client cache table
// which is referred to as nfs_client_cache

typedef struct c_cache {
    int fd;          // file descriptor
    int fp;          // file position for flushing operation
    int block;       // number of cached block
    int launched;    // a flag indicating if nfs_cache_flusher
                    // task is launched or not
    double ts;       // data block insertion time
} c_cache;

// -----
// This structure is used to represent an entry in the vnode
// table nfs_vnode_table

typedef struct vnode {
    int location;    // REMOTE/LOCAL
    int rfd;         // fd in nfs_rnode_table
    int ifd;         // local fd
    int type;        // file/dir
    int mode;        // file mode
} vnode;

// -----
/* The following structure is used as information carrier between NFS and its users. Since
PARASOL ps_send function has a limited number of arguments, and a bundle of
information have to be sent back and forth between NFS and its users, this "work-around"
was developed, which is basically making a use of the pointer to a character parameter
```

char *tp in the PARASOL function **ps_send** where a pointer to the **nfs_req** structure is passed to that parameter after it's been appropriately casted.*/

```
typedef struct nfs_req {
    int op;           // operation type
    int type;         // message type
    char *rpath;      // remote complete path
    char *lpath;      // local complete path
    int fd;           // file descriptor
    int len;          // length of read/write
    int mode;         // file mode
    int fp;           // user file position
    int offset;       // for seek operation
    int relative_to;  // for seek operation
    int reply_port;   // user reply port
    int reply_node;   // user reply node
} nfs_req;

// -----

class NFS {
public:
    char *nfs_name;   // NFS name
    int nfs_index;    // NFS global index
    int nfs_client_port; // Client shared port
    int nfs_server_port; // Server shared port
    int nfs_user_port; // User shared port
}
```

```

int          vnodes_created;      // # created vnodes
int          my_node_ind;        // nfs node index
int          my_car_ind;         // carrier index
int          net_mess_size;      // network message size
int          local_fs_index;     // NFS local file system
int          local_ds_index;     // NFS local dir. server
int          rnfs_ind;           // remote nfs index
double       nfs_cache_hit_delay;
double       nfs_client_functional_delay;
double       nfs_server_functional_delay;
double       nfs_functional_delay;
int          current_cache_index;
vnode        nfs_vnode_table[MAX_VNODES];
rnode        nfs_rnode_table[MAX_RNODES];
c_cache      nfs_client_cache[MAX_CLIENT_CACHE];

    NFS( char *name,             /* given name                */
        int node,               /* node id of NFS tasks      */
        int cpu,               /* host id of NFS tasks      */
        int scnoth,            /* system_calls # threads    */
        int cnoth,             /* client # threads           */
        int snoth,             /* server # threads           */
        int car_ind,           /* NFS carrier index          */
        int node_ind,          /* NFS node index             */
        int ms,                /* NFS message size           */
        int lfs_ind,           /* NFS local file system index */
        double cfd,            /* client functional delay    */
        *

```

```
double sfd,      /* server functional delay */
double chd,      /* cache hit delay */
int *index       /* NFS returned index */

);

virtual void      nfs_client(void);
virtual void      nfs_server(void);
virtual void      nfs_system_calls(void);
virtual SYSCALL   nfs_mount(char *remote_dir,
                           char *local_dir);
virtual SYSCALL   nfs_file_open(char *path,int mode);
virtual SYSCALL   nfs_file_write(int fd,int len,int fp);
virtual SYSCALL   nfs_file_read(int fd,int len,int fp);
virtual SYSCALL   nfs_file_close(int fd);
virtual SYSCALL   nfs_file_seek(int fd,int offset,
                           int rel_to,int fp);
virtual SYSCALL   nfs_file_flush(int fd);
virtual void      nfs_traverse_tree(dir *rdp,dir *ldp);
virtual void      nfs_cache_flusher(int op,int fd,int len,
                           int fp,int ind);
virtual SYSCALL   nfs_client_server_intrfc(int op,
                           int fd,int len,int fp,char *path);
virtual void      nfs_abort(char *message);
virtual SYSCALL   nfs_cache_insert(int fd,int fb,
                           double ts,int fp);
virtual SYSCALL   nfs_cache_exist(int fd,int fb);
virtual ~NFS() {}
```

```

static void    nfs_run_client(void);
static void    nfs_run_server(void);
static void    nfs_run_user_intrfc(void);
static void    nfs_run_cache_flusher(void);
};

// -----

```

5.2.1 NFS Constructor

The NFS constructor method is the first function a user calls to create an NFS prototype. Before invoking the NFS constructor method, the user should have created local file systems for each NFS class. This is the first interaction between NFS components and POOL components. Each file system becomes the NFS file system so that the NFS can export one or more of its directories for access by remote clients. On the other side NFS users can mount exported directories. The main task of the NFS constructor after initializing the variables and recording the given data is launching the NFS tasks; **system_calls**, **client** and **server**. These tasks are multithreaded where the number of threads for each task is static and must be determined by the user through the constructor parameters *snoth*, *cnoth* and *cnnoth*. So this is a static approach to determine the number of long-lived threads to be created.

There are two more alternative approaches for determining the number of threads: the first one is to create long-lived threads dynamically, that is threads are created only when needed, and the second approach is also a dynamic one, however, threads are short-lived, that is, a thread is destructed when it's done its job. The latter approach incurs heavy

overheads as task(thread) creation and destruction is expensive. The former approach would have been an efficient alternative. In reality, even this approach would require limits, however, so the very first approach, the approach currently implemented, is realistic and slightly easier to implement.

5.2.2 System_Calls Task

The function of this task is to accept requests from NFS users, so that it mimics the function of the system calls layer. Users invoking NFS calls should be local. That is, they should reside on the same node as the NFS. The basic behavior of this task is [1)receives a request, 2)invokes the appropriate function in the VFS layer and blocks waiting for the result, and finally 3)returns the result to the user.] When the task blocks waiting for the result, it can not receive any other request, for this reason system_calls task is multithreaded to model some parallelism so that efficiency can be achieved. All the threads of the system_calls task receive their requests on one shared port. Figure 5.4 depicts the behavior of the system_calls task.

Figure 5.4 The System Calls Layer

The following is a pseudo C++ code that describes the behavior of the `system_calls` task. This pseudo code describes the complete functionality of the `system_calls` task, that is, the functionality of the VFS layer is also included.

```

While (TRUE) {      // while the duration of the simulation

    • receive a request on the shared port from the local user.
    • switch(received request type) {

        NFS_REMOTE:      // request is for remote file i/o operation
            • send_request_to(local NFS client)
            • receive result on standard port (private port)
        NFS_LOCAL:       // request for local file i/o operation
            • send_request_to(local file server)
            • receive result on standard port (private port)
    }      // switch
    /* At this stage the result of the operation is returned and system_calls
    should reply to the user with the result */
    • send_result_to(blocked local user)
} // while

```

5.2.3 VFS Layer

The Virtual File System layer (Figure 5.5) consists of the functions `nfs_mount`, `nfs_file_open`, `nfs_file_close`, `nfs_file_read`, `nfs_file_write`, and `nfs_file_seek`. In each of these functions the file handle is examined whether the file site is remote or local, if it is remote then a message is sent to the client task, otherwise the request is satisfied by calling the appropriate function in the local file system. Most of the NFS operations are performed by the functions in the VFS layer which includes reading

from a file and writing to a file where both functions use a caching mechanism for accessing data. Stored in the cache are some data blocks of the remote files, when a request for reading from a file arrives, the cache is first checked if the data block exists in the cache, if so the operation is completed without the need for sending a request to the remote server. For writing to files the behavior is different. Any data to be written must be inserted into the cache using the method `nfs_cache_insert`.

Figure 5.5 VFS Layer

The insertion function `nfs_cache_insert` first checks if there is an available entry, if no space available, a least recently used entry must be removed. The `nfs_cache_insert` function must deal with the incoherence of the cache; suppose two clients are both caching the same file block, and one of them modifies it. When the other one reads the block, it gets the old value. To solve this problem, upon receiving a request for cache insertion, `nfs_cache_insert` launches a task called `nfs_cache_flusher` that would act like a timer by sleeping for a limited time and then flushes the cache within that time. A flag is switched on during that "flushing" time so that any subsequent cache write before the flushing is over are ignored. The other cache incoherency which this implementation does not address is the invalidation of other client caches. The NFS server, after receiving a request for

writing a block of data, should send a message to all clients who hold a copy of that data block so that they update it. The following is a pseudo code of `nfs_cache_flusher` :

```
nfs_cache_flusher() {  
    • sleep for a pre specified time  
    • invoke the client_server interface routine nfs_client_server_intrfc  
    /* nfs_client_server_intrfc is given all the information needed to send the block of  
    data to be written into the remote file system. */  
    • kill itself  
}
```

5.2.4 Client Task

The client task behaves in a way similar to the `system_calls` task where it is a multithreaded task and its threads share a common port. The client task [1] receives a request from the VFS layer, 2) calls the function `nfs_client_server_intrfc` and blocks waiting for the result, and finally, 3) returns the result to the upper layer.]

The interface function `nfs_client_server_intrfc` packages the request, and then sends it to the server task. For efficiency reasons, transfers between client and server are done in large chunks, normally 8192 bytes, even if fewer bytes are requested. These are sent across the carrier via the Carrier class method `carrier_send`. `Nfs_client_server_intrfc` blocks waiting for the result from the server task, then it sends the result to the private port of the calling thread. Figure 5.6 gives a graphic description of the steps.

Figure 5.6 The Client Layer

The following is a pseudo C++ code that describes the behavior of the client task:

```

While (TRUE) {      // while duration of simulation
    • receive a request on the shared port.
    switch(request type) {
        NFS_OPEN_FILE:
            • send a request for a file handle to the remote NFS server
            • receive file handle on private port
            • send file handle to local system_calls task.
        NFS_READ_FILE:
            • if data is not available in the local cache {
                • send to remote NFS server for the requested data block
                • receive block requested on private port
                • insert block into local cache
            }
            • send result to local system_calls task.
        NFS_WRITE_FILE:
            • insert block into cache
    }
}

```



```
        // nfs_cache_insert function handles the insertion of data
        // blocks besides the flushing of blocks through launching of
        // nfs_cache_flusher tasks.
        • send result to local system_calls task.
NFS_SEEK_FILE:
        • send a request for file handle to remote NFS server
        • return result to local system_calls task
NFS_CLOSE_FILE:
        • flush all cached data blocks
        • send result to local system_calls task
NFS_MOUNT:
        • send a request for file handle to remote NFS server
        • send result to local system_calls task
    } // switch
} // while
```

5.2.5 Server Task

The multithreaded server task has one main job only and that is to satisfy requests coming from the client task by invoking the appropriate function in the local file system, and returns the result to the calling client task. Figure 5.7 shows the steps the server task takes to do its job where it [1) receives a request on the shared port, 2) calls the appropriate function from the local file system and finally 3) returns the result to the client task.] Server caching can exist in real NFS, this model does not support caching at the server side, however an adaptation from this model can incorporate server caching to increase the overall efficiency of the NFS model.

Figure 5.7 NFS Server

The following is a pseudo C++-like code of the server task:

```
While (TRUE) {
```

- receive a request on the shared port.

```
switch(operation) {
```

```
    NFS_OPEN_FILE: // opening an NFS file
```

```
        call fs_file_open() // in local file system
```

```
    NFS_CLOSE_FILE: // closing a file
```

```
        call fs_file_close() // in local file system
```

```
    NFS_READ_FILE: // reading from a file
```

```
        call fs_file_read() // in local file system
```

```
    NFS_WRITE_FILE: // writing to a file
```

```
        call fs_file_write() // in local file system
```

```
    NFS_SEEK_FILE: // file seek operation
```

```
        call fs_file_seek() // in local file system
```

```
} // switch
```

- send the result to the client via `carrier_send()`

```
// while
```

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Chapter 6

Conclusions

This chapter starts with a review of the goals and objectives of this thesis. Next it discusses prototyping performance where it shows a set of results obtained from testing one of the tools in POOL. The rest of this chapter identifies future work that can be undertaken to extend POOL.

6.1 Summary

Developing software for distributed system applications is a difficult process. Rarely is there a single correct solution, so developers rely on intuition and experience when generating potential designs. The prototyping approach to system design is ideal for constructing distributed systems. By observing the behavior of many potential designs before making commitment, the best one can be selected, thereby reducing the number of problems encountered during maintenance. In order to make experimenting with designs an

inexpensive and effective endeavor, developers must be provided tools to assist this process. This was one of the primary objectives of this project which were:

- 1) To augment PARASOL with a set of distributed system abstractions with which to facilitate software development through prototyping, i.e. to construct POOL (the core of a prototyping environment). By using PARASOL as a basis for a prototyping tool, it was possible to take advantage of its abilities, viz., providing user-defined emulated execution environments together with realistic user-defined executable models of system and application software. These strengths were augmented by POOL's abstractions which simultaneously added to the realism of the prototyping environment.
- 2) To demonstrate the value of POOL through example, namely: the Network File Server (NFS) prototype that was presented in chapter 5 which demonstrated the usage of POOL in building a prototype of a well-known distributed application.
- 3) To provide these extensions to PARASOL in the form of an upper-level layer as a C++ class library (viz. POOL), and thereby promote efficient prototyping through code reuse.

The integration of the conventional C-based PARASOL with the object-oriented C++-based POOL was highly successful and retained very good execution efficiency as shown in the following section.

6.2 Performance

An important aspect of any new system components is an investigation and analysis into the way the components interact and respond with the system. Although functional

testing was conducted on the system to verify its correct operation, its performance can also be important. The performance of POOL may best be assessed by comparing it with other similar prototyping toolkits. However, this option was not possible since such toolkits that carry relevant features are unattainable. The other option followed was to compare the performance of the POOL Carrier class with the built-in PARASOL bus abstraction.

Despite the fact POOL was built on top of a simulator (PARASOL), the results obtained after conducting the test were quite impressive. The test conducted involved a network of 100 nodes where each node uses the medium connecting these nodes to send a specified number of messages. The test was done with the same configuration but with two different connection medium. One environment contains 100 nodes connected by PARASOL bus medium and the other uses POOL Carrier medium.

Table 6.1 shows the results from testing the two systems. The results show a slight additional overhead incurred using the POOL Carrier over the PARASOL bus. However, this performance penalty is not substantial (of the order of 30%) and would be acceptable, as performance is generally much less important in prototyping than it is in simulation.

System	Total number of messages sent				
	5000	10000	15000	20000	30000
PARASOL Bus	6.2	9.7	13.7	17.5	25.1
POOL Carrier	7.5	12.9	17.9	22.8	32.9

Table 6.1 Comparing Bus & Carrier Total Execution Times(seconds)

6.3 Future Work

One of the main goals of this thesis is to provide a prototyping environment for modeling and building prototypes of distributed systems. The toolkit developed is only a start and many abstraction are needed to augment the prototyping toolkit.

The modular approach followed in designing and developing POOL provides a framework for the development of future abstractions. Micro-kernel based distributed operating system are more superior to monolithic kernel systems and many modern operating systems were designed using this paradigm such as Amoeba. This trend can be applied to the POOL and PARASOL (Figure 6.1) where the microkernel role can be played by parasol since it provides the minimum operating system services which include task management and inter-task communication. The role of the servers can be played by the entities currently in POOL and new entities added to POOL.

Figure 6.1 Microkernel Prototyping Environment

POOL can be augmented with many servers and abstractions, the list of candidate classes can include classes that deal with networking issues such as gateway servers and server dealing with store and forward in networks.

Other additions can be in the area of RPC and communication streams and network forwarding server. Besides fresh classes that can be added to POOL, existing classes offers a base for adapting more specialized abstractions, for example, the file server class can be used as a super class for deriving file servers such a bullet file server or an immutable file server. Load balancing server, pipe and socket servers, and log servers are also candidate entities for POOL extensions.

Appendix A

Pool_User Class Users' Manual

This appendix lists the methods of the class Pool_User. This POOL class, first introduced chapter 3, is the class that sits on top of POOL and acts as an interface layer for POOL users. The appendix is divided into five sections where each section includes descriptions of the methods related to the classes Carrier, Disk, Directory, File_Server, and Network File Server (NFS). The Pool_User class offers users an interface layer to POOL abstractions that hide many details of POOL entities and attempt to model UNIX file system semantics in UNIX-related classes cases.

A.1 The Carrier Class

NAME:

Pool_User :: pool_carrier_send

DECLARATION:

```
virtual SYSCALL    pool_carrier_send(int port,int snd_node,int snd_tag,  
                                     int trgt_node,int msize,int type,char *tp,int ap);
```

DESCRIPTION:

This method is used for sending a message to the port identified as *port*. The sender node should supply its node index *snd_node*, and should supply a unique id value *snd_tag* (task id will do). The target node index *trgt_node* should also be given. The rest of the arguments are concerned with the message where message size *msize*, message type *type*, and pointer to message text *tp* should all be supplied by the sender. Finally, the sender may supply a port id for acknowledgment called *ap*. Successful operation is indicated by a returned value of OK; otherwise SYSERR is returned.

A.2 The Disk class**NAME:**

```
Pool_User :: pool_disk_read
```

DECLARATION:

```
virtual SYSCALL    pool_disk_read(int disk, int block);
```

DESCRIPTION:

A user should use this method for reading blocks of data from the disk. The user should provide the disk id *disk* and the desired block index *block*. Operation returns OK if successful; SYSERR otherwise.

NAME:

```
Pool_User :: pool_disk_write
```

DECLARATION:

```
virtual SYSCALL    pool_disk_write(int disk, int block);
```

DESCRIPTION:

This method is similar to *pool_disk_read* except it is used for writing blocks of data to the disk.

A.3 The Dir_Server Class

NAME:

Pool_User :: pool_dir_create

DECLARATION:

virtual SYSCALL pool_dir_create(char *path);

DESCRIPTION:

This method is used to create a directory, the directory can even be the root directory, that is , a user can start with this function to create a new directory tree. The single parameter *path* should represents the full path starting from the root dir. and should contain the directory name at its tail. The operation returns SYSERR in case of a failure, OK otherwise.

NAME:

Pool_User :: pool_dir_delete

DECLARATION:

virtual SYSCALL pool_dir_delete(char *path);

DESCRIPTION:

This method is used to delete a directory. The single parameter *path* should contain the directory name and should be relative to the root. The operation returns SYSERR in case of a failure, OK otherwise.

A.4 The File_Server Class

NAME:

Pool_User :: pool_fs_file_create

DECLARATION:

virtual SYSCALL pool_fs_file_create(char *path, int mode);

DESCRIPTION:

This function is used to create a file. The name of the file to be created must be present at the end of *path* which is the absolute path, the parameter *mode* is used to specify the creation mode of the file; O_RDONLY for read only files, O_WRONLY for write only

files, and O_RDWR for files that are created for reading and writing. Operation returns file descriptor of the create file, SYSERR otherwise.

NAME:

Pool_User :: pool_fs_file_open

DECLARATION:

virtual SYSCALL pool_fs_file_open(char *path, int mode);

DESCRIPTION:

This is a file server function which is used for opening a previously created file. The file name should exist at the end of parameter *path*. a user should also supply the file open *mode*. to specify the creation mode of the file; O_RDONLY for read only files, O_WRONLY for write only files, and O_RDWR for files that are created for reading and writing. Operation returns file descriptor of the opened file; SYSERR if the operation was not successful.

NAME:

Pool_User :: pool_fs_file_close

DECLARATION:

virtual SYSCALL pool_fs_file_close(int fd);

DESCRIPTION:

To close a file, users should use this function supplying the file descriptor *fd*. Function returns OK if operation was successful; SYSERR otherwise.

NAME:

Pool_User :: pool_fs_file_delete

DECLARATION:

virtual SYSCALL pool_fs_file_delete(char *path);

DESCRIPTION:

This method is used to delete a file, file name should be at the tail of the absolute *path*. Operation returns file descriptor if successful, SYSERR otherwise.

NAME:

Pool_User :: pool_fs_file_read

DECLARATION:

```
virtual SYSCALL    pool_fs_file_read(int fd, int len);
```

DESCRIPTION:

This function is used to read data starting at the current file position for "len" number of bytes. Returns the number of bytes read or SYSERR if there was an error.

NAME:

```
Pool_User :: pool_fs_file_write
```

DECLARATION:

```
virtual SYSCALL    pool_fs_file_write(int fd, int len);
```

DESCRIPTION:

This function is used to write data starting at the current file position for the length specified by "len" . Returns the number of bytes written or SYSERR if there was an error.

NAME:

```
Pool_User :: pool_fs_file_seek
```

DECLARATION:

```
virtual SYSCALL    pool_fs_file_seek(int fd, int offset, int rel_to);
```

DESCRIPTION:

This function moves the file *fd* read/write cursor to the position specified by the *offset* from the desired relative location. The relative location value *rel_to* can be; SEEK_CUR for seeking from file's current position, SEEK_END for seeking from the end of the file or SEEK_BEG for seeking from the beginning of the file. Function returns number of bytes seeked; SYSERR otherwise.

NAME:

```
Pool_User :: pool_fs_dir_create
```

DECLARATION:

```
virtual SYSCALL    pool_fs_file_create(char *path);
```

DESCRIPTION:

Users should use this function to create directories in their file system, the difference between this function and `pool_dir_create` is that the latter is to be used only if the `Dir_Server` class is used by itself as a single entity, otherwise the file server function should be invoked. The parameter *path* should be relative to the root and should be tailed

with the name of the directory to be created. Operation returns fd of the created directory; SYSERR otherwise.

NAME:

Pool_User :: pool_fs_dir_delete

DECLARATION:

virtual SYSCALL pool_fs_file_delete(char *path);

DESCRIPTION:

To delete a previously created directory. Name of directory must exist at the end of the absolute *path*. Returns fd if successful; SYSERR otherwise.

A.5 The NFS Class

NAME:

Pool_User :: pool_nfs_mount

DECLARATION:

virtual SYSCALL pool_nfs_mount(char *rdir, path *ldir);

DESCRIPTION:

This function is used to mount a remote directory on a local one. The two parameters; rdir and ldir should specify the remote directory absolute path and the local directory absolute path respectively. Operation returns OK if successful; SYSERR otherwise.

NAME:

Pool_User :: pool_nfs_file_open

DECLARATION:

virtual SYSCALL pool_nfs_file_open(char *path, int mode);

DESCRIPTION:

This NFS method is used to open an NFS file. The file name should be at the end of the absolute *path*, the parameter mode is used to specify the file opening mode, O_RDONLY for read only files, O_WRONLY for write only files, and O_RDWR for files that are created for reading and writing. Function returns file descriptor if successful; SYSERR otherwise.

NAME:

Pool_User :: pool_nfs_file_close

DECLARATION:

virtual SYSCALL pool_nfs_file_close(int fd);

DESCRIPTION:

Closes an opened file with the file descriptor *fd*. Returns OK if successful; SYSERR otherwise.

NAME:

Pool_User :: pool_nfs_file_read

DECLARATION:

virtual SYSCALL pool_nfs_file_read(int fd, int len);

DESCRIPTION:

Reads data from a file for the length *len* in bytes starting at the current file position of the file. Returns number of bytes read; SYSERR otherwise.

NAME:

Pool_User :: pool_nfs_file_write

DECLARATION:

virtual SYSCALL pool_nfs_file_write(int fd, int len);

DESCRIPTION:

Writes data to a file starting at the current file position for the length *len*. Returns number of bytes written; SYSERR otherwise.

NAME:

Pool_User :: pool_nfs_file_seek

DECLARATION:

virtual SYSCALL pool_nfs_file_seek(int fd, int offset, int rel_to);

DESCRIPTION:

Moves the read/write file cursor to the offset specified by offset from the desired relative location (see seek operation for file server). Returns number of bytes seeked; SYSERR otherwise.

Appendix B

Pool_User Example

This appendix contains a listing with description of an example that shows how one usage of the Pool_User class layer is performed.

In this example a simple class was constructed, methods in this class emulate simple i/o disk operation on POOL Disk abstraction. The Disk_Tester class is derived from POOL user interface class Pool_User (Figure B.1). Besides showing how a user can use the class Pool_User, this example also demonstrates the concept of wrappers previously introduced in chapter 4 (Implementation Issues).

Figure B.1 The Disk_Tester Class

The following is a listing of the Disk_Tester class declaration and related data

structures:

```

// -----
#include "pool.h"    // This header file contains header file
                    // of other POOL classes.

#define    MAX_DISK_TESTERS    10    // to determine the size
                                    // of the global array
                                    // disk_testers_list[]

int    disk_testers_used = 0;        // global variable
                                    // indicating the
                                    // incremental value of
                                    // indices in
                                    // disk_testers_list

int    nodes[3];                    // array of nodes ids
// -----

class Disk_Tester : public Pool_User {
private:    // Variables

char *my_name;    // class name, although not used in here,
                  // it can see usage in future usages, so
                  // it is convenient to obtain it in this
                  // early stage.

```

```

int  my_node_ind;    // node index of Disk_Tester in the
                    // global nodes array.
int  my_index;       // Disk_Tester class index in global
                    // array disk_testers_list
int  my_car_ind;     // Disk_Tester carrier index value in
                    // carriers_list array
int  my_disk_ind;    // Disk index in disks_list to be used by
                    // Disk_Tester
int  net_mess_size;  // Disk_Tester network message size
int  ds;             // Disk_Tester task id

public:

    // Disk_Tester constructor

    Disk_Tester(    char *name,
                    int  car_ind,
                    int  node_ind,
                    int  disk_ind,
                    int  ms,
                    int  *ind);

virtual ~Disk_Tester() {}
virtual void    dt_task(void);
static void    run_dt_task(void);
virtual SYSCALL my_disk_read(int disk,int block);
virtual SYSCALL my_disk_write(int disk,int block);

};

// -----

Disk_Tester*    disk_testers_list[MAX_DISK_TESTERS];

// -----

```

The above is the global array `disk_testers_list` used solely by the wrapper `run_dt_task`. This array demonstrates part of the wrapper mechanism first explained in chapter 4 under (POOL & PARASOL Integration Method). In summary, this global accessing mechanism is needed since POOL wrappers (`run_dt_task` in this example) are declared static the thing that prevents these wrappers from accessing other class members (data&methods).

The following is a code listing of the methods in the class Disk_Tester:

```
// -----

Disk_Tester :: Disk_Tester(char *name,int car_ind,
                           int node_ind,int disk_ind,int ms,
                           int *ind) : Pool_User("",car_ind,
                           node_ind,0,0,disk_ind,0,ms,ind)

    // The constructor of Disk_Tester calls the constructor
    // of the base class Pool_User and pass to it some
    // relevant data.
{

if ( disk_testers_used == MAX_DISK_TESTERS )
    pool_abort("Max disk testers reached!");

// recording of the given data
my_name = new char[strlen(name)+1];
if ( !my_name )
    pool_abort("Disk_Tester :: memory allocation (my_name)");
sprintf(my_name,"%s",name);
my_node_ind = node_ind;
net_mess_size = ms;
my_car_ind = car_ind;
my_disk_ind = disk_ind;
my_index = *ind = disk_testers_used++; // incrementing the
                                        // global indexing
                                        // value
disk_testers_list[my_index] = this;    // global array points
                                        // to this class
ds = ps_create("ds_task",
               nodes[my_node_ind],ANY_HOST,run_dt_task,0);

    // launching the wrapper task run_dt_task which in turn
    // launch the main class task dt_task, this is part of
    // the mechanism of integrating POOL and PARASOL
    // previously introduced in chapter 4.

ps_resume(ds);
ps_send(ps_std_port(ds),my_index,"",ps_my_std_port);
    // ps_send is used to send the global index value to the
    // wrapper run_ds_task.
}
// -----
```

```
SYSCALL Disk_Tester :: my_disk_read(int disk, int block)
{
```

/* This very basic method is used to read from the Disk class of POOL, it is basically calling the method pool_disk_read in the base class Pool_User, however, this is only intended to be an example showing the re-use of the previously defined methods in the Pool_User class. More functionality can be added here to perform other needed operations besides the basic disk read operation. */

```
    if ( pool_disk_read(disk,block) == SYSERR )
        return(SYSERR);
    return OK;
}
```

```
// -----
```

```
SYSCALL Disk_Tester :: my_disk_write(int disk, int block)
{
```

```
    // my_disk_write is the same as my_disk_read but it is
    // used for writing blocks of data to the disk

    if ( pool_disk_write(disk,block) == SYSERR )
        return(SYSERR);
    return OK;
}
```

```
// -----
```

```
void Disk_Tester :: dt_task(void)
```

```
    // this code is for the dt_task which randomly performs
    // i/o disk operations.
{
```

```
    int      ioop;      // operation value READ or WRITE
    int      block_id;  // accessed block id
```

```
    while(TRUE) {
        ioop = ps_choice(2); // returns values either 0 or 1
```

```

        block_id = ps_choice(100); // choosing a block id (0-99)
        if (iloop == READ) {
            if ( my_disk_read(my_disk_ind,block_id) == SYSERR )
                pool_abort("ds_task : reading from a disk");
        }
        else {
            if ( my_disk_write(my_disk_ind,block_id) == SYSERR )
                pool_abort("ds_task : reading from a disk");
        }
    }
}

// -----

void Disk_Tester :: run_dt_task(void)

    /* This the wrapper task which basically receives the
    class Disk_Tester index dt_index and calls the method
    dt_task by accessing the global array disk_testers_list */
{
    int         ap;
    int         dt_index;
    double      ts;
    char        *tp;

    ps_receive(ps_my_std_port,NEVER,&dt_index,&ts,&tp,&ap);
    disk_testers_list[dt_index]->dt_task();
}

// -----

void ps_genesis(void)
{
    /*
        Ps_genesis task (previously described in chapter 2) is
        used to construct the hardware needed for this example where
        a simple network is constructed. This network consists of
        three nodes connected by POOL Carrier.    */

    int         i;           // used in for loop
    int         disk_ind;    // global indexing for Disk class.
    int         car_ind;     // global indexing for Carrier class.
    int         ds_ind;      // Disk_tester class global index.

    for ( i=0; i<3; i++ )    // building nodes using PARASOL call
        nodes[i] = ps_build_node("",10,1.0,0.0,FCFS,FALSE);

```

```
Carrier *cp = new Carrier("",
                        nodes[0], ANY_HOST, 3,
                        nodes, 512, 0.0000012, &car_ind);
    // Connecting nodes by building POOL Carrier

Disk *dp = new Disk("", nodes[1], ANY_HOST, 10, 20, 30, 0.000012,
                    0.000012, 0.012, 0.021, car_ind, 1, 50, &disk_ind);
    // Creating Disk class

Disk_Tester *dt = new Disk_Tester("disk_tester",
                                   car_ind, 2, disk_ind, 50, &ds_ind);
    // Creating Disk_Tester class object

ps_suspend(ps_myself);
}
```

References

- [Alexander 90] Alexander, H., Jones, V., Software Design and Prototyping using me too, Prentice-Hall, Hertfordshire, UK, 1990.
- [Atkinson 91] Colin Atkinson, Object-Oriented Reuse, Concurrency and Distribution, An Ada-Based Approach, (ACM Press Books) Addison-Wesley Publishing Company, 1991.
- [Bar-David 93] Tsvi Bar-David, Object-Oriented Design for C++, Prentice-Hall, Englewood Cliffs. 1993.
- [Black 86] A. Black, N. Hutchinson, E. Jul, and Levy. Object structure in the Emerald system. In ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, Oct. 1986.
- [Black 87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter., "Distribution and abstract types in Emerald", IEEE Transactions on Software Engineering, SE-13(1): 65-77, Jan. 1987.

-
- [Black 88] A. Black, N. Hutchinson, E. Jul, and Levy., "Fine-grained mobility in the Emerald system",. ACM Transactions on Computer Systems, 6(1):109-133, Feb. 1988.
- [Blum 82] Blum, B. I, "The Life Cycle : A Debate over alternative models," ACM SIGSOFT Software Engineering notes 12(9), 1982. pp. 988-993.
- [Booch 91] Grady Booch, Object-Oriented Design with Applications, The Benjamin/Cummings Publishing Company, Inc. 1991.
- [Brewer 91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, William E. Weihl, "PROTEUS: A High-Performance Parallel-Architecture Simulator", MIT/LCS/TR-516, Laboratory of Computer Science, MIT, Sept. 1991.
- [Bruy 84] Bruyuooghe, M. and Venken, R., "Prolog as a Language for Prototyping of Information Systems", In Approaches to Prototyping, R. Budde, K Kuhlenkamp, L. Mathiassen and H. Zullighoven, Eds., Springer-Verlag, 1984, pp. 447-458, (Proceedings of the Working Conference on Prototyping, Namur, Oct. 1983).
- [Budde 92] Budde, R., Kuhlenkamp, K., Zullighoven, H., Prototyping: An Approach to Evolutionary System Development, Springer-Verlag, New York, 1992.
- [Buhr 90a] Buhr, R.J.A., Practical Visual Technical in System Design with Application to Ada, Prentice Hall Software Series, Prentice Hall, 1990.

-
- [Buhr 90b] Buhr, R.J.A., Karam, G.M., Woodside, C.M., Casselman, R., Franks, G., Scott, H., Bailey, D., "TimeBench: a CAD Tool for Real-Time System Design", SCE-90-17, Real-Time and Distributed Systems Group, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, Sept. 1990.
- [Chang 87] Chang, C.K., et al., 1987. "A New Design Approach of Real-Time Distributed Software Systems." In Proceedings of COMPSAC87 (Tokyo, JAPAN, Oct 7-9). IEEE, Piscataway, N.J., 474-479.
- [Chase 89] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield., "The Amber system: Parallel programming on a network of multiprocessors", In Proceedings of the 12th ACM Symposium on Operating Systems Principles, pp. 147-158, Litchfield Park, Arizona USA, Dec. 1989. ACM.
- [Cieslak 88] Cieslak R., A. Fawaz, S. Sachs, P. Varaiya, J. Walrand, and A. Li, 1988. "The Programming Network Prototyping System," IEEE Software, pp. 67-76.
- [Coad 90] Peter Coad, Edward Yourdon, Object-Oriented Analysis, Yourdon Press, Prentice Hall, 1990.
- [Condry 92] Michael W Condry, Swee Boon Lim and Lup yuen Lee, "The Object-Oriented Advantage in Prototyping a Remote File System", Proceedings Second International Workshop on Object-Orientation in Operating Systems, France, Sept. 1992.

-
- [Coulouris 88]** Coulouris , George F., Dollimore Jean. ,Distributed Systems, Concepts and Design, Addison-Wesley Publishers Ltd. 1998
- [Dasgupta 86]** Dasgupta, P., 1986, "A Probe Based Monitoring Scheme for an Object-Oriented Distributed Operating Systems", OOPSLA '86 Conference Proceedings, SIGPLAN notices (special issue) Vol. 21 no. 11, pp. 57-66.
- [Dasgupta 88]** P. Dasgupta, R. Leblanc, Jr., and W. Appelbe."The Clouds distributed Operating Systems: Functional description, implementation details and related work", In Proceedings of 8th Int. Conf. on Distributed Computing Systems, pp.2-9, S.Jose CA, USA, June 1988. (IEEE).
- [Dwig 83]** Dwiggins, D., "Prolog as a System Design Tool", In Proceedings of the 16th Annual Hawaii International Conference on System Sciences, 1983, pp. 14-23.
- [Frankowski 86]** Frankowski, E., Advantages of the Object Paradigm for Prototyping, Golden valley, MN:Honeywll, March, 1986.
- [Gold 84]** Goldberg, A., Smalltalk-80: The Interactive Programming Environment, Addison-Wesely, 1984.
- [Habert 90]** Sabine Habert, Laurence Mosseri, and Vadim Abrossimov, "COOL: Kernel support for object-oriented environments", In ECOOP/OOPSLA '90 Conference vol. 25 of SIGPLAN Notices, pp. 269-277, Ottawa, Canada, Oct. 1990. ACM

-
- [Hekmatpour 88] Hekmatpour, S., and Ince, D., Software Prototyping, Formal Methods and VDM, International Computer Science Series, Addison-Wesely, 1988.
- [Henderson 86] Henderson, p. "Functional Programming, Formal Specification and Rapid Prototyping", IEEE trans. on Software Engineering, Dec. 1986. pp. 241-250.
- [Jacobson 92] Jacobson Ivar ,Object-Oriented Software Engineering, A Use case Driven Approach, Addison-Wesley Publishers Ltd. 1992.
- [Karam 89] Karam, G. M., MLog: A Language for Prototyping Concurrent Systems, Tech. Rep. CU-SCE-88-3 (Revised), Department of Systems and Computer engineering, Carleton University, Ottawa, Ontario, Canada, September 1989.
- [Kern 84] Kernighan, B. W. and Pike, R., The UNIX Programming Environment, Prentice-Hall Software Series, Prentice-Hall, 1984.
- [Lalonde 94] Wilf Lalonde,Discovering Smalltalk, The Benjamin/Cummings Publishing Company, Inc. 1994.
- [Lieb 84] Liebrandt, U. and Schnupp, P., "An Evaluation of Prolog as a Prototyping System", In Approaches to Prototyping, R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Zullighoven, Eds., Springer-Verlag, 1984 pp. 424-433, (Proceedings of the Working Conference on Prototyping, Namur, Oct. 1983).

-
- [Liskov 84] B. Liskov. Overview of the Argus language and system. Technical report, MIT, Feb. 1984. Programming Methodology Group Memo 40.
- [Liskov 87] B. Liskov, D. Curtis, P. Jhonson and R. Scheifler. "Implementation of Argus", In Proceedings of the 11th ACM Symposium on Operating Systems Principles, pp. 111-122, Austin TX, USA, Nov. 1987, ACM.
- [Liskov 88] B. Liskov and R. Scheifler. "Guardians and actions: Linguistic support for robust, distributed programs", ACM Transactions on Programming Languages and Systems, 5(3):381-404, July, 1988.
- [Luqi 88] Luqi, Ketabchi, M., "A Computer-Aided Prototyping System", IEEE Software, March, 1988, pp. 66-72.
- [Luqi 89] Luqi, Royce, W., "Software Evolution Through Rapid prototyping", IEEE Computer, May. 1989, pp. 13-25.
- [Luqi 92] Luqi, Royce, W., "Status Report: Computer-Aided Prototyping", IEEE Software, Nov., 1992, pp. 77-81.
- [Madany 88] Madany, P., Leyens, D., Russo, V. and Campbell, R. "A C++ Class Hierarchy for Building UNIX-like File System.", Proceedings of USENIX C++ Conference, Berkeley CA (USENIX Association) 1988.
- [Masini 91] Masini, Napoli, Colnet, Leonard, Tomber, Object-Oriented Languages, Academic Press 1991.

-
- [McCrackon 82] McCrackon, D., Jackson, M. "Life Cycle Concepts Considered Harmful", ACM SIGSOFT Software Engineering notes 7(2) 1982, pp. 29-32.
- [Mullender 89] Mullender, Sape, Distributed Systems, ACM Press, 1989.
- [Mullender 90] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse and H. van Stavern, "Amoeba: A Distributed Operating System for the 1990s", IEEE Computer, May 1990. pp. 44-53.
- [Neilson 91a] John E. Neilson, "PARASOL: A Simulator for Distributed And/Or Parallel Systems" Users' Manual", SCS-TR-192, School of Computer Science, Carleton University, Ottawa, Canada, May 1991.
- [Neilson 91b] John E. Neilson, "PARASOL Users' Manual", School of Computer Science, Carleton University, Ottawa, Canada, 1991.
- [Neilson 91c] John E. Neilson, "A PARASOL Tutorial ", School of Computer Science, Carleton University, Ottawa, Canada, 1991.
- [Nichols 90] K. M. Nichols, "Performance Tools", IEEE Software, May, 1990, pp. 21-30.
- [Rumbaugh 91] Rumbaugh, James., Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Object-Oriented Modeling and Design, Hall, 1991.
- [Rebecca 90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, Designing Object-Oriented Software, Prentice-Hall, Inc. 1990.

- [Sessions 92]** Roger Sessions, **Class Construction in C and C++**, **Object-Oriented Programming Fundamentals**, PTR Prentice Hall, 1992.
- [Shaprio 86]** Marc Shaprio, "Structure and encapsulation in distributed systems: the Proxy Principle", **Proc. 6th Int. Conf. on Distributed Computing Systems**, pp.198-204, Cambridge, Mass, USA, May, 1986, IEEE.
- [Shaprio 89a]** Marc Shaprio, "Prototyping a distributed object-oriented OS on Unix", In Eugene Spafford, editor, **Workshop on Experience with Building Distributed and Multiprocessor Systems**, Ft. Lauderdale FL, USA, Oct. 1989. USENIX.
- [Shaprio 89b]** M. Shaprio, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. "SOS: An object-oriented operating system - assessment and perspectives", **Computing Systems**, 2(7), Dec. 1989.
- [Shaw 85]** Shaw, M., "What Can We Specify? Issues in the domain of Software specification", **Proc. 3rd Int. Workshop Software Specification & Design**, London, 1985, pp. 214-215.
- [Smith 91]** Smith, M.F., **Software Prototyping: Adoption, Practice and Management**, McGraw-Hill Book Company, London, 1990.
- [Snodgrass 83]** Snodgrass, R., 1983, " An Object_Oriented Command Languages", **IEEE Transaction on Software Engineering**, Vol. SE-9 No. 1, Jan. pp. 1-8.

-
- [Stroustrup 90] M. Ellis and B Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley Publishing Company, Don Mills, ON, Canada, 1990.
- [Swartout 82] Swartout, W., Balzer, R., "On the inevitable intertwining of Specification and Implementation", Comm. of the ACM, 1982, 25(7), pp. 438-440.
- [Tanenbaum 85] A. S. Tanenbaum and R. Renesse, "Distributed Operating Systems", ACM Computing Surveys, Vol. 17, No. 4 Dec. 1985.
- [Tanenbaum 88] A. S. Tanenbaum, R. van Renesse and H. van Staveren, "Performance of the World's Fastest Distributed Operating System", Operating System Reviews, 22(4): 22-34, Oct. 1988.
- [Tanenbaum 92] A. S. Tanenbaum, Modern Operating Systems, Prentice-Hall Inc. 1992.
- [Tavo 84] Tavolato, P. and Vincena, K., "A Prototyping Methodology and its Tool", In Approaches to Prototyping, R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Zullighoven, Eds., Springer-Verlag, 1984, pp. 434-446, (Proceedings of the Working Conference on Prototyping, Namur, Oct. 1983).
- [Taylor 82] Taylor, T. and T.A. Standish. 1982. "Initial Thought on Rapid Prototyping Techniques." ACM SIGSOFT Software Engineering Notes 7, no.5 (Dec.): 160-166.

- [Zave 86]** **Zave, P., W. Schell Salient., "Features of an Executable Specification Language and its Environment", IEEE Trans. on Software Engineering, Feb. 1986, pp. 312-325.**
- [Zedan 90]** **H. S. M. Zedan, Distributed Computer Systems, Butterworth & CO(Publisher) Ltd., 1990.**

END

3 1 - 0 5 - 9 5

FIN