The Design & Implementation of a Simple Persistent Object Server

by

Simon Ma

32

A thesis

submitted to the Faculty of Graduate Studies

in partial fulfillment of the requirements

for the degree of

Master's of Science

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

Canada

April 1996

ISBN 0-612-13326-5

Name _____

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

Computer Science
SUBJECT TERM

0984
SUBJECT CODE

UMI

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
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

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

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- 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
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology ... 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

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
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology .. 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984

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

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

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

Nom _____

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

_____ ☐☐☐ U·M·I

SUJET — CODE DE SUJET

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS
Architecture 0729
Beaux-arts 0357
Bibliothéconomie 0399
Cinéma 0900
Communication verbale 0459
Communications 0708
Danse 0378
Histoire de l'art 0377
Journalisme 0391
Musique 0413
Sciences de l'information 0723
Théâtre 0465

ÉDUCATION
Généralités 515
Administration 0514
Art 0273
Collèges communautaires 0275
Commerce 0688
Économie domestique 0278
Éducation permanente 0516
Éducation préscolaire 0518
Éducation sanitaire 0680
Enseignement agricole 0517
Enseignement bilingue et multiculturel 0282
Enseignement industriel 0521
Enseignement primaire. 0524
Enseignement professionnel 0747
Enseignement religieux 0527
Enseignement secondaire 0533
Enseignement spécial 0529
Enseignement supérieur 0745
Évaluation 0288
Finances 0277
Formation des enseignants 0530
Histoire de l'éducation 0520
Langues et littérature 0279
Lecture 0535
Mathématiques 0280
Musique 0522
Orientation et consultation 0519
Philosophie de l'éducation 0998
Physique 0523
Programmes d'études et enseignement 0727
Psychologie 0525
Sciences 0714
Sciences sociales 0534
Sociologie de l'éducation 0340
Technologie 0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE
Langues
- Généralités 0679
- Anciennes 0289
- Linguistique 0290
- Modernes 0291

Littérature
- Généralités 0401
- Anciennes 0294
- Comparée 0295
- Médiévale 0297
- Moderne 0298
- Africaine 0316
- Américaine 0591
- Anglaise 0593
- Asiatique 0305
- Canadienne (Anglaise) 0352
- Canadienne (Française) 0355
- Germanique 0311
- Latino-américaine 0312
- Moyen-orientale 0315
- Romane 0313
- Slave et est-européenne 0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE
Philosophie 0422
Religion
- Généralités 0318
- Clergé 0319
- Études bibliques 0321
- Histoire des religions 0320
- Philosophie de la religion 0322

Théologie 0469

SCIENCES SOCIALES
Anthropologie
- Archéologie 0324
- Culturelle 0326
- Physique 0327

Droit 0398
Économie
- Généralités 0501
- Commerce-Affaires 0505
- Économie agricole 0503
- Économie du travail 0510
- Finances 0508
- Histoire 0509
- Théorie 0511

Études américaines 0323
Études canadiennes 0385
Études féministes 0453
Folklore 0358
Géographie 0366
Gérontologie 0351
Gestion des affaires
- Généralités 0310
- Administration 0454
- Banques 0770
- Comptabilité 0272
- Marketing 0338

Histoire
- Histoire générale 0578
- Ancienne 0579
- Médiévale 0581
- Moderne 0582
- Histoire des noirs 0328
- Africaine 0331
- Canadienne 0334
- États-Unis 0337
- Européenne 0335
- Moyen-orientale 0333
- Latino-américaine 0336
- Asie, Australie et Océanie 0332

Histoire des sciences 0585
Loisirs 0814
Planification urbaine et régionale 0999
Science politique
- Généralités 0615
- Administration publique 0617
- Droit et relations internationales 0616

Sociologie
- Généralités 0626
- Aide et bien-être social 0630
- Criminologie et établissements pénitentiaires 0627
- Démographie 0938
- Études de l' individu et de la famille 0628
- Études des relations interethniques et des relations raciales 0631
- Structure et développement social 0700
- Théorie et méthodes. 0344
- Travail et relations industrielles 0629

Transports 0709
Travail social 0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES
Agriculture
- Généralités 0473
- Agronomie. 0285
- Alimentation et technologie alimentaire 0359
- Culture 0479
- Élevage et alimentation 0475
- Exploitation des pâturages 0777
- Pathologie animale 0476
- Pathologie végétale 0480
- Physiologie végétale 0817
- Sylviculture et faune 0478
- Technologie du bois 0746

Biologie
- Généralités 0306
- Anatomie 0287
- Biologie (Statistiques) 0308
- Biologie moléculaire 0307
- Botanique 0309
- Cellule 0379
- Écologie 0329
- Entomologie 0353
- Génétique 0369
- Limnologie 0793
- Microbiologie 0410
- Neurologie 0317
- Océanographie 0416
- Physiologie 0433
- Radiation 0821
- Science vétérinaire 0778
- Zoologie 0472

Biophysique
- Généralités 0786
- Medicale 0760

SCIENCES DE LA TERRE
Biogéochimie 0425
Géochimie 0996
Géodésie 0370
Géographie physique 0368
Géologie 0372
Géophysique 0373
Hydrologie 0388
Minéralogie 0411
Océanographie physique 0415
Paléobotanique 0345
Paléoécologie 0426
Paléontologie 0418
Paléozoologie 0985
Palynologie 0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT
Économie domestique 0386
Sciences de l'environnement 0768
Sciences de la santé
- Généralités 0566
- Administration des hôpitaux 0769
- Alimentation et nutrition 0570
- Audiologie 0300
- Chimiothérapie 0992
- Dentisterie 0567
- Développement humain 0758
- Enseignement 0350
- Immunologie 0982
- Loisirs 0575
- Médecine du travail et thérapie 0354
- Médecine et chirurgie 0564
- Obstétrique et gynécologie 0380
- Ophtalmologie 0381
- Orthophonie 0460
- Pathologie 0571
- Pharmacie 0572
- Pharmacologie 0419
- Physiothérapie 0382
- Radiologie 0574
- Santé mentale 0347
- Santé publique 0573
- Soins infirmiers 0569
- Toxicologie 0383

SCIENCES PHYSIQUES
Sciences Pures
Chimie
- Généralités 0485
- Biochimie 487
- Chimie agricole 0749
- Chimie analytique 0486
- Chimie minérale 0488
- Chimie nucléaire 0738
- Chimie organique 0490
- Chimie pharmaceutique 0491
- Physique 0494
- PolymÈres 0495
- Radiation 0754

Mathématiques 0405
Physique
- Généralités 0605
- Acoustique 0986
- Astronomie et astrophysique 0606
- Électronique et électricité 0607
- Fluides et plasma 0759
- Météorologie 0608
- Optique 0752
- Particules (Physique nucléaire) 0798
- Physique atomique 0748
- Physique de l'état solide 0611
- Physique moléculaire 0609
- Physique nucléaire 0610
- Radiation 0756

Statistiques 0463

Sciences Appliqués Et Technologie
Informatique 0984
Ingénierie
- Généralités 0537
- Agricole 0539
- Automobile 0540
- Biomédicale 0541
- Chaleur et thermodynamique 0348
- Conditionnement (Emballage) 0549
- Génie aérospatial 0538
- Génie chimique 0542
- Génie civil 0543
- Génie électronique et électrique 0544
- Génie industriel 0546
- Génie mécanique 0548
- Génie nucléaire 0552
- Ingénierie des systèmes 0790
- Mécanique navale 0547
- Métallurgie 0743
- Science des matériaux 0794
- Technique du pétrole 0765
- Technique minière 0551
- Techniques sanitaires et municipales 0554
- Technologie hydraulique 0545

Mécanique appliquée 0346
Géotechnologie 0428
Matières plastiques (Technologie) 0795
Recherche opérationnelle 0796
Textiles et tissus (Technologie) 0794

PSYCHOLOGIE
Généralités 0621
Personnalité 0625
Psychobiologie 0349
Psychologie clinique 0622
Psychologie du comportement 0384
Psychologie du développement 0620
Psychologie expérimentale 0623
Psychologie industrielle 0624
Psychologie physiologique 0989
Psychologie sociale 0451
Psychométrie 0632

THE UNIVERSITY OF MANITOBA

FACUTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

THE DESIGN & IMPLEMENTATION OF A SIMPLE PERSISTENT OBJECT SERVER

BY

SIMON MA

A Thesis/Practicum submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Table of Contents

Abstract

Object-based systems are quite heavily investigated in recent years because many database researchers now agree that the relational database technology of the 1980s falls far short of providing the necessary data abstraction and modelling capability. This thesis presents the implementation design of a simple object-based system called the ***Persistent Object Server*** (POServer). The primary function of POServer is to provide object access and persistence. A system architecture based entirely on Mach's RPC mechanism is presented and the Application Programming Interface (API) is illustrated. A simple object model that consists of the core object-oriented features is defined. The implementation design of the Object Manager and the Storage Manager are discussed in detail. Also, different implementation techniques and design alternatives are examined. Finally, the algorithms for the implementation of inheritance and behavior are introduced. The thesis contributes towards a foundation for object-based systems. The system architecture, the object model and the algorithms form the framework of the POServer system and enable the system designer to take one step closer to building a full-blown object-based system.

Acknowledgements

I would like to take this opportunity to express my appreciation and thank a few people for their support and encouragement. First of all, I would like to thank my supervisor, Dr. Ken Barker, for giving me the opportunity to pursue this practical topic. He has provided me with a lot of valuable insights and clear directions during the course of writing my thesis. It was a pleasant experience to have worked with him. The thesis has been proven to be very challenging. I have learned a lot about the implementation techniques for object design and development. Most importantly, I am able to make use of what I have learned from this thesis and apply the concepts and the knowledge to my work. I am currently involved in the system design of a *three tier* application architecture at Great-West Life. *Three tier* application architecture is a refinement of the Client-Server architecture which separates each application into data, application, and presentation layers. Secondly, I would also like to thank my peers in the Advanced Database Systems Laboratory, especially Dr. Peter Graham and Mr. Gilbert Detillie, for their technical expertise in setting up Mach that allowed me to test out the RPC mechanism. Last, and definitely not the least, I wish to thank my lovely wife Alice. Your encouragement over the last two years has been invaluable. Without the love and support from you, I would have never made it this far.

List of Figures

Chapter 1

Introduction

Database management systems (DBMS) were introduced in the late 1960s to overcome the inherent limitations of file management systems such as data redundancy, inflexibility, and lack of data independence. A DBMS is an implementation of a data model that consists of a set of services such as backup and recovery, transaction management, concurrency control, and security protection to help maintain the integrity of the database. Since their introduction in the late 1960s, DBMSs have undergone several generations of evolution. The principle data models implemented include network model, hierarchical model and relational model. The relational model as defined by Codd [14] represents the current generation of DBMSs, in which data are represented in the form of tables and relations, has met the needs of simple business data processing applications such as payroll, accounting, inventory control, and so on. However, the relational model offers the simple notion of tables and character-based data as the only data structures for user interaction. The set of operations in relational systems is limited to set theory and first-order logic. The relational model cannot capture the semantics of complex objects. To model a complex object the information often has to be split into several tables. This makes each access to such objects slow since the DBMS must join a lot of tables to gather the object's information. In summary, the flat nature of the relational data model is not able to adequately support applications that require large

complex data structures. These include images, voice, graphics and documents. For example, most insurance companies have a substantial on-line database system that records information for policyholders and their benefit payments. Ideally, these databases should be enhanced with multimedia data such as digitized images of hand-written applications, audio transcripts of underwriters' evaluations, and photographs of specially insured objects, as discussed in Koehler [32].

Many database researchers now agree that the relational database technology of the 1980s falls far short of providing the necessary data abstraction or modelling capability to act as the repository of hypermedia objects and applications. There has been a movement within the database field to incorporate more and more semantics into the data model so that the database can more closely reflect an application. Relational databases cannot provide direct, natural representation of graph-structured object spaces. Relational systems are subject to the limitations of a finite set of data types and the need to normalize data. In contrast, object-based systems offer flexible abstract data-typing facilities and the ability to encapsulate data and operations via the message protocol. The intuitive appeal of an object model is that it provides better concepts and tools to represent the real world as closely as possible. As a result, the object model has attracted much interest in the database research community. The need for more flexible type systems has been one of the major driving forces in the development of object-based systems. As pointed out in Kim [29, 31], object-based systems are replacing relational

systems and becoming the model of choice for advanced applications that require a rich type system, highly flexible data structures, and an enriched set of data modelling constructs. These new applications include geographical information systems (GIS), software engineering, computer-aided design (CAD) and computer-aided manufacturing (CAM). The object model represents the latest step in the evolution of database technology.

Object-oriented (OO) concepts have evolved in three different areas: first in programming languages, then in artificial intelligence, and then in databases. Simula is generally regarded as the first OO programming language. Simula [17] introduced as its basic building block the *object*, a package containing both the data and procedures which acted on that data. This was also the beginning of the use of *data encapsulation* within programming languages. Since Simula, researchers in programming languages have taken two different paths to promote OO programming. One was the development of new OO languages, most notably Goldberg's Smalltalk [23] and such languages as Eiffel [39]. It is important to note that object models can contain tuples, for example, the dictionary class in Smalltalk is a form of table. Another approach was the extension of conventional languages, for example, Objective C and C++ [21] as extensions of C. OO programming has shifted the focus considerably closer to that of databases, by emphasizing the organization of software around the data objects, rather than around flow of control. In the OO approach, objects have a prescribed behavior that enables them to

respond to messages from users and from other objects. Objects are grouped into classes and the prescribed behavior of a class applies to all objects that are instances of the class. The rich modelling semantic of an object model provide intrinsic elements that facilitate *classification, data abstraction, encapsulation* and *inheritance*. These new features lead to a greater degree of program modularization, software reusability, modifiability and maintainability.

Object DBMS (ODBMS) research has been under way in universities and research laboratories for many years. To fully understand the possibilities of the ODBMS technology, two of the most common approaches taken by database researchers are considered. The early object database systems were developed as extensions of OO programming languages such as C++ and Smalltalk. This approach, sometimes called the persistent language approach uses existing programming language type system as the object model. Permanent storage is then added to objects defined in the object programming language. Data definition and type checking are borrowed from the programming language. The systems from this category can be viewed as persistent versions of C++ and Smalltalk. For example, GemStone [10] illustrates this approach. The second approach taken by the researchers is to add object extensions to a relational DBMS. These systems are often called extended relational DBMSs. This is typically accomplished by relaxing the data type constraints imposed by conventional systems. The extended features include binary language objects (BLOBs), set and tuple-valued

fields, user-defined and abstract data types, path expressions, triggers/rules, and so on. Two of the most prominent research prototypes that use this approach are POSTGRES from University of California, Berkeley [55] and Starburst from IBM's Almaden Research Centre [25, 36]. Relational DBMS commercial products (e.g., Ingres, Oracle, Sybase) are gradually incorporating some of the above object-oriented extensions. For example, Sybase has added "stored procedures" in their SQL Server product.

The persistent language approach provides an easier, more flexible way for users to store language objects. However, they are often restricted to the capabilities of the specific object-oriented language. That is, it provides no simple means for accessing the objects from any other language. Meanwhile, an explicit goal of the relational extension approach is to minimize changes to the relational model, thus the basic model in this approach is still based on the notion of records and tables. As a result, some problems remain unresolved such as *impedance mismatch* [15] and inefficient single-object access. The extent to which the relational model and the object-oriented model can be narrowed while remaining within the bounds of relational theory is thus uncertain.

1.1 Motivation And Objectives

The ground up Object DBMS approach is quite heavily investigated in recent years. The purpose of this thesis is to present and discuss the implementation design of a simple

object-based system called the ***Persistent Object Server*** (POServer). The approach taken by POServer is broadly similar to the research prototypes Iris [22], TIGUKAT [26] and O_2 [18, 19]. The strategy adopted in these OO systems is to build an object model on a database foundation. The object model foundation allows the user to benefit from the OO features of encapsulation, information hiding, inheritance, polymorphism, and dynamic binding. The DBMS foundation provides the traditional DBMS features such as recovery, concurrency, security, and so on. The result is a true ODBMS. These systems also allows access to the database from multiple languages and a query language to access and manipulate objects. The separation of the object model implementation from programming languages and storage management is considered a step in the right direction because this approach can obtain most benefits and offer the highest level of object-orientation. Essentially, POServer can support applications in any programming language for which a language binding exists since it is based on a language independent object model.

Object-oriented systems such as EXODUS [11] and Mneme [40] are being classified as *Persistent Object Stores* because these systems have been built without providing all the functionality of a DBMS, for example, recovery and transaction management. POServer can be considered to be in the same group. The primary functions of POServer are to provide object persistence and object access. Persistence refers to the capability of storing objects in non-volatile storage and allowing the programmer to read the objects

back later in core memory for further processing. A persistent object is one which continues to exist after the application that created it has terminated. Computer programs use volatile memory for their runtime data storage. This applies equally to C++ and COBOL programs. Both objects created by a C++ program and records created by a COBOL program must be transferred to secondary storage if they are to persist. POServer is based on a client/server architecture and consists of two major components: 1) Object Manager (OM) and 2) Storage Manager (SM). The OM is responsible for the implementation of the conceptual object model and object management. The SM provides persistent capability and buffer management. The implementation issues which are of direct concern to system designers are highlighted.

This thesis is also motivated by the advanced capabilities of Mach. The server market has moved rapidly toward UNIX and it is widely accepted that UNIX now dominates the client/server environments. Mach is a Unix-compatible microkernel-based operating system developed by Carnegie Mellon University (CMU). Mach's original development was based on BSD 4.3 Unix as described in [1]. It provides users with extensive interprocess communications, improved memory management, multiprocessors support, and multithreading facilities. Mach is also a distributed operating system that is extremely suitable for developing distributed object-based systems [27]. The benefits of using Mach for building a client/server system will be fully discussed. The focus of this

thesis is not to present a complete object model, but rather to demonstrate how to design and implement a simple persistent object-based server.

There is currently a research project in the Advanced Database Systems Laboratory (ADSL) to develop a Distributed Object-Based System. During the early stage of the research project, arrangements were made with CMU's software distribution manager to enable one of our workstations in the DB lab to obtain source code and binary code for Mach over the Internet. The netcrypt.c file was installed which allows us to do periodic software updates. The microkernel Mach system was ported to two DecStation 5000's. It was set up to run on Carbon and Boron under the directory /usr/mach. A few client-server modules were developed to validate Mach's IPC facilities. These programs can be found under /usr/mach/machpgm. The concepts for the POSerer RPC was then formed.

This thesis contributes to the broad objective of building an object-based system in the following ways:

1. Provides a working environment running Mach in the DB lab for system testing and prototype development.
2. Validates the facilities provided in Mach and determines how to make the best use of it to build an object-based system.

3. Defines a framework for a simple object-based server and it is served as a prototype for the Distributed Object-Based System to be developed in the lab.
4. Provides algorithms for the design and implementation of inheritance and method resoultion.

1.2 Related Work

In this section, two object-based systems reported in the literature that have direct influence on this thesis are reviewed. The first one is Iris [22] (Hewlett-Packard's Open ODB is based on the Iris prototype). Iris implements an object model which is managed by an object manager on top of a relational storage and transaction manager. In Iris, the object-oriented model is based upon three components: *objects, types* and *functions*. Objects are a combination of data and stored functions. Types allow you to classify similar objects. Functions operate on data in the database and also define the behavior of that data in the database. Iris uses a client/server architecture and allow users to interactively enter Object SQL (OSQL). The object manager executes OSQL calls made by the Iris clients. The OSQL interface layer is constructed to transparently connect client's programs to a relation database manager, HP-SQL. The interface layer defines a general algorithm to map from objects to database representations and back again.

Another research prototype that demonstrates similar approach is the TIGUKAT (tee-goo-kat) system [26]. TIGUKAT is a term in the language of the Canadian Inuit people meaning "object". This system is currently under development at University of Alberta. The object manager encloses the core object model and is responsible for the interaction between TIGUKAT Query Language (TQL) and persistent storage manager (ESM). ESM supports a client/server topology where the client module is linked with the host application program and interacts with the ESM server. The TIGUKAT object model is very well defined. It includes *T_object, T_type, T_behavior, T_semantics, T_function, T_collection* and *T_atomic.* T_atomic contains primitive objects such as reals, integers, characters, sets, bags, lists, etc. T_type is used for defining and structuring objects; T_behavior and T_semantics together provides support for specifying the semantics of the operations which may be performed on the objects; T_function is used for specifying the implementations of behaviors over various types and T_collection supports the grouping of objects in the system.

1.3 Organization Of Thesis

This thesis is divided into six chapters. This chapter has discussed the evolution of database management systems and the shortcomings of relational systems. The rapid change in application environments and the applicability of object-based systems were also presented. This was followed by a brief history of object orientation and high-level

descriptions of OO concepts. Two most common approaches taken by the researchers to building object-based systems, 1) adding persistence to a language environment and 2) extending a relational DBMS, were examined and compared. Object-based systems represent the latest generation of database technology. The trend is to use object-based systems for the design and development of complex applications. Section 1.1 addressed the purpose of this thesis, an approach for implementing the persistent object server, and the motivation of using the micro-kernel operating system Mach. POServer is based on a client/server architecture because client/server computing has received widespread recognition as a leading foundation technology. A couple of related research prototypes, Iris and TIGUKAT, were briefly reviewed in section 1.2.

The remainder of the thesis is organized as follows. Chapter 2 gives an overview of the system architecture. The application programming interface is presented in Section 2.1. Section 2.2 and 2.3 briefly discuss the roles and responsibilities of the two key components, the Object Manager and the Storage Manager, in the POServer system and how they fit into the overall picture. Section 2.4 and 2.5 discuss the benefits of Mach and how to use its RPC facility to implement a client/server system. Chapter 3 presents the conceptual object model. Sections 3.1-3.4 define and justify what an object-based system should be, on the basis of a small set of central object-oriented concepts such as the notion of abstract data typing, encapsulation, inheritance and object identity. Section 3.5 defines the primitive class hierarchy for the POServer object model. Section 3.6 briefly

discusses referential integrity in the object model. In the main part of the thesis, Chapter 4 discusses the implementation design of the Object Manager. It first outlines the major design guidelines and system requirements. Section 4.1 is broken into seven subsections. It focuses on object representation in memory and the physical implementations of the data structures used in the system. The implementation algorithms for inheritance and behavior are addressed in subsequent sections. Issues and features which are of direct concern to system designers are also highlighted. Chapter 5 discusses the implementation design of the Storage Manager. Section 5.1 first gives a brief comparison between the two most common persistence models. The persistence model for POServer is then presented. Section 5.2 describes in detail the logical and physical OIDs. The object manager interface routines are presented in Section 5.3. Finally, Chapter 6 concludes with some suggestions for enhancing the POServer system.

Chapter 2

System Architecture Overview

Before presenting the details of the conceptual object model in Chapter 3, this chapter begins by giving an abstract view of the system architecture and how the components of this architecture interact. The purpose of this chapter is to outline and define the functional specifications for each component in the POServer system. The POServer system architecture is based in part upon Iris architecture as presented in Fishman et al. [22]. It adopts a three-schema architecture and shares very similar functionality as found in most conventional database systems. The top layer represents the external model which links the communication between application users and the system. This is usually implemented as a graphical user interface (GUI) or a language interface, consisting of a set of programmatic and interactive interfaces that supports the Data Definition Language (DDL) and the Data Manipulation Language (DML) as illustrated in SQL/DS [16]. The middle layer represents the conceptual or logical model. It supports the core data model and is independent of how information is stored. The physical model is represented at the bottom layer which is a low level disk manager that performs storage and file access management. It defines the internal representation of information in the system. The architecture consists of four major components: 1) Graphical User Interface; 2) Language Interface; 3) Object Manager (OM); and 4) Storage Manager (SM). These four

components form the foundation of a general-purpose object-based system. Figure 2.1 depicts an overview of the POServer system architecture.

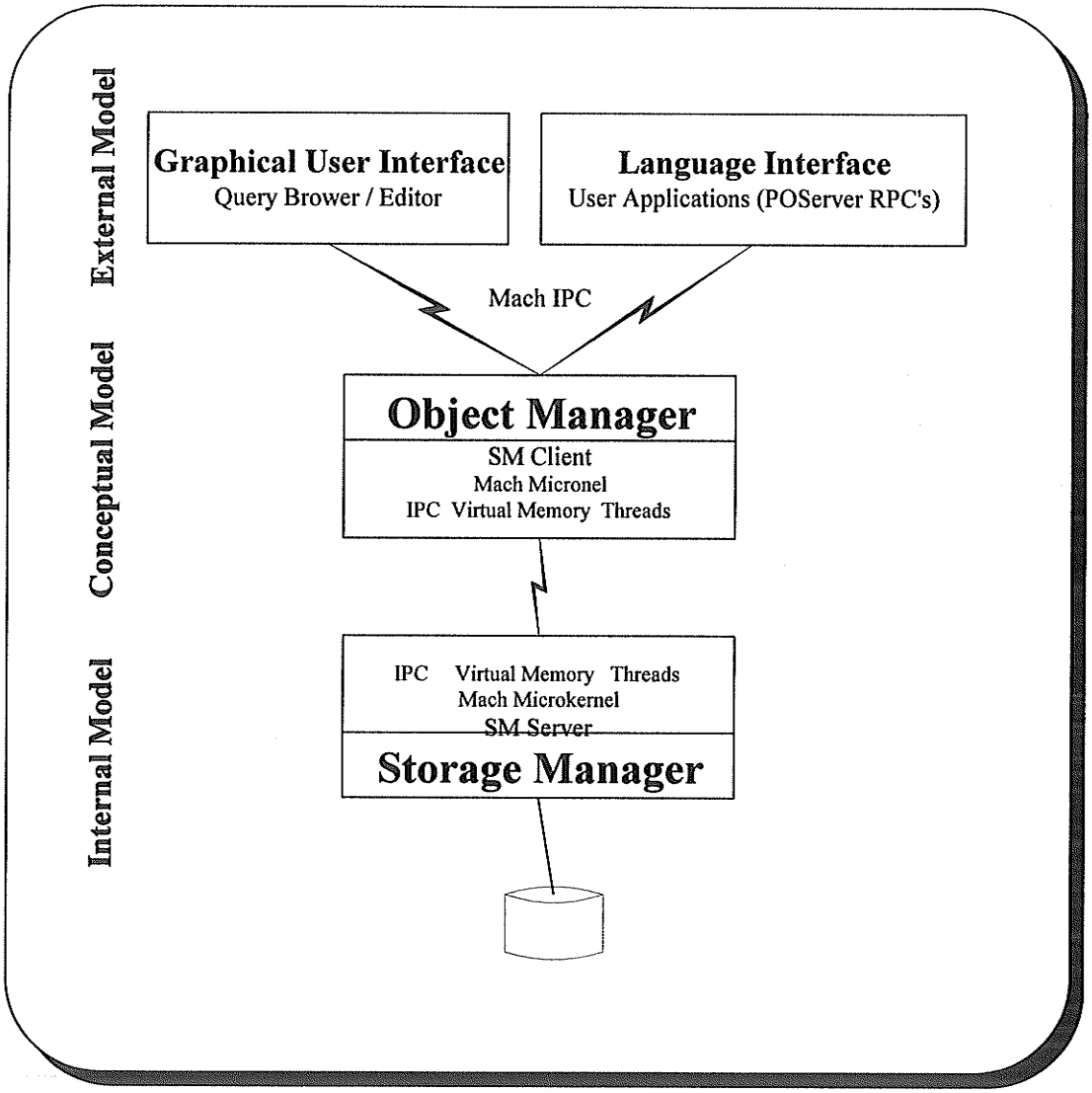

Figure 2.1 The POServer System Architecture

This basic system architecture may be extended in several dimensions to make it a more complete object-based system. First, integrity features such as transaction management and concurrency control may be added. Zapp [60] provides a model of transactions in an object-based system. Second, performance-related features such as secondary indexing and query optimization may be included. Third, the architecture may be extended with security and authorization for a multi-user environment. A model of authorization for object-based systems has been discussed in great details in Rabitti's [48]. The POServer architecture is unique in the following respects. First, the overall system architecture is based on a client/server (C/S) model. POServer's C/S model is supported by Mach's *Remote Procedure Call* (RPC) mechanism. All interfaces are built as clients to the lower level components in the hierarchy and they always communicate to each other by RPC's. As mentioned in Mullender [41], a remote call has the same semantics as a local call so the server may be running on the same or a different machine. The RPC mechanism can determine where the requested procedure is located on the network, translate the generalized call into the appropriate machine or OS specific call, and return the values to the caller via the network. The client/server model enables users to efficiently utilize the available computing power. Second, the advanced features provided in the Mach microkernel such as interprocess communication and memory management are fully utilized in the system. This greatly reduces the complexity of the Object Manager as compared to O_2 [19] where a dedicated communication manager is needed to handle the passing of messages. Third, a normal storage manager includes a number of facilities

such as disk space management, clustering and recovery, etc. For the purpose of this thesis, only the features necessary to make an object persistent are demonstrated. To provide a better understanding of the POServer architecture, the responsibilities of each component of the system architecture are we described in the following sections.

2.1 Application Programming Interface

All database systems require a mechanism for describing new structures to the database and interacting with the metamodel. The mechanism can be graphical, a declarative DDL, or an application programming interface (API). A subset of "standard" SQL commands comprise the DDL for Relational DBMSs. The traditional approach to language interface taken by relational DBMSs is by defining a sublanguage (e.g. SQL) to be embedded in a general-purpose programming languages such as COBOL or C. The combined source code is passed through a pre-processor and translated into subroutine calls in the syntax of the host programming language. The generated object code is then linked with the DBMS runtime library. This approach was adopted by SQL/DS [16]. A standard DDL does not exist for object DBMSs, although a number of choices are competing for adoption. Some object DBMSs do not use a specific data manipulation language to store and retrieve information, but use the OO programming language directly, for example C++ or Smalltalk. This means that a specific interface to the object DBMS is not necessary. These systems are thus closely coupled with the object

programming language. The class declaration syntax of the C++ programming language is currently the most common DDL for object DBMSs. Smalltalk also has an advantage as a DDL since it is extensible. Despite all this we should bear in mind that different languages are appropriate for different jobs. Most programming languages are built to achieve a specific set of goals or requirements. C++ and Smalltalk alone are not sufficient for writing all applications and in many situations an interactive ad hoc query language is more desirable. For POServer, the object model serves as the core of the system and multiple language interfaces can be developed on top of it. This approach provides openness to other languages and the ability to reuse existing applications. Therefore, classes created by one programming language may be used by another programming language, instances populated by one language may be read by another language, instance methods may be executed by multiple languages.

There are three main steps in object manipulation. First, a scheme for the class is designed, which defines the objects and their attributes. Second, the class is populated with objects. Third, the objects are queried for processing. For the illustration of the POServer API, an object query language similar to the ones used in Iris [22] and O_2 [19] is chosen. The POServer API consists of three types of statements: 1) Object Definition Language (ODL); 2) Object Manipulation Language (OML); and 3) Object Query Language (OQL). ODL is the language interface to be used by application designers and has constructs for defining and generating new classes, behaviors, and relationships

among classes, e.g. subclass and superclasses. For example, CREATE CLASS and CREATE METHOD statements. Once an application model is established, the next step is to populate the class and update the data and code contained within the system. OML is the language interface to be used by programmers and has constructs for generating new objects and manipulating existing objects, for example, CREATE INSTANCE and EXECUTE METHOD statements. OQL is a declarative SQL-like query language. It takes the well known SELECT-FROM-WHERE structure of SQL by adding the object extensions, IN collection or list of aggregates, as follows:

query result = SELECT <list of objects>
FROM <range variable> IN <collection/class>
WHERE <predicate>

OQL's syntax is simple and flexible. It provides very high level operators that enable users to sort, group, or aggregate objects. This syntax makes it possible to select any set of attributes or functions in the list of objects from a class or from any aggregate in the system. One major difference between relational SQL and OQL is the use of dot notation to navigate through objects. OQL takes advantage of the semantic knowledge built into object structures, for instance, if we have an Employee "john" and we want to know the name of the Department he works in, the OQL is john.dept.name. As another example, to get the names of the employees of the Department "d254".

SELECT e.name

FROM e IN d254.emps

This query inspects all employees of Department d254 and its result is a collection of employee names. Other object extensions to SQL include the use of methods in projection lists and WHERE clauses to take advantage of the implementation hiding inherent in the object approach, for example, WHERE e.age > 65. The notion for calling a method is exactly the same as for accessing an attribute or traversing a relationship. This flexible syntax frees the user from knowing whether the property is stored (an attribute) or computed (a method). More information about object query languages can be found in Iris [22] and O_2 [18, 19]. The POServer system is accessed by a remote procedure call interface called POServer(aHandler, anAPIstmt, listofargs). A complete C program with POServer API is illustrated in Figure 2.2. It shows how a POServer schema is declared and how to populate and manipulate objects in the system.

The Query Browse/Editor is a facility which allows users to retrieve and update object values and metadata with graphical and forms-based displays. The user interface accepts input from a workstation, provides a graphical interface, and formats queries for the underlying system. This kind of tool is essential because it facilitates rapid prototyping and increases the speed of application development. It also enables ad hoc query and reporting systems because application users are not forced to write, compile, link-edit, and debug a program just to get the answer to simple queries.

```
#include <POServer.h>                    /* Contains POServer data structures */
main () {
        POShanlder    myHandler;    /* Contains return code & error messages */
        int           tempAge;
        char          * tempName;

/* Creates a new class Department */
POServer(myHandler, "CREATE CLASS department ATTRIBUTES(
                                name        String,
                                number      Integer,
                                location    Char(4),
                                emps        Set", 0);

/* Creates a new class Employee */
POServer(myHandler, "CREATE CLASS employee ATTRIBUTES(
                                name        String,
                                dob         Date,
                                dept        Department
                                job_grade   Integer", 0);

/* Creates a new Department d254 Technical Services */
POServer(myHandler, "CREATE INSTANCE department :d254
                        ATTRIBUTES(name, number, location, emps)
                        VALUES('Technical Services', 254, '5N', Nil)", 0);

/* Creates a new Employee instance John Smith */
POServer(myHandler, "CREATE INSTANCE employee :john
                        ATTRIBUTES(name, dob, dept)
                        VALUES('John Smith', Date'1960-01-01', :d254,10)", 0);

/* Creates a new Method for Class Employee */
POServer(myHandler, "CREATE METHOD age FOR employee
                        RETURNS(Integer)", 0);

/* Returns John's name and age */
POServer(myHandler,"SELECT john.name,john.age",2,&tempName,&tempAge);
printf("Name  : %s, Age : %i", tempName, tempAge);

/* Makes d254 and John persistent */
POServer(myHandler, "EXECUTE METHOD john.persistent", 0);
POServer(myHandler, "EXECUTE METHOD d254.persistent", 0);
        }
```

Figure 2.2 POServer C Application Programming Interface

2.2 Object Manager (OM)

The Object Manager (OM) is a program that directly supports and implements the POServer object model. It corresponds to the query processor of the relational system (e.g. Relational Data System [16]) which performs query compilation and optimization with a flexible rule-based optimizer. The OM validates and executes the RPC calls, and returns the data in a form that is usable by the clients. The actual query processing and access path evaluation involve typechecking and execution plan generation. Data access requires a means for specifying what data to access and mechanisms to ensure that the correct data is obtained efficiently. The query optimizer chooses an efficient access path for the query using information about the structure of the query, the size of the objects, the number of instances, and the indexing or clustering strategies. Data independence is supported so the presence or absence of indexes in the physical database and the application programs will still run regardless of changes to file storage organizations. The OM achieves a higher level of data independence than relational systems by providing data abstraction and information hiding. The internal structure of an instance variable, the implementation of a function or the class relationship can be changed entirely transparent to the existing applications. Query optimization will not be addressed in this thesis. Musteata [42] describes query optimization in considerably more detail.

The object structure and the corresponding operators are supported by the Object Manager. It is the OM which attaches the object semantics to the bytes returned from the

Storage Manager and presents them as an object to the client modules in the higher level of the hierarchy. For example, system defined classes such as Date, Time, Bag, List, and Set are manipulated by the Object Manager to hide the internal implementations. Date is presented as a Character string of the form 'YYYY-MM-DD' but stored as a 4-byte decimal internally. Through the use of ODL all object definitions and class relationships are defined and stored in the system. This information is often referred to as meta-data which is a schema of classes that make up the database. The OM is responsible for schema management and uses the information for typechecking and validation. For instance, we define two new classes Canadian$ and US$. Both are actually Real data types. However, each is regarded as a separate and distinct data type. An application would fail if it tried to add a Canadian$ to a US$ because of data incompatibility. This concept is known as strong typing in the OO paradigm and is enforced by the OM.

In relational systems, Codd [14] introduced the notion of user-defined identifier keys to represent the identity of an item. An identifier key is some subset of the attributes of an item which is unique for all items in the relation. There are several problems with identifier keys because the concepts of data value and identity are mixed. In POServer, each object stored in the system has a system-provided, unique handle called an object identifier (OID). OIDs relieve users from creating unique keys to identify stored information. The OM is responsible for maintaining the OIDs of persistent objects and providing a correspondence between OIDs and objects on disk. The user application may

request objects in the system from OM by passing it an OID. The OM elementarily supports calls to get, put, and delete objects by OID, for instance, DELETE OBJECT WHERE OID = '00001002:00000010'.

2.3 Storage Manager (SM)

The Storage Manager is responsible for the management of persistent objects on secondary storage and the management of cache for the SM clients. This includes allocation and deallocation of pages on disk, movement of pages to and from disk and the cache area, object clustering and indexing on collections etc. The SM corresponds to the Data Manager (DM) layer of SQL/DS [16] and performs very similar functionality. The SM handles all physical level details and deals only with pages. The POServer SM is essentially a page server. The SM server stores and retrieves pages of data in response to request from the OM. Requests to SM might include reading or writing data and code within the object-based system. The Storage Manager only interacts with the OM and acts as a server to OM that requests persistent object services. The main advantage of a page server architecture is that it has no knowledge of the contents of a page and does not understand the semantics of objects. Since entire pages are transferred between the OM and the SM server, the overhead on the server is minimized. This makes the SM very simple and can support more object managers concurrently. The SM simply passes pages to and from the OM, and stores them on disk.

The SM maintains a cache area in the OM, a pool of object pages that have recently been used. The cache is maintained according to a "least recently" used policy: the least recently used page is replaced when a new one has to be fetched into the cache. When an application signals a memory fault, the OM determines whether the page being accessed is in its cache. If the OM cannot locate a particular object in the cache, it generates a page fault to the Storage Manager. The SM server transmits the page and puts it into the cache. The address of the page containing the requested object is returned by SM and then an offset is added to it so the OM can get the particular object.

2.4 The Benefits Of Mach

As illustrated in Figure 2.1, the entire system architecture is based upon Mach. Mach's microkernel architecture has important advantages for users and applications over today's monolithic operating systems such as UNIX. Robustness, scalability, maintainability and extendibility are just some of the more noticeable ones. Mach provides five different classes of services: 1) Virtual memory management; 2) Tasks and threads; 3) Interprocess communications (IPC); 4) I/O support and interrupt management; and 5) Host and processor set services. Mach was proposed and chosen because its primary services support distribution and the level of control required for the implementation of object and storage management in a client/server platform. The Open Software Foundation (OSF)

Research Institute has been a proponent of Mach technology. The industry, through OSF and others, has entered into agreements with Carnegie Mellon University to license Mach technology and contribute work back into Mach. This represents one of the essential benefits of open systems, with work from many resources benefiting all. As a result, Mach receives wide acceptance in the research and academic communities. The advantages of using Mach as the foundation of the POServer system is summarized below.

2.4.1 Microkernel Concepts

Mach adopts the microkernel approach. It is designed to isolate the most essential services and platform-specific functions of an operating system in a small core of code that runs in the most privileged state of the computer. The rest of the system is supported as set of applications running in nonprivileged space (user space), isolated from the kernel by a clearly defined set of interfaces. The limited set of well-defined interfaces enables orderly growth and evolution. The system can be enhanced with new functionality in a modular fashion without retesting and rebuilding everything. Services that were traditionally integral parts of an operating system such as file systems and windowing systems are becoming peripheral modules that interact with the kernel and each other. This reduces the size of the OS code running in kernel space and maximizes the amount of space available for user applications. The microkernel approach makes the

machine-dependent modifications for different architectures easier. CMU has successfully implemented support for DOS, UNIX, OS/2 and MacOS on top of the Mach 3.0 kernel. The ability to concurrently support multiple operating systems, such as UNIX and DOS has been demonstrated in [24, 38].

CMU is distributing the Mach kernel, libraries, PMAX etc. (no license required) to outside research groups. The distribution is done by electronic transfer over the Internet using the software upgrade protocol (SUP) [52]. SUP is a client program, run by system maintainers, which initiates the upgrade activity on a machine requesting the latest version of a collection of files. Arrangements with CMU have been made and an encryption key for starting SUP has been obtained. The netcrypt.c file has been installed so that periodic SUP updates can be made to keep up with bug fixes, additions and other changes. Instructions for setting up and building Mach 3.0 for different types of computers such as DecStation 5000, i386, Sun3, MircoVax etc., are distributed from CMU and documented in [57, 58]. These documents explain exactly what is required to build the Mach microkernel and the Unix-server; to set up the directory structure ./RFS/.LOCALROOT; and to boot a Mach 3.0 system up multi-user. The Mach 3.0 kernel plus the Unix-server is compatible with UNIX BSD 4.3 programs. The emulation uses the original AT&T and U.C. Berkeley source code and is complete enough to run executable files compiled and linked under BSD UNIX.

2.4.2 Support For Client/Server Development

Message-based interprocess communications (IPCs) and threads are primarily of interest to system designers of client/server development. Mach is a message-passing kernel that supports the client/server paradigm, and makes heavy use of shared virtual memory to facilitate communication between tasks. IPC is the mechanism whereby the different parts of distributed applications can communicate. The IPC facility allows clients and servers to call each other and exchanges data regardless of where they are executed in a network configuration. Message passing is a very natural way to structure systems in which components are distributed over a loosely-coupled set of individual processors because it enables location and distribution transparency. Without Mach's IPC support, system designers would have to be concerned about the low level networking protocols (e.g. TCP/IP) and their compatibility; and programmers would have to write applications through socket interface. Mach provides a capability-based interprocess communication facility. IPC facility in Mach is integrated with the virtual memory system and capable of transferring large amounts of data. Message passing can take the form of simple send/receive protocols (e.g. system calls *mach_msg_send(request_msg)* and *mach_msg_receive*(reply_msg)) or simple send/receive messages can be combined into a form of remote procedure call (RPC) to better suit client/server types of communications. RPC provides a standardized network communication process interface that can hide the network details that a developer would have to know to make use of network or distributed resources. In order to help clients find servers in a flexible and portable

manner, Mach provides a name service called *NetMsgServer* to store network communication information. Because the mircokernel does not need to know whether the message comes from a local or remote process, the message-passing scheme offers an elegant foundation for RPC's. Also, Mach IPC messages are typed collections of data. This allows the microkernel to perform data conversion and achieve heterogeneous operations.

The traditional UNIX process is divided into two separate components in Mach. The first is the task, which is a basic unit of resource allocation that includes a page address space, protected access to system resources such as processors, ports, and memory. The second is the thread which is a basic unit of CPU utilization. A Mach task may have many threads of execution, all running simultaneously. Threads are lightweight processes that share a single address space and the task's resources. A task is a passive collection of resources; threads are active entities in Mach because they execute instructions and manipulate their registers and address spaces. Most importantly, system designers can now easily develop concurrent program by using multiple threads of control. This often results in better performance than would be possible without threads. The ability to use threads is critical in some applications, for instance, multithreading a database server can offer increased concurrency and parallelism. Mach provides the system call *mach_thread_create*(parent_task, &new_thread) for requesting a new thread and the thread can be terminated by calling *mach_thread_terminate*(new_thread). Threads in

different tasks communicate with each other by exchanging messages through a communication port. The system call *mach_port_allocate*(parent_task, &new_port) is used to allocate a new port and the port can be deallocated by calling *mach_port_deallocate*(parent_task, new_port). Mach also allows a set of ports to be grouped together and a single *mach_msg_receive* system call can then read the first available message from any of the ports in the set.

2.4.3 Support For External Pager

For efficient implementation of the persistent object store in a client/server environment, some control at the virtual memory level is needed, that other operating systems like SUN OS do not provide. Mach has a powerful and highly flexible memory management system based on paging. It supports large, potentially sparse address spaces with flexible memory-sharing; copy-on-write virtual copy operations; read/write memory sharing between tasks; and memory mapped files. *vm_allocate*(target_task, &new_memory, vm_page_size, find_space) allocates a chunk of memory. vm_page_size is the default system page size and find_space controls the allocation of the virtual memory. *vm_deallocate*(target_task, new_memory, vm_page_size) returns the virtual memory back to the operating system. Copying data within the virtual memory of the current task can be done using *vm_read, vm_write* and *vm_copy*. The notion of a memory object is generalized to allow general purpose user-state external pager tasks to be built. Mach

also permits user-level (also known as external) pagers to manage memory regions. Mach microkernel and the memory management facility communicate through a well-defined protocol, making it possible for users to write their own memory managers. Users can control memory sharing and paging operations directly, for instance, *memory_object_data_request* and *memory_object_data_provided* are used for page-in and page-out operations. Main memory may be used as a cache for user-level data objects, such as databases and files. This ability allows database designers to implement paging systems with very special requirements. For a more complete discussion of the concepts in Mach, the reader should see Accetta [1] and Black [7], and for details of specific system calls, Baron [5] and Walmer [59].

2.5 POServer RPC Implementation

Some approaches to providing object persistence proceed by modifying existing compilers [15] to provide as clean an interface as possible. Instead, Mach's Remote Procedure Call (RPC) mechanism is used because it fits well into the client-server architecture. The interface for Mach 3.0 is currently only available in C, therefore the POServer interface is also illustrated using C. It is not necessary to use any object-oriented language to implement the interface or the OM as long as the programming language is capable of building the foundation system. This has been proven in the case of GemStone[10] which was implemented using C and runs on most

UNIX environments such as Sun3 and DEC. In this section, Mach's RPC mechanism and how it can be effectively used to implement the POServer API is illustrated. This should help to make the abstract discussion of the core object model in next chapter concrete.

The POServer RPC is the high level programming interface to the POServer system. The RPC can be defined with the following C language specification:

```
POServer( POShandler aHandler, Char *APIstmt, Int numargs, listofargs );
        POShandler aHandler /* contains return code and error messages */
        Char *APIStmt       /* ODL, OML or OQL statement */
        Int numargs         /* number of arguments, 0 if no arguments */
        listofargs          /* optionally include one or more arguments */
```

aHandler is a POServer data structure which contains information about the RPC call statement. *APIstmt* is any valid OML, ODL or OQL statements. The POServer RPC can optionally include one or more arguments. *numargs* indicates the number of additional arguments to be passed and should be 0 if no argument is required. *listofargs* is implemented by calling the C macro *va_arg*() which allows a variable number of arguments to be passed to a function. The most common example of a function that takes a variable number of arguments is *printf*() [50]. The OM validates and interprets the API statement. It then parses the list of parameters using the C macro *va_start*(argptr,

numargs). This macro returns a pointer to *numargs*. The subsequent parameters are then retrieved via calls to *va_arg*(argpr, string) in a loop.

Every remote procedure must be pre-defined so that the client and server programs follow the same protocols when communicating to each other. Mach provides an interface generator called Mach Interface Generator (MIG). The POServer RPC can be defined using a C-like language called the Interface Definition Language (IDL). The IDL is then used to generate three outputs: 1) a client stub, 2) a server stub and 3) a C header file. The generated client stub contains the client code supporting the POServer(aHandler, APIstmt, listofargs) function. The server stub contains the OM server side code to make an upcall to the POServer function, which executes in the server address space. The C header file contains the type and data structure definitions used in the client/server interface. The user application program written in C source code can invoke the POServer RPC directly. The user application is then linked to the client stub and the RPC runtime library, generating an executable POServer client module. Development of the OM server is accomplished in exactly the same way, using the same header file as the client; the only difference is that the OM server code is linked to the server stub, rather than the client stub. By using MIG, the client application is guaranteed to communicate with the OM using the same specification. The IPC calls *mach_send_msg* and *mach_receive_msg* are hidden in the client and the server stubs respectively and are

transparent to application users. The RPC mechanism for the POServer RPC is illustrated in Figure 2.3.

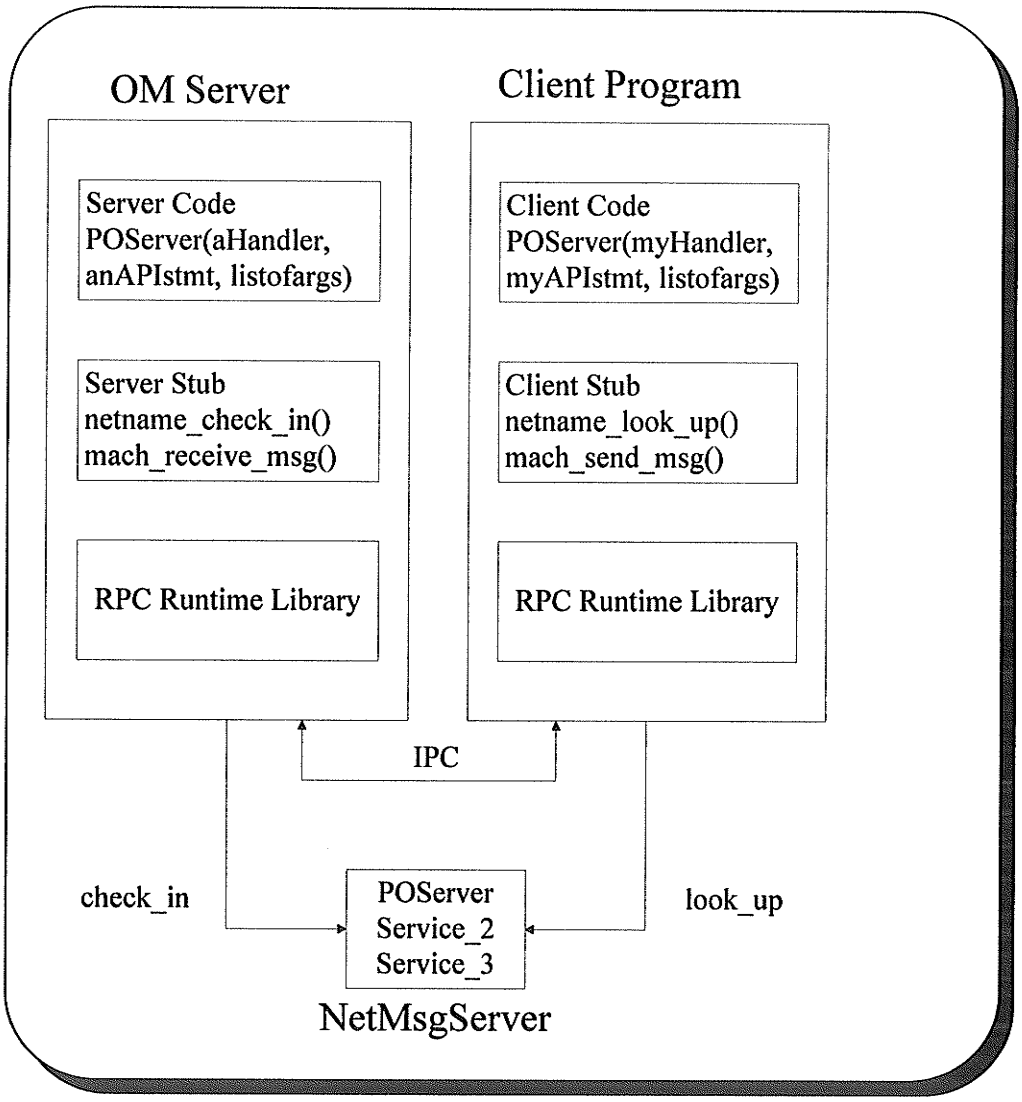

Figure 2.3 The POServer RPC

Notice that in Figure 2.3, there are two functions called POServer. The client application uses the POServer function just like a local procedure call because it is actually a remote

call to the OM POServer function which carries out the real operation. The OM server process the API request, provides the result, and passes it to the server stub, to RPC, and back to the application program via the OM client stub. The OM server must make the POServer RPC information available to application clients. The network messaging mechanism is based on a name service provided by Mach called network message server, *NetMsgServer*. By using *NetMsgServer*, a client may obtain the required service without having to know how to talk directly to the server. The OM server stub first registers the POServer function and the OM server port to the name server by calling the system function, *netname_check_in*, with the following parameters:

netname_check_in(NetMsgServer_port,"POServer", task_self(), OM_server_port)

The NetMsgServer_port is a global variable initialized by crt.o during program creation and initialization. The network information is obtained during system startup. The second argument is the name of the function, POServer, that is going to be registered. The third argument, task_self() is known as the signature which prevents unauthorized tasks from deleting a service from NetMsgServer because the same signature must be given when calling the system function *netname_check_out*. The client stub later uses another system function, *netname_look_up*, to locate the POServer function and obtain a send right to the OM server port:

netname_look_up(NetMsgServer_port, "POServer", &OM_server_port)

Chapter 3

The POServer Object Model

Database researchers disagree as to what constitutes an object-based system because various definitions exist. Although the object approach does not yet include a clearly defined data model, there is general agreement on the concepts and basic capabilities an object-based system should provide. One of the first attempts to define the requirements for an object-oriented (OO) DBMS appeared in a paper by Bancilhon, et al. entitled "The Object-Oriented Database Manifesto" [3]. According to [3], an object model seeks to group objects into similar classes that have common attributes and behaviors, and to factor common behavior up and out into more abstract representations. To develop applications in an object model, we identify objects, describe their behaviors, and then allow them to interact by passing messages.

In this chapter, constructs and vocabulary found in Iris [22] and O_2 [18, 19] are utilized to form a conceptual object model. The POServer object model is based on a set of fundamental OO concepts common to most OO programming languages; it has been particularly influenced by Smalltalk [23]. Prior to the discussion of system implementations in next chapter, it is important to examine the basic concepts of object orientation that form the foundation of an object-based system. This chapter establishes a set of terminology that can be used in further discussions. Further, an object model is one

which includes, at the minimum, the core OO concepts discussed and justified in this chapter.

A relational data model supports a predefined set of data types and a relational system is viewed as a collection of tables. An object model supports a user-defined extensible set of data types and can model complex data structures that closely match real-world entities. It can be regarded as a prescription for how objects might be represented and how that representation might be manipulated. The POServer object model is based upon three major constructs: *objects*, *methods*, and *classes*. The following aspects of the object model are considered: 1. Object representation; 2. Behavior representation; 3. Class representation; 4. Class relationships; 5. POServer class hierarchy; and 6. Referential integrity. This chapter will examine each of these aspects in turn. Using this object model, systems designers can directly model real world applications in an intuitive way. Six basic OO concepts will be discussed: *object identities*, *data encapsulation, class hierarchy, class inheritance, polymorphism*, and *data abstraction*. In broad terms, the POServer object model is used to describe conceptually the objects in an application system, relationships between the objects, and the attributes and methods that characterize each class of objects. Although the OO paradigm has been around since Simula [17] in the mid '60s, it is still relatively immature and there is no universal agreement on how to characterize it. However, other definitions on the OO model can be found in Barker et al. [4], TIGUKAT [26], ORION[30] and Eiffel [39].

3.1 Object Representation

For information modelling, the term *object* means a representation of some real-world thing. For object-oriented programming languages, the term *object* means a run-time instance of some processing and values. Objects are the units into which we divide the world, that is, the molecules of the object model. For the POServer object model, each object has two aspects: 1. *Identity*. An identity is that property of an object which distinguishes each object from all others. 2. *State*. A state is the set of values for the attributes of the object. The term *attribute* is from relational systems [16] and is an *instance variable* in Smalltalk [23]. Let I be the domain of identifiers used to reference objects. Let D be the union of system-defined fundamental types and user-defined *abstract data types* (ADT's). Fundamental types are {Boolean, Real, Integer, Char, String, Date, Time, Oid}. Let A be the domain of attribute names. Thus, an object is:

Definition 3.1.1 An object is simply a pair o = (oid, s) where:

1. oid is the unique identifier of o and oid $\in$ I.
2. s is the state or a set of values, it can be one of the following:

    2.1 An *atomic value* is an element of D.

    2.2 $[a_1 : v_1, ..., a_n : v_n]$ is called a *tuple value*, where $a_i \in A$ and $v_i \in D$.
    [] represents an empty tuple value.

    2.3 $\{v_1, ..., v_n\}$ is called a *set value* where $v_i \in D$.
    {} represents an empty set value.

    2.4 The special value Nil, represents an *undefined object*. ●

The following are examples of Department and Employee objects:

Department = (c_1, {d_1, d_2, d_3, d_4})

Employee = (c_2, {e_1, e_2, e_3, e_4, e_5, e_6, e_7})

(d_1, [**name**:'Technical Services', **number**:254, **location**:'5N'], **emps**: {e_1, e_3, e_4})

(d_2, [**name**:'Application Services', **number**:130, **location**:'5S'], **emps**:{e_2, e_5})

(d_3, [**name**:'Operations Services', **number**:112, **location**:'3W'], **emps**:{e_6, e_7})

(d_4, [**name**:'New Department', **number**:123, **location**:Nil], **emps**:{})

(e_1,[**name**:'John Smith', **dob**:'1961-01-01', **job_grade**:10, **dept**:d_1, **age**:34, **salary**:50000])

(e_2, [**name**:'Dave Jones', **dob**:'1962-01-01', **job_grade**:9, **dept**:d_2, **age**:33, **salary**:45000])

(e_3, [**name**:'Rob Webb', **dob**:'1963-01-01', **job_grade**:8, **dept**:d_1, **age**:32, **salary**:40000])

(e_4, [**name**:'Jim Wong', **dob**:'1964-01-01', **job_grade**:6, **dept**:d_1, **age**:31, **salary**:30000])

(e_5, [**name**:'Pat Lee', **dob**:'1965-01-01', **job_grade**:7, **dept**:d_2, **age**:30, **salary**:35000])

(e_6, [**name**:'Donna Ma', **dob**:'1965-01-01', **job_grade**:7, **dept**:d_3, **age**:30, **salary**:35000])

(e_7, [**name**:'Ken White', **dob**:'1966-01-01', **job_grade**:6, **dept**:d_3, **age**:29, **salary**:30000])

Definition 3.1.2 Two objects o_i and o_j such that $i \neq j$ are identical iff o_i.oid = o_j.oid. ●
The definition states that two objects are identical if they have the same object identifier.

Definition 3.1.3 Two objects o_i and o_j such that $i \neq j$ are equal iff o_i.state = o_j.state. ●
The definition states that two objects are equal if they represent the same identical semantics.

Objects are implemented internally in POServer as a number of variables which store information and a set of methods which operate on that data. It is important to note that the relational tables can only have single-valued columns whereas the state in the POServer object model is not restricted to atomic values or system-defined types. It is possible to have tuple values and set values. This implies tremendous modelling power because it means that any repeating values occurring naturally in the application can be represented directly in the POSever system without having to be normalized.

3GL programming languages use variable names to identify temporary objects. This mixes addressability with identity. Relational systems are value-based in which identifier keys constitute part of the state of a record and is used to identify persistent objects. Both of these approaches compromise identity. In contrast, the POServer system is identity-based. An object's identity is independent of state, type, and addressability. All objects in the system have identity and are distinguishable. Uniqueness is enforced via system-maintained *object identifiers* (OIDs). An OID is permanently associated with an object. An object is given a unique OID at its creation and keeps it until it is destroyed, and is never reused. The object identifiers can be used in a variety of ways, for example, to pinpoint and retrieve an object; and to perform sorting and ordering. For persistent objects the identity is maintained by the system across multiple programs or transactions. Identity also facilitates the notion of object sharing. Such sharing reduces the *update anomalies* [16] that exist in the relational data model.

3.2 Behavior Representation

The *behavior* of an object is the set of methods which operate on the state of the object. Using the POServer object model, the state encapsulated in an object is accessed from outside only through its behavior. POServer allows a method definition to be done in two steps. First, the programmer declares the method by giving its name and *signature*. The signature represents the external interface to users and is the only means whereby the contents of an object can be changed. Second, the implementation of the method is specified. The method implementation specifies what the method does and how the result is obtained when it is invoked. The separation of method declaration and its implementation provides a degree of data independence. Objects in the system communicate with each other by passing messages. The messages are uniquely identified by their signature. Messages and methods correspond roughly to procedure calls and procedures in conventional systems. Thus, methods enable *data encapsulation* by hiding the internals of the object implementation from users and exposing only the signature. Let C be the domain of all classes in the system. A method is defined as follows:

Definition 3.2.1 A method is a pair $m = (n, \delta)$ where

1. n is the name of the method.
2. δ is the signature of the method such that $\delta = (c, p, r)$ where

2.1 c is the class to which the method is attached, $c \in C$.
2.2 p is a possibly empty ordered set of parameters, $p \in C$.
2.3 r is the class of the result, also possibly empty, $r \in C$.

3. A method with its signature is expressed as follows:

$c.n(p_1, ..., p_n) \rightarrow r$ ●

Definition 3.2.2 If m is a method and c is a class of C, then m is defined in c if there is a method with signature : $c.n(p_1, ..., p_n) \rightarrow r$. ●

For example, **age** calculates the age of an employee based on his day of birth, **dob** and **salary** calculates gross salary based on the employee's **job_grade**. Therefore, we define **age** and **salary** with the following signatures:

Employee.**age** → Integer
Employee.**salary** → Integer
Employee.**dept** → Department
Object.**identical**(Object) → Boolean

e_1.**age** → 34
e_1.**salary** → 50000.00
e_1.**dept** → d_1
e_1.**identical**(e_2) → False

Methods **age** and **salary** are called *computed functions* because they are associated with executable code. The method construct can also be used to specify relationships among

classes. For example, method **dept** defines behavior on class Employee and specifies the relationship between an employee and his department. In contrast to **age**, **dept** is called a *stored function* because it is simply a reference to the class Department. The interface to each method is defined to reveal as little as possible about its implementation details and object structures. The object's behavior is defined by the abstract signatures. Note, Iris further classifies computed functions into *derived functions* and *foreign functions* [22]. Derived functions are computed by evaluating an Iris expression whereas a foreign function is implemented as a subroutine written in some general-purpose programming language.

3.3 Class Representation

The *class* construct allows similar objects to be classified. Each object in the system must know its class. A class is the description of the behavior and structure of a set of objects. This is referred to as *object type* in some systems, e.g. Iris [22] and TIGUKAT [26]. The class definition includes both the attributes and methods of the class. From the point of view of a strongly typed language, a class is a construct for implementing a user-defined *abstract data type* (ADT) and is a formal description of an entity. The class concept captures the *instance-of* relationship (this is called the *part-of* relationship in Bancilhon [3]) between an object and the class to which it belongs. Classes can describe and model concrete concepts such as Employee and Department, or more intangible

abstractions such as Stack or Queue. A class definition includes a *class name* (begins with a capital letter) and the following three aspects: 1. *Identity*. This is a unique identifier for the class since a class is also considered an object in the system. 2. *Structure*. This defines the object's internal representation, that is, its attributes. It represents the static part of an object. 3. *Behavior*. This defines the active part of an object and includes a set of methods and their internal implementation. Let A be the domain of attribute names such as **job_grade** and **location**. Let C be the domain of all classes in the system. A class is:

Definition 3.3.1 An class c = (oid, S, M) where:

1. oid is the unique class identifier of c, oid $\in$ I.
2. S is the object's structure and is defined with a set value $[a_1 : c_1, \ldots , a_n : c_n]$ where

   $a_i \in A$ and $c_i \in C$, also $\nabla\, a_i, a_j \in S, i \neq j \Rightarrow a_i \neq a_j$

   The definition states that a class structure requires each attribute be uniquely identified by its name. The domain of an attribute may be any class; system-defined or user-defined.
3. M is the object's behavior; composed of identifiable methods such that

   $\nabla m_i, m_j \in M,\ i \neq j \Rightarrow m_i \neq m_j$ ●

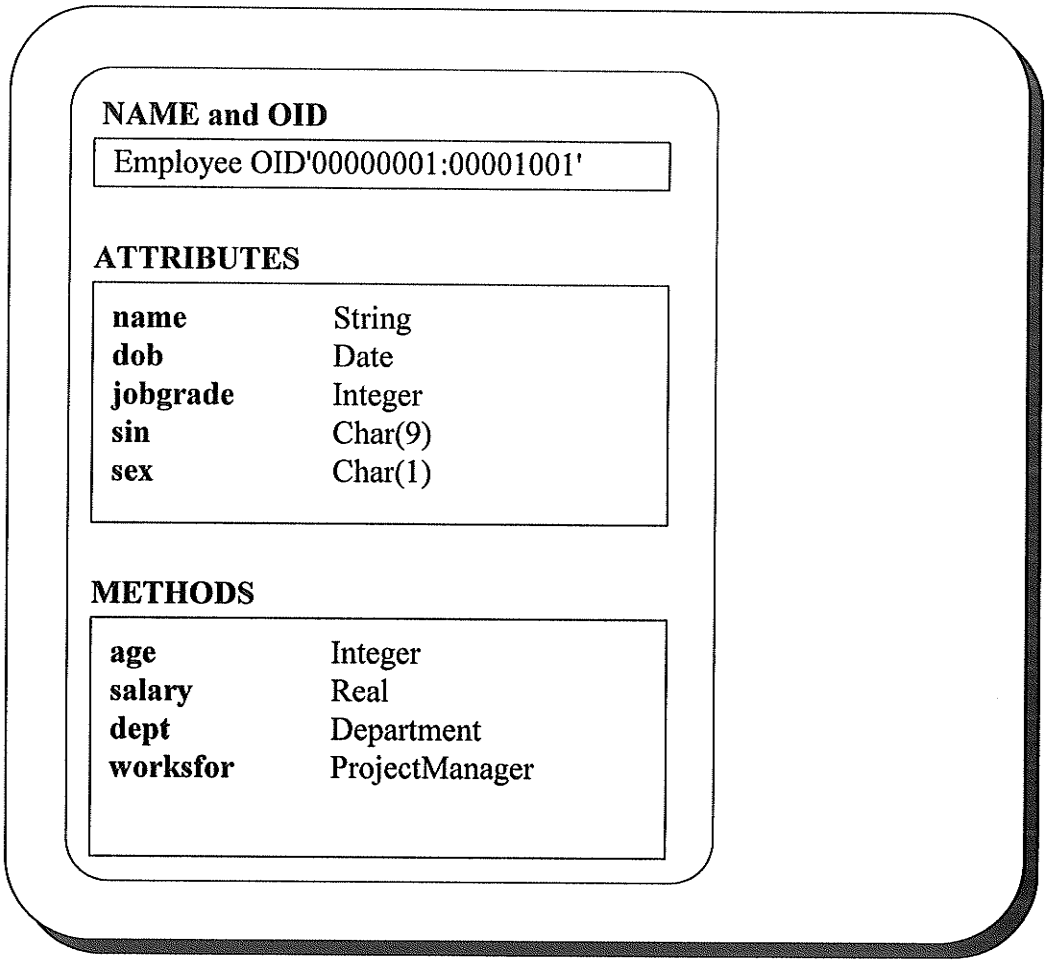

Figure 3.1 Class Employee

Figure 3.1 is a graphical representation of the class Employee. There are three distinct parts : 1. The class name and its OID; 2. A set of [attribute-name : attribute-class] pairs; and 3. A set of [method-name(method-parameters) : return parameter] pairs. The fact that the domain of an attribute may be an arbitrary class gives rise to the nested structure of the definition of a class. This also represents a significant difference from the normalized relational model where the domain of an attribute is restricted to a fundamental data type. When creating a class, the class definition can be momentarily

incomplete. For instance, class Department can be defined with an additional attribute **div#** of class Division which is not yet defined. Thus the class is behaviorally defined, but functionally incomplete. This capability offers great flexibility for schema evolution.

3.4 Class Relationships

Classes within a given problem domain are usually related in some way. A new class may be defined as a specialization of an existing class and additional attributes and methods may be added. This new class is a *subclass* of the existing class and the existing class is referred to as the *superclass* of the new class. In that case, all objects belonging to the subclass also belong to the superclass. The subclass is considered as a refinement of the superclass. This process is called *specialization* of entities and the related classes form a *class hierarchy*. The concept of a subclass specializing its superclass captures the *inheritance relationship*. The more general classes are placed higher up in the class hierarchy, whereas the more specialized ones are placed lower down. This is referred to as the *is-a* relationship in Banchilhon [3] and Eiffel calls this the *is-plus-but-except* relationship [39]. When classes are grouped into subclasses, subclasses inherit the attributes and methods shared by all instances of the superclass. By definition, inheritance means the "properties or characteristics received from an ancestor" [13]. *Inheritance* in the POServer object model allows us to specify common attributes and methods once and then specialize and extend those attributes and methods into specific

cases. Inheritance directly facilitates extensibility within a given system. POSever, like Smalltalk, supports only *single inheritance* [23]. That is, a class inherits behavior and structure from only one superclass. Some other systems (e.g. C++ and Iris) allow a class to have any number of direct superclasses and to inherit behavior from more than one superclass; which is referred to as the *multiple inheritance* capability [21, 22]. For systems that support multiple inheritance, several parallel inheritance hierarchies can exist and it is usually the responsibility of the class designer to avoid potential name conflicts. An object-based system should support at least single inheritance. Classification and inheritance are useful in organizing information. The POServer object model allows class designers to use generalization to organize an object space and then make use of inheritance to share and reuse code. *Inheritance, class hierarchy* and *instance-of* form the *structural relationship* of the object model. The structural relationship is central to the operation of the POServer system. It allows the schema to be better structured and gives tremendous modelling power to the class designer. The structural relationship concept also distinguishes the POServer object model from programming with ADT's.

The concept of class hierarchy and inheritance is demonstrated with a concrete example. Figure 3.2 illustrates the notion of inheritance and class hierarchy through a simple employee model within an Information Systems (IS) Division. In this example, class Employee is a superclass or a base class. Systems Engineer, Project Manager, and

Technical Consultant are all subclasses or derived classes (note that each subclass *is-a* Employee). These three *specialized* classes inherit all attributes and methods from the common superclass Employee.

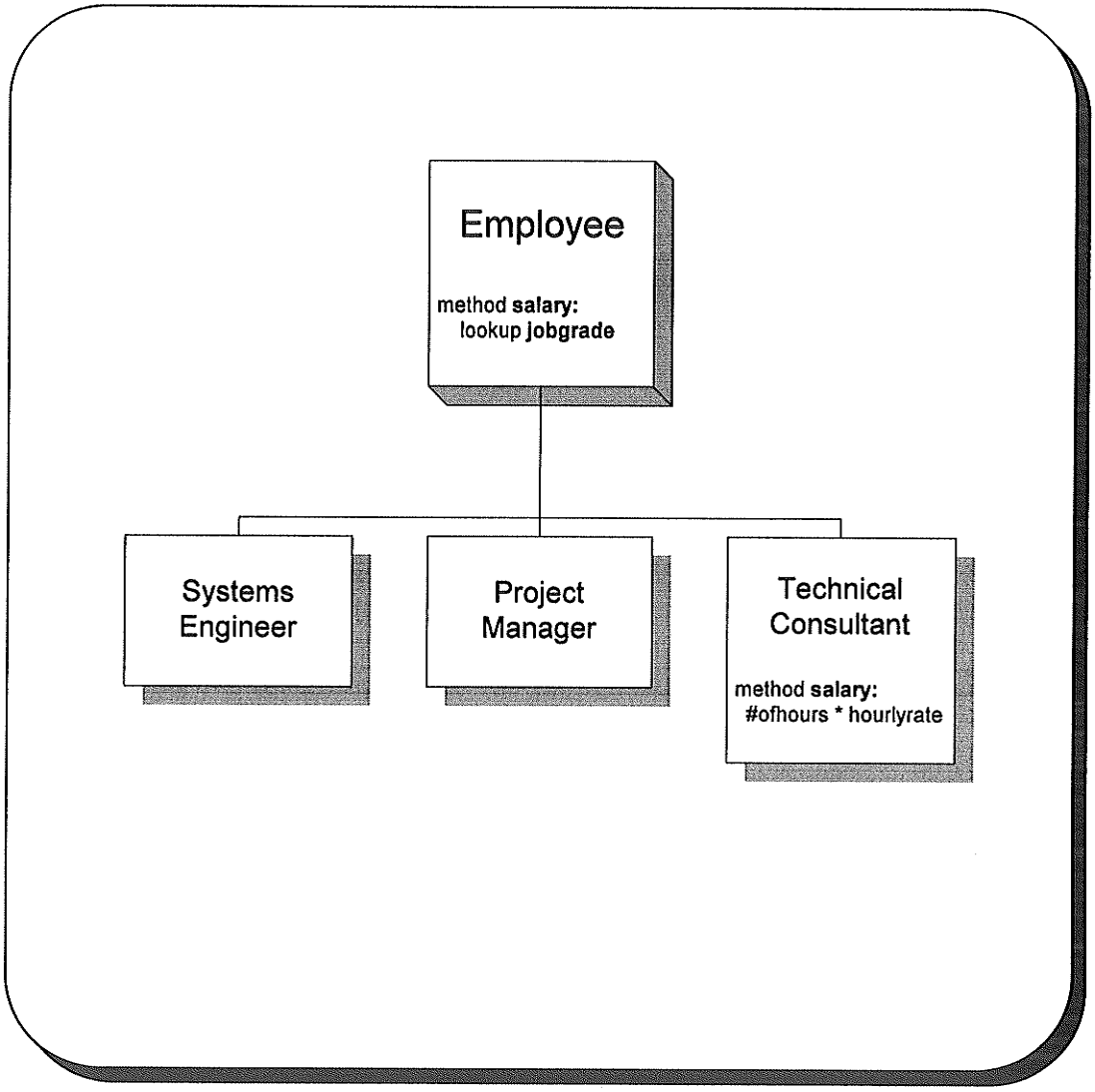

Figure 3.2 Employee class hierarchy

Therefore, the features common to all employees reside in class Employee (e.g. personal details) and features unique to specific classes of employees appear in individual subclasses. For example, a project manager must have at least two persons under his supervision and a technical consultant must specialize in at least one technical area. The class Employee can also be referred to as an *abstract class* if all employees in the IS division must exist in one of the subclasses. Abstract classes are classes which specify common behavior for its subclasses, but no instances of abstract classes may be created.

Definition 3.4.1 A method m is inherited from c where $c \in C$, if there is one superclass of c in which m is defined. ●

Besides inheritance, a method in a class may be overridden by a method of the same name defined on its subclasses which is sometimes called *overloaded.* Inheritance and overloading of methods gives rise to *polymorphism.* The word *polymorphism* originates from Greek and means 'to take many forms'. Polymorphism in the POServer object model allows different classes to use the same method name and signature (the number of arguments and the class of result value must be the same as defined in the superclass). Assume Technical Consultants are hourly employees rather than salaried employees, then the **salary** method in each employee object must know what class of employee it is before invoking the method. The **salary** method of the base class Employee uses **job_grade** to calculate the payroll, while the **salary** method of the class Technical Consultant works with number of hours and hourly rates.

Relational databases do not store explicit relationships between tables. Instead, relationships are formed by storing similar data values in table columns and relationships are made at runtime by *joining* tables together [16]. The POServer object model supports the *association relationship* to connect two or more object instances. This is referred to as the *object composition* or *composition through references* in O_2 [19]. This relationship is achieved by embedding references to objects within the state of other objects. For example, the class Employee has an attribute **dept** and its domain is class Department. This implies a link from class Employee to class Department. Similarly, the class Department has an attribute **emps** and it contains a set of Employee objects. This represent a one-to-many relationship between a department and its employees. A Nil pointer indicates no relationship. The POServer system supports access from one object to another by traversing a reference stored within an object. By capturing relationships directly, the POServer object model can much more easily represent complex data structures and nested relations, for instance, one-to-one, one-to-many, and many-to-many relationships can all be supported.

3.5 POServer Class Hierarchy

The POServer Class Hierarchy borrows most heavily from Goldberg and Robson's Smalltalk-80 Class Library [23]. In Smalltalk, there is only one inheritance hierarchy,

and thus one root class. POServer adopts the same approach. All classes are organized into a class hierarchy and inherit from a single root class Object, either directly or indirectly. Behavior, which is common to all classes, is collected and stored in the class Object. Figure 3.3 illustrates the class hierarchy for the POServer object model.

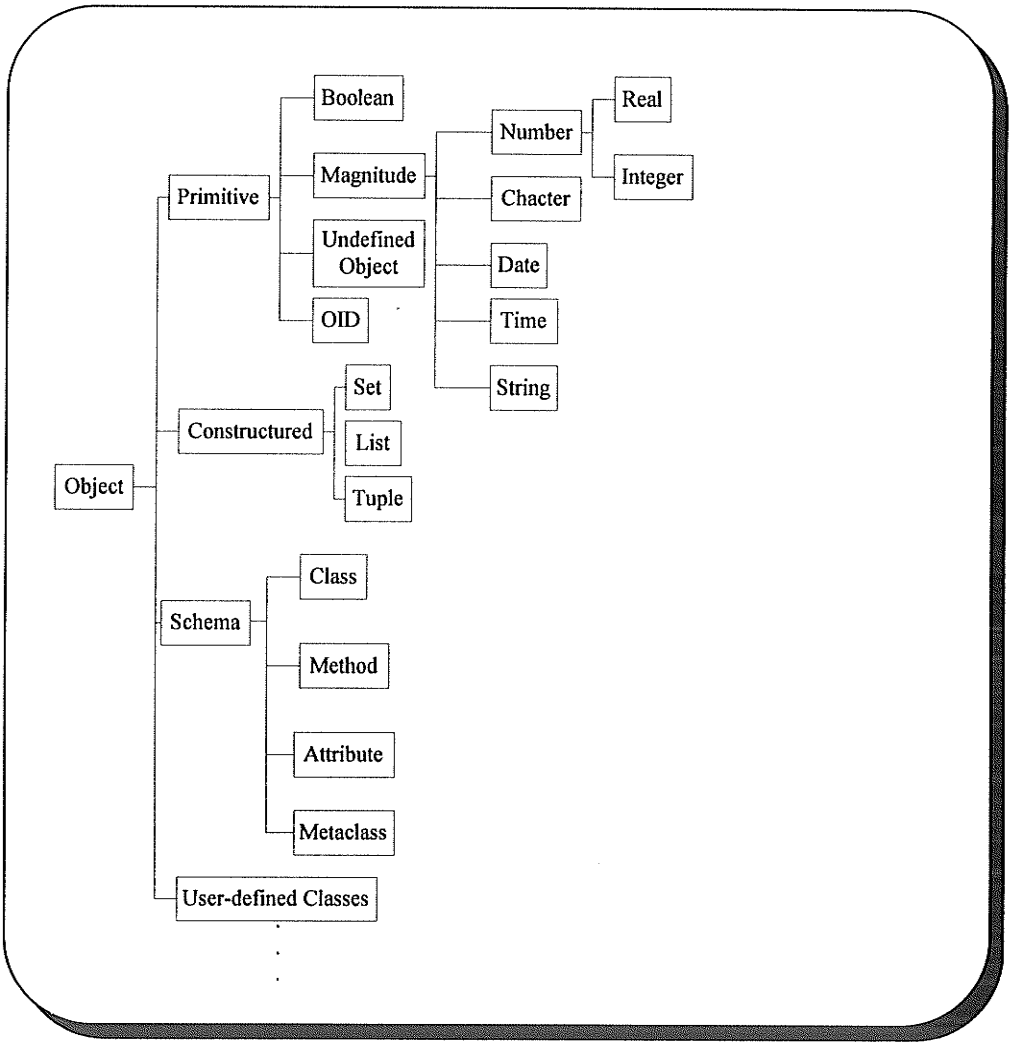

Figure 3.3 POServer Class Hierarchy

Examples of behavior which may be common to instances of all classes are: the test for determining which class an instance is associated with and comparison between

instances. When a new class is to be created, the class has to be placed somewhere in the class hierarchy. All classes in the POServer system are grouped under four abstract classes: 1. Class Primitive; 2. Class Constructed; 3. Class Schema; and 4. Class User-defined. The class hierarchy also provides a general taxonomy of the fundamental data types in the system. A class helps organize data and the class hierarchy helps organize the classes.

3.5.1 Primitive Classes

The POServer object model distinguishes between primitive objects and user-defined objects. Primitive objects are system-defined objects and are instances of all fundamental data types supported in the system, for example, Boolean, Char, Integer, Real, String, OID, Date and Time. A primitive class has associated instances, but has no attributes. These objects are self-identifying since their values indicate the kinds of data types being used. For instance, the boolean TRUE, the character 'A', the integer 99, the real 12.25, the OID '00000001:00001001', the string 'John Smith', the date '1995-01-01', and the time '12:45' are recognized by the system as primitive literal objects. Each literal object has a primitive class to which it belongs. Class Boolean defines the common behavior for two logical values: True or False. Class Magnitude is an abstract class used for comparing and measuring instances of its subclasses: Character, Date, Time, Number and String. Class Character defines the behavior for all the characters in the system (i.e. ASCII codes

0 to 255). Class Date and Time define the behvaior for comparing and computing dates and times, respectively. Class Number is an abstract class used for comparing, counting and measuring instances of its numerical subclasses: Real and Integer. Class String defines a variable-length and indexable sequence of characters. Class OID defines the behavior for all the object references, this represents any possible constructed or user-defined classes. Class UndefinedObject is used to indicate the lack of a value and has a single value, Nil. In the POServer object model, all the basic data types are encapsulated as classes. The primitive classes form the groundwork and the basic building blocks of the system.

3.5.2 Constructed Classes

The POServer object model also makes a distinction between atomic and constructed objects. Atomic objects are single-valued objects like integers, characters, reals, etc. whereas constructed objects are made up of a structure and may take on a set of values. Constructed classes provide the basic data structures used to store objects in a collection. They are also the means by which one-to-many relationships are modelled. Constructed classes are sometimes referred to as *container classes* [30] or *collection classes* [23]. To keep the discussion manageable, only the three most common constructed classes are included in the POServer model: List, Set and Tuple. Complex data types may be defined recursively using the tuple, list and set constructors. These classes can be used to

hold all elements of the same class and have slightly different behavior as suggested by their names. Class Constructed is the superclass of all the constructed classes. This class provides the behavior for accessing and manipulating a particular element or all the elements of a collection. A *set* represents a collection of objects of the same class in which duplicates are not allowed. There is no limit to the number of objects in a set, however, positioning of the objects is not guaranteed. A *list* is a collection of objects of the same class in which duplicates can occur and the positioning of the objects in a list is guaranteed. A list behaves as a flexible and insertable array. A *tuple* resembles a table in relational systems, consisting of an ordered set of [attribute name:value] pairs. This tremendous modelling power also makes it feasible to incorporate a value-based relational model in the POServer system. For instance, user-defined classes can be created to represent tables, rows, and indexes.

In the POServer system, there is a system-supported mechanism for grouping objects into collections or maintaining the class extents automatically. Each object's oid is saved in a constructed class that represents the extent of all objects in the class. This makes iteration through a class possible. The extents for classes Department and Employee may be represented as follows.

Department = $(c_1, \{d_1, d_2, d_3, d_4\})$

Employee = $(c_2, \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\})$

Note that this approach is similar to O_2 and TIGUKAT. However, these systems include constructs for both *types* and *classes* [19, 26], so as to distinguish between the collection of objects and their definition.

3.5.3 Schema Classes

The core of any relational DBMS is the system catalog tables, e.g. the SYSIBM tables in DB2. Rows are added to these tables in response to SQL data definition language commands. These tables are maintained through commands such as CREATE TABLE, ADD COLUMN, etc. [42]. The rows in these tables represent both user-defined tables and the catalog tables themselves. Research into schema evolution for an object model has been conducted in the context of ORION [30] and Gemstone [10]. The POServer system adopts a very similar approach. Classes are viewed as objects, that is, as instances of another class. Note that OO languages such as Eiffel and C++ have chosen not to consider classes as objects. In Eiffel, classes are regarded as implementing data types. A class is static and described in the program text, while an instance is dynamic and exists only during execution [39]. In contrast, by regarding classes as objects in POServer, a more flexible schema system is obtained, allowing classes to be created or modified during execution. As classes are now referred to as instances, they must be instances of some classes. These classes are called *schema classes* in the model. An application

schema may be represented in terms of several system-defined schema classes, analogous to the system catalogs in relational database systems.

For each user-defined class, attribute, and method, there is a corresponding instance in the schema classes Class, Attribute and Method, respectively. Instances of these schema classes represent all objects in the system including the primitive classes. The schema classes contain the definition and design of the application meta-model and also represent the POServer model itself so the system is self defining. Every object in the POServer system, whether it belongs to a schema or a user-defined class, is accessible in a uniform manner. The uniformity aspect allows every entity to be managed as an object. The schema classes define behavior for adding new classes to the system and determining the superclass or subclass(es) of a specific class. Also, instances of these schema classes can be used for the verification of query processing, type checking, validation of attributes and relations such as "What attributes and methods each class contains?" The notion of a Metaclass as the class of a class is also needed. Normally, methods are only performed on instances, however, the Metaclass makes it possible to define methods to operate on classes directly and to have class variables. An example of a class variable may be the number of instances that are created for this class.

3.6 Referential Integrity

Referential integrity is a term introduced by Codd and Date [14, 16]. It involves two relations and imposes the constraint that a group of attributes in one relation is the *key* of another relation. An identity-based system like POServer enable referential object sharing. It also prevents the need for *foreign-key joins* [16]. Generally speaking, referential integrity arises whenever one relation includes references to another. Referential integrity in POServer can be thought of as a pair of inverse pointers, so that if one object points to another, the second object will have an inverse pointer that points back to the first object. This kind of relationships is maintained by users. For example in the case of class Department, a set of employees is maintained via the behavior **emps**. Thus, Department 'Technical Services', d_1, consists of three employees e_1, e_3, and e_4. For maintaining proper referential integrity, a corresponding reference to the class Department must appear in the **dept** attribute of Employee. That is, **dept** of e_1 , **dept** of e_3 and **dept** of e_4 must refer to d_1, Technical Services. Similarly, if Employee e_1 is deleted, then the reference e_1 stored in d_1.**emps** should be removed by the application correspondingly.

(d_1, [**name**:'Technical Services', **number**:254, **location**:'5N'], **emps**: {e_1, e_3, e_4})

(e_1, [**name**:'John Smith',**dob**:'1961-01-01', **job_grade**:11, **dept**:d_1, **age**:34, **salary**:50000])

(e_3, [**name**:'Rob Webb', **dob**:'1963-01-01', **job_grade**:8, **dept**:d_1, **age**:32, **salary**:40000])

(e_4, [**name**:'Jim Wong', **dob**:'1964-01-01', **job_grade**:6, **dept**:d_1, **age**:31, **salary**:30000])

To enforce integrity automatically, the system must at least maintain for each object a list of identifiers of the objects that reference the object. A link from a referenced object back to the referencing object is called an *inverse relationship* in ORION [30]. This ensures the integrity of the relationship and can be useful for later access. On the other hand, Smalltalk provides powerful referential integrity through a mechanism called *garbage collection* [23]. Therefore, explicit deletion of an object is not supported and class destructors are not needed, because unreferenced objects are automatically removed from the system.

3.7 Summary

In summary, the POServer object model extends the relational model with the notions of *objects*, *methods* and *classes*. Using this object model, a rich set of semantic relationships among real world entities can be captured. These include the *instance-of relationship* between an instance and its class and the *inheritance relationship* between a superclass and its subclasses. The POServer object model can be characterized by the following main features:

1. Objects are associated with classes. A class describes common structure and behavior for all objects in the class. The metadata model information is stored in *schema classes* which can later be used for static or dynamic type-checking.

2. *Primitive classes* form the fundamental data types of the system.

3. Complex structures may be constructed by embedding references to objects within other objects.

4. *Constructed classes* can be used for storing class extents.

The POServer object model is similar in many aspects to Smalltalk with the following differences:

1. Objects can be made persistent.

2. Objects are uniquely identified by system-maintained references, not memory pointers.

3. In Smalltalk all run-time entities are objects. Even integers are instances of the Integer class. Smalltalk assumes there is a class Integer that is an abstraction or representation of integers [23]. POServer allows the primitives such as an integer to be implemented in the traditional way. As pointed out in Barker [4], we claim that it is undesirable to force computer applications to build from such a low level of type abstraction. Therefore, POServer treats integers conceptually as objects, but handle them differently at the implementation level for efficiency and performance reasons.

4. Objects can be shared across systems.

Chapter 4

The POServer Object Manager

As discussed in Chapter 2, POServer is a three-level schema architecture. The Object Manager (OM) is responsible for the in-memory data structures of the conceptual objects. This middle layer directly supports all the features of the POServer object model as outlined in Chapter 3. It is also responsible for enforcing the semantics of inheritance and for checking the consistency of a schema. However, the object model described in Chapter 3 was purely conceptual, as it did not suggest how objects should be stored on either disk or main memory. In relational systems, the data representation is relatively straight forward because primitive types can be stored in disk format and represented in memory directly. Also, there is no embedded pointer references in relational systems except for the use of indexes which are created and maintained separately by the system. The object-oriented technology has made it easy for the application programmers to access and manipulate data in the program because more complex structures are offered. However, this yields some problems for the implementation design of an object-based system, especially the representations of the complex object structures and their behaviors. Recently, considerable research efforts have been spent to design a more efficient mechanism for in-memory object management, see O_2 [19] and ORION [30]. These object-oriented systems have assumed that all objects reside in a large virtual memory. Objects are the basic run-time entities in the system and have an associated

address like a record in Pascal or a structure in C. To support applications that require significant pointer chasing for a large number of memory-resident objects, these systems have been designed to store objects in disk format and map them into memory format for manipulation by applications. The mapping includes, at the minimum, the conversion of object identifiers stored in objects to memory pointers. In this chapter, the implementation design and techniques for the POServer system are described. More information on the discussion of design and architectural issues for OO systems can be found in Goldberg [23], Kim [29] and Meyer [39].

POServer follows four basic design principles: 1. All objects must be represented in the same way in the system. This includes both system objects and user objects. Unique object representation provides a conceptual simplicity with the result that the implementation of the system is easier to understand and maintain. An added advantage is that it will be possible to treat system objects like ordinary user objects, so the system becomes extensible. 2. Dynamic schema evolution must be supported. New understanding or changing requirements may lead to reorganization of the class hierarchy. Schema changes should not affect existing applications or require recompilation. To support this flexibility, the schema of the class structure held in the system must also be changeable in the same ways. 3. Performance is another important design criteria. The efficiency of the system depends heavily on the design of the behavior implementation and the amount of copying that must take place within the

system. As a rule of thumb, copying should be deferred until the object is being used whenever possible. 4. The POServer system must also scale well as the number of objects grows. The technique for supporting object identity must allow flexibility for the movement of objects.

The fundamental requirements of POServer are persistence and efficient sharing of objects. The design of the Object Manager is abstractly defined by requiring it to provide three basic functions, namely:

1. Implementation of schema classes and primitive classes. All these classes must be represented efficiently in memory because they form the basic building blocks of the system. The object structures and their behaviors will be discussed in detail. Once the schema classes and the primitive classes are established in the system, they can be used to build other system components and data structures.

2. Implementation of inheritance. Inheritance complicates the design and implementation of the schema a lot. There are two types of schema changes to an object-based system. One is to the definition of a class. This includes changes to the attributes and methods defined for a class, such as changing the name or domain of an attribute, and adding an attribute or a method. Another type of change is to the class-hierarchy structure. This includes adding or dropping a class, and changing the superclass/subclass relationship. An efficient mechanism for keeping track of the inherited attributes and methods will be described.

3. Behavior implementation. An efficient mechanism for selecting and invoking methods will be described and an algorithm for method resolution will be introduced.

4.1 Implementation of Schema and Primitive Classes

The state of an object consists of values for the attributes of the object and the values are themselves objects, possibly with their own states. Thus, a natural representation for the state of an object is a set of identifiers of the objects. Object representation in memory can be thought of as a flexible structure with insertable slots. Each slot contains a pointer to an attribute of the object. Since an attribute of an object can be another object or a set of values, it can also be a pointer to a set of associated objects. From the structural viewpoint, POServer's in-memory object representation can be implemented as a variable linear array of pointers. Each element of which is an 8-byte pointer and may potentially reference any other object in the system. This vector of pointers is called the *Object Descriptor* (OD) and is the run time representation of an object. This data structure possesses the ability to grow or shrink as required. The dynamic and insertable nature of the object descriptor fits our design requirements very well and provides the necessary functionalities. The object structures in the POServer system have been refined from Smalltalk's *Large Object-Oriented Memory* (LOOM) system [15].

An extremely powerful feature of object modelling is *polymorphism*. To achieve this, every stored object must know to what class it belongs. Therefore, the first slot of the OD always contains a pointer to that object's class. The class of an object contains a template (a list of attributes) that represents the internal organization of the object. This template is used whenever the OM needs to enforce semantics or to perform type-checking. This attribute is the **class_of** of the object by which each object is linked to its associated class.

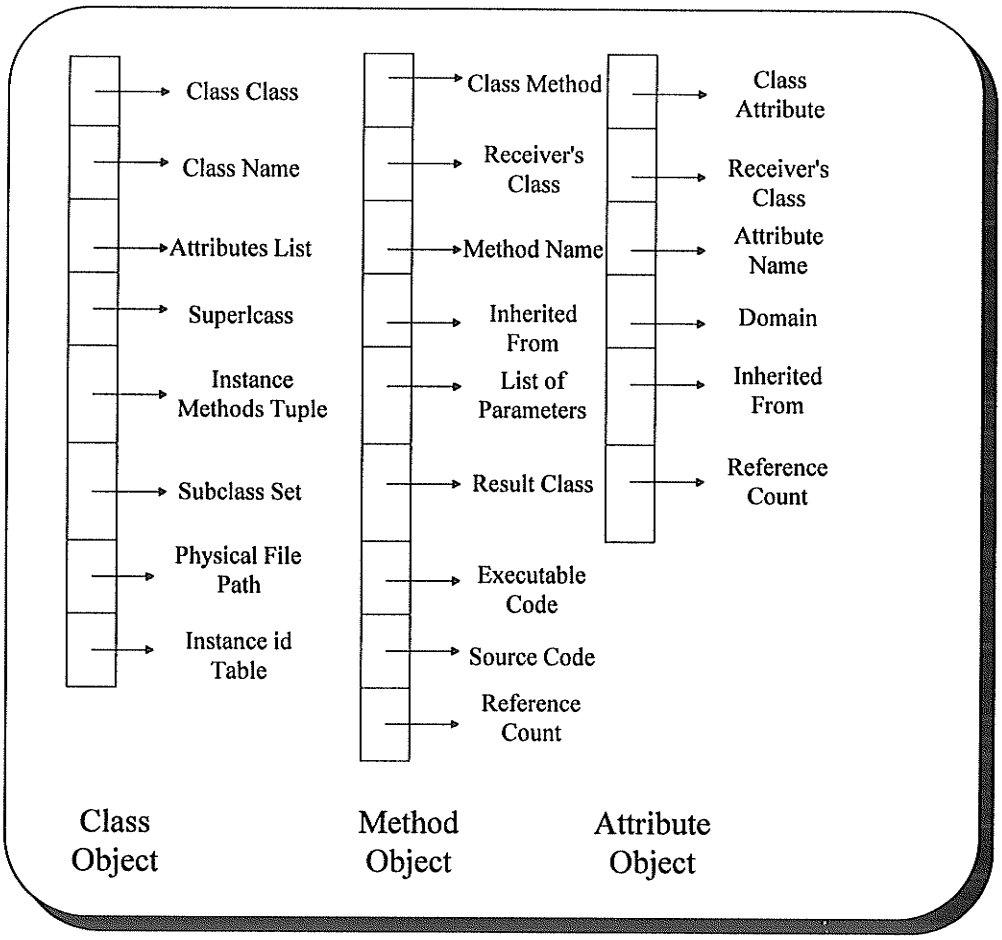

Figure 4.1 Object Structures for Schema Classes

For example, the first slot of a method object is an attribute called **class_method** to indicate that it is associated with the class Method. Similarly, the first slot of a class object is an attribute called **class_class** to indicate that it is associated with the class Metaclass. The schema classes and the primitive classes are set up the same way but with appropriate contents in the slots so as to reflect a semantic correspondence to the class system and to support the associated behaviors. Figure 4.1 illustrates the object structures for schema classes.

4.1.1 Class Class

In POServer, three different types of objects are used to capture schema information. They are instances of the system defined schema classes: Class, Method and Attribute, respectively. Every class in the system is represented by a class-defining object that describes the structure and behavior of the class, as well as the position of the class in the inheritance hierarchy. The object structure for the class Class contains **class_name, attributes, superclass, subclasses, instance_methods, physical_file** and **iid_table**. The attribute **class_name** contains the name of the class; **attributes** is a list of all attributes defined for or inherited into the class; **superclass** and **subclasses** are the direct superclass and the set of subclasses of the class, respectively. **instance_methods** consists of a table of methods defined for or inherited into the class. The method table can be search

directly by the receiver's class and the method name. **physical_file** is the file path for storing objects of the class and **iid_table** is the instance id table. These two attributes will be discussed in great detail in Chapter 5. It is important to note that the attributes **superclass, subclasses,** and **instance_methods** do not form part of the schema for a relational system. A class object is like an accessible class descriptor. Like user-defined classes, schema classes have their associated behaviors. These system-defined methods for schema classes represent the external interface and hide the internal structures from users. Behaviors for schema classes are mainly used to retrieve and update metadata. The most basic methods for the schema class Class can be defined with the following signatures:

aClass.**class_name** →aString /* returns the name of a class */

aClass.**instance_methods** →aSet /* returns a set of methods for a class */

aClass.**attributes** →aList /* retrieves a list of attributes for a class */

aClass.**subclasses** →aSet /* returns a set of subclasses for a class */

aClass.**superclass** →aClass /* returns the superclass of a class */

aClass.**is_subclass**(aClass) →aBoolean /* a class is a subclass of another class? */

aClass.**is_superclass**(aClass) →aBoolean /* a class is a superclass of another class? */

aClass.**create** →anObject /* returns an empty object descriptor for a class*/

aClass.**method_selectors** →aSet /* returns a set of method names for a class */

aClass.**lookup_behavior**(aString) →anObject /* returns a behavior for the class */

anObject.**class** →aClass /* returns the associated class for an object */

4.1.2 Class Method

The class Method has an instance for every method defined for or inherited into each class. It contains the representation of the executable code and linkage information. The object structure for the class Method includes attributes such as **method_name**, **executable_code**, **source_code**, **parameters**, **result_class**, **inherited_from** and **reference_count**. The attribute **method_name** contains the name of the method; **executable_code** points to the executable module in memory while the **source_code** contains the location of the source file including the directory path information; **inherited_from** refers to another instance of the class Method and it indicates from which the method is inherited; **parameters** and **result_class** specify the signature of the method. The following signatures make up the behaviorial part of the schema class Method:

aMethod.**method_name** →aString /* returns the name of a method */

aMethod.**executable** →anObject /* returns a reference to a piece of executable code */

aMethod.**execute**(aList)→anObject /*executes the piece of code using the list of parms*/

aMethod.**parameters** →aList /* returns a list of attributes for the parameters */

aMethod.**result_class** →aClass /* returns the result class */

aMethod.**ref_count** →anInteger /* returns the reference count */

aMethod.**in_cache** →aBoolean /* tests if the method has been loaded in memory */

aMethod.**lib_path** →aString /* returns the library path for the executable code */

4.1.3 Class Attribute

The class Attribute has an instance for every attribute defined for or inherited into each class. It contains the representation of the state of an object. The object structure for the class Attribute is defined as follows: **attribute_name, receiver_class, inherited_from, reference_count** and **domain**. The attribute **attribute_name** contains the name of the attribute; **receiver_class** is the class that owns this attribute; **domain** specifies the class to which the value of the attribute is bound; **inherited_from** refers to an instance of the class Attribute and it indicates the attribute of the superclass from which the attribute is inherited. The attributes **inherited_from** and **reference_count** in the schema classes Method and Attribute are used for the implementation of inheritance and will be discussed further in the next section. The following signatures define the behaviors for the schema class Attribute:

anAttribute.**attribute_name** → aString /* returns the name of an attribute */
anAttribute.**domain** → aClass /* returns the domain of an attribute */
anAttribute.**receiver** → aClass /* returns the receiver's class */
anAttribute.**is_attribute**(aClass) →aBoolean/* an attribute is in a class's structure? */
anAttribute.**ref_count** →anInteger /* returns the reference count */

4.1.4 Primitive Classes

POServer allows primitive classes to be viewed as objects but are implemented in the traditional way. It is assumed that instances of primitive classes are self-identifying and

serve as state, identity and reference simultaneously. Therefore, instances of a primitive class have no identifiers associated with them. This saves the creation and maintenance of object identifiers for primitive classes. Also, a primitive class has no attributes associated with it and a primitive object is not further decomposed. As shown in Figure 4.2, a primitive object borrows the language primitive types and points to the underlying data structure directly. That is, if the domain of an attribute is a primitive class such as integers and reals, the values of the attribute are directly represented. This save memory space and execution overhead.

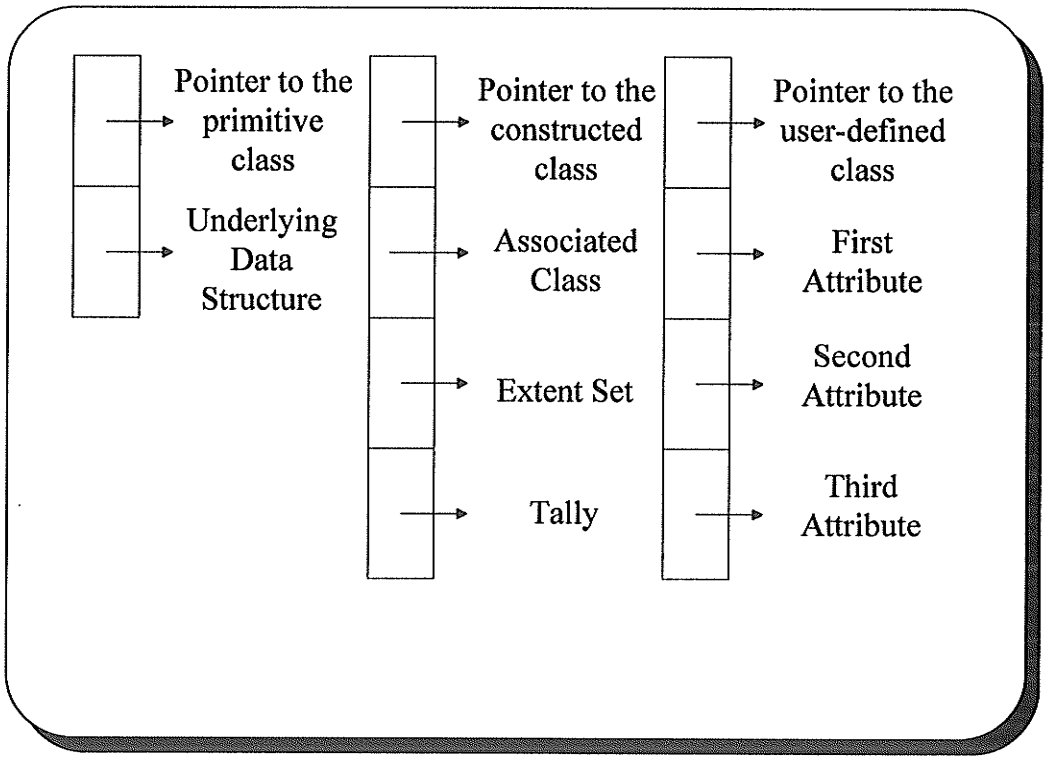

Figure 4.2 Object Structures for Primitive Classes and Constructed Classes

However, from the user's perspective, it is expected that the entire domain of these primitive classes exists and every primitive object ever needed is assumed to be in existence and always available when required. In order to allow the primitives to be treated as ordinary classes and manipulated like other objects, the normal primitive operators such as the logical operators, the comparison operators and the arithmetic operators are defined as behaviors in the corresponding primitive classes. For example, the following signatures define the behaviors for the primitive class Boolean:

aBoolean.**not** →aBoolean /* returns the opposite value of a boolean */
aBoolean.**or**(aBoolean) →aBoolean /* performs the OR operation on two booleans */
aBoolean.**and**(aBoolean) →aBoolean/* performs the AND operation on two booleans */

The standard comparison operators are provided for the abstract class Magnitude. The signatures are defined as follows.

aMagnitude.**greater_than**(aMagnitude) →aBoolean/* testing for greater than */
aMagnitude.**less_than**(aMagnitude) → aBoolean /* testing for less than */
aMagnitude.**max**(aMagnitude)→aMagnitude/*selects the max between two magnitudes*/
aMagnitude.**min**(aMagnitude)→aMagnitude /*selects the min between two magnitudes*/

Since all numbers are essentially the same type, therefore the common behaviors are defined on the abstract class Number, not on the actual class like Real or Integer.

aNumber.**plus**(aNumber) →aNumber /* returns the sum of two numbers */

aNumber.**minus**(aNumber) →aNumber /* subtracts the second number from the first */

aNumber.**times**(aNumber) →aNumber /* multiplies two numbers together */

aNumber.**divide**(aNumber) →aNumber /* divides the first number by the second */

4.1.5 Constructed Classes

The constructed classes are mainly used for the manipulation of collections. POServer provides direct support for collections of objects through classes Set, List and Tuple. One-to-many and many-to-many relationships can be constructed. In designing the collection facility, there are several reasonable implementation choices. For example, it is reasonable to embed a fixed-length array of pointers into the object structure if the cardinality of the relationship is known to be fairly small. However, a collection can be arbitrarily large if it is used to store all objects of some class e.g. all employees. POServer stresses uniformity of access to all system objects and behaviors. Therefore, the same Object Descriptor approach is used for the implementation of collections as shown in Figure 4.2. Thus, the object structure for the class Constructed contains **associated_class, extent_set** and **tally**. The attribute **extent_set** is implemented as a pointer to a set of references; **associated_class** refers to the class of the elements in the collection. It is possible for one object to hold a collection of references to other objects and for that collection to be made persistent. A **tally** is maintained in a collection to make the behavior aCollection.**count** more efficient.

The following signatures define the basic behaviors for the constructed/collection classes:

aCollection.**includes**(anObject) →aBoolean /* tests if the object is in the collection */
aCollection.**count** →anInteger /* returns the number of elements */
aCollection.**first** →anObject /* returns the first element in the collection */
aCollection.**last** →anObject /* returns the last element in the collection */
aCollection.**is_empty** →aBoolean /* tests whether it is a empty collection */

All the collection classes need a basic mechanism for inserting an element, another for accessing an element, yet another for removing an element and so on. A set of standard behaviors are used for the manipulation of collections. These include **update_element**, **add_element**, **get_element** and **remove_element**. These behaviors are defined in the abstract class Constructed and assume that collections will have a specific element order. Since sets do not preserve element order, therefore different result may return when repeated. It is interesting to note that collection objects can be indexed using array notation as follows.

aCollection.**get_element**(anInteger) →anObject /* returns the specified element */
aCollection.**update_element**(anInteger, anObject) →aCollection /* returns a new
collection constructed from the original collection by modifying a specified element */
aCollection.**remove_element**(anInteger) →aCollection /*returns a new
collection constructed from the original collection by removing a specified element */
aCollection.**add_element**(anInteger, anObject) →aCollection /* returns a new

collection constructed from the original collection by adding a specified element */

In addition, behaviors associated to a specific class can be added accordingly. For example, the signatures for **union** and **append** are defined for the class Set and List, respectively.

aSet.**union**(aSet) → aSet /* returns a set constructed as the union of two sets */

aList.**append**(anObject) → aList /* appends an object to a list */

aTuple.**lookup**(aKey) →anElement /* returns an element with the associated key */

4.1.6 Class Object

The important comparisons specified in the root class Object are equivalence and equality testing.

anObject.**identical**(anObject) →aBoolean /* tests whether the receiver and the argument are the same object */

anObject.**equal**(anObject) →aBoolean /* tests whether two objects represent the same identical semantics */

The implementation of the behavior **identical** is quite straightforward. It is a simply comparison between the two OID's. It returns True if they have the same OID, otherwise returns False. For the implementation of the behavior **equal**, the decision as to what it means to "represent the same identical semantics" is made by the receiver of the message. Typically each class must re-implement the method **equal** in order to specify which of its attributes should enter into the test of equality. In POServer, the default implementation

of the method **equal** is the same as that of **identical** at the root class Object and each of the primitive classes redefine this method **equal** to do a comparison of values which they represent. For equality of collection objects, two lists are equal if they have the same number of components and all corresponding component pairs are equal; two sets are equal if they have the same number of components and for each component in one set there is an equal component in the other set; equality of two numbers is determined by testing whether the two numbers represent the same value.

4.1.7 Object Access Table (OAT)

Other important structures used in POServer are *Object Access Table* (OAT) and object caches. The object caches will be discussed in Chapter 5. OAT is maintained by the Object Manager and is implemented as a tuple. Each entry in the Object Access Table consists of triplets:

[object_identifier, object_descriptor_location, object_descriptor_state]

as shown in Figure 4.3. The structure of the Object Descriptor (OD) has been discussed in detail in the previous sections. The **object_descriptor_location** is simply a pointer to the corresponding cached OD which points to the physical location of the object in memory. The **object_descriptor_state** is used to indicate whether the object has been created in memory or is being loaded from disk.

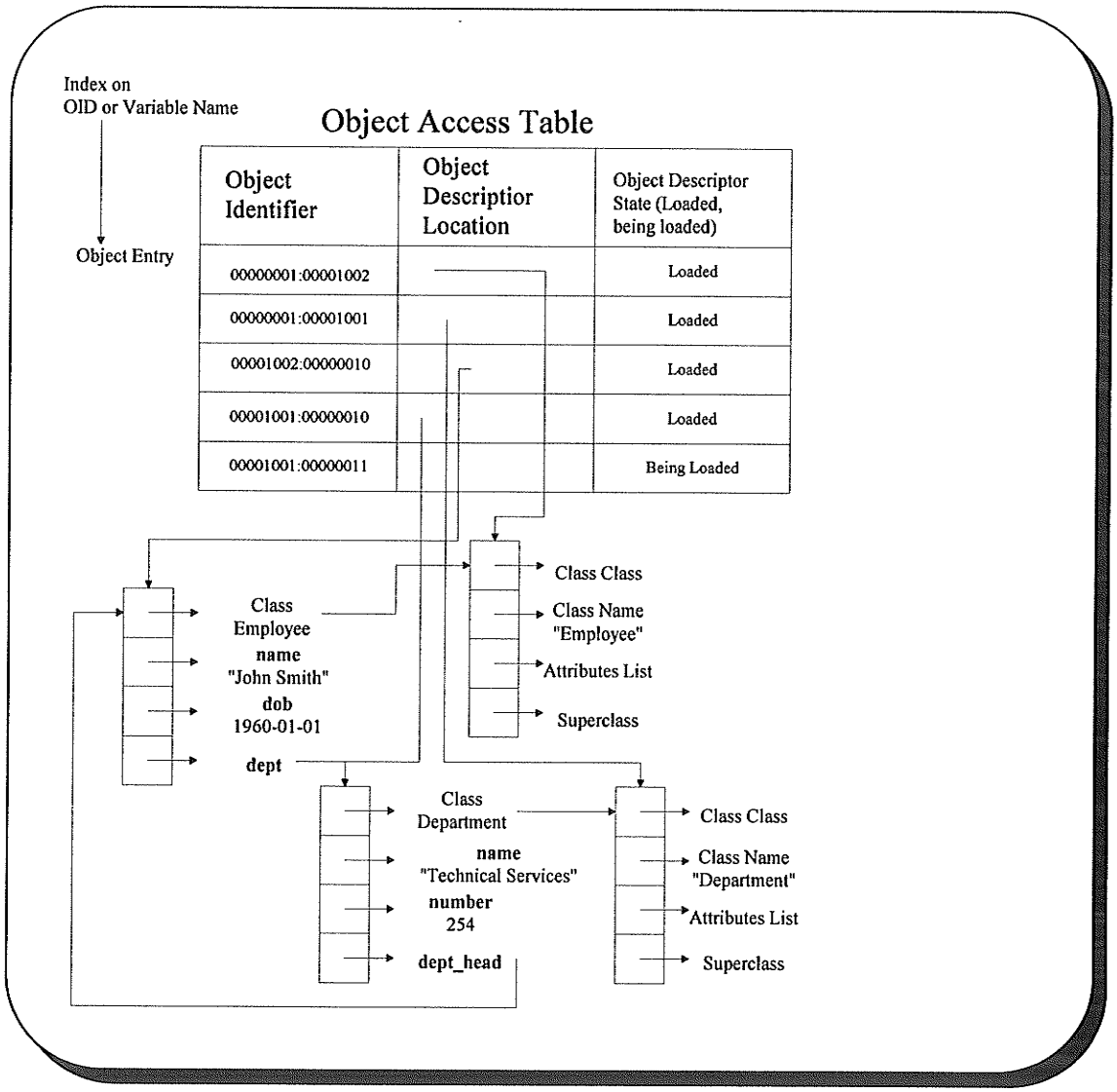

Figure 4.3 Object Access Table and object instances in memory

Objects are referenced through an identifier or an object name. POServer enables explicit names to be given to any object or collection. As illustrated before (in Figure 2.2), variable names :d254 and :john are associated with Department 254 and Employee 'John Smith', respectively. From a name, an application can directly retrieve the named object

and then operate on it or navigate to other objects following the relationship links. All the named variables of the receiver are available via their names. Two indexes are maintained by the system to reference an object. One index contains the OID for normal access to object data and another for translating object variable names to OIDs. Next we illustrate some of these concepts through an example shown in Figure 4.3. For example, when Employee :john is referenced, the variable name index is searched first. The corresponding object entry for OID '00001002:00000010' is then returned. By following the object descriptor for :john, his associated attributes can be retrieved. An object reference in memory is effectively a system-allocated pointer that can be followed directly to find that object. Since :john holds a reference to the department he works in, therefore his department can be located by following the pointer in **dept** and :john.**dept.name** returns 'Technical Services'. Retrieval of an object which is not in the Object Access Table will result in the Object Manager sending a request to the Storage Manager to retrieve the object. The Storage Manager provides the low-level storage management by interfacing with the operating system.

4.2 Implementation Of Inheritance

Inheritance means that we can develop a new class merely by stating how it differs from another, already existing class. The new class then inherits the existing class. Since a class contains object structure and associated behaviors, the reuse of classes is a much more powerful feature than the reuse of procedures as in traditional programming

languages. The main advantage with this approach is that existing classes can be reused to a great extent. Smalltalk is delivered with an extensive class library and the *inheritance algorithm* [23] works as follows. When a message is sent, the methods in the receiver's class are searched for one with a matching selector. If none is found, the methods in that class's superclass are searched next. The search continues up the superclass chain until a matching method is found. This approach obviously causes performance problem and is highly inefficient because several accesses are required to validate an object and to execute a method. This performance issue can be solved by the notion of a technique called *class-hierarchy flattening* as discussed in Kim [31]. That is, both attributes and methods are defined for a class and inherited by all subclasses. Features inherited directly or indirectly are put in a flattened form at the same level. The implementation strategy is fairly straightforward. It involves taking the union of the interface sets of all the classes declared as immediate superclasses of the new class being created. The algorithm iterates through the relevant interfaces and selects all the behaviors with unique signatures as candidates for insertion into the new class's inherited set. Class-hierarchy flattening is the best for polymorphic retrieval because only a direct access is required to get information about one object. However, the size of classes will increase dramatically since the inherited attributes must be duplicated. Additionally, if changes occur in the inherited attributes, these changes will affect the attributes of all subclasses.

The recommended approach to implementing inheritance in the POServer system is an intermediate between Smalltalk's algorithm and the class-hierarchy flattening technique. Assume that :jim is a Systems Engineer and we are trying to invoke :jim.**salary**. First we retrieve a set of method selectors for the Class Systems Engineer by invoking the behavior **method_selectors**. Since method **salary** is not found, therefore SystemsEngineer.**superclass** is executed to proceed to the class' parent, the class Employee is then returned. If the method does not exist in the class referred to by **superclass**, then the search is continued upwards in the class hierarchy until, if necessary, the Object class is accessed. So far the mechanism is similar to Smalltalk's inheritance algorithm. However, the search up the class hierarchy only needs to be performed once if the behavior is defined in one of the superclasses. At this point, we will borrow the class-hierarchy flattening technique. A new instance consists of the receiver's class (Systems Engineer), the method name (**salary**) and the address of the executable code is added to the class Method. Also, the inherited method, **salary**, is then added to the set of methods for Systems Engineer. Subsequent invocation of the method, :jim.**salary**, can directly retrieve and execute the inherited method. This approach improves performance significantly and includes less redundancy since not all attributes and behaviors are copied unnecessarily. The inheritance algorithm for the POServer system is shown in Figure 4.4.

Algorithm 4.1 The Inheritance Algorithm

```
Begin
  Input  ReceiverCls      Receiver's class
         MethodSel        Method selector
  Var    CurrCls          Current class
         RMthSet          A set of methods for receiver's class
         CurrMthSet       Current method set
         TargetMth        Target method
         InheritedMth     Inherited method
         Found            Search indicator

  Found = False
  CurrClass = ReceiverCls.superclass
  Do While CurrCls ~= Class Object and not Found
        CurrMthSet = CurrCls.instance_methods
        For each CurrMthSet_i ∈ CurrMthSet Do
              If CurrMthSet_i.method_name = MethodSel
                    Found = True
                    TargetMth = CurrMthSet_i
                    TargetMth.ref_Count = TargetMth.ref_count + 1
                    InheritedMth = Method.create(TargetMth)
                    RMthSet = ReceiverCls.instances_methods
                    RMthSet.add_element(InheritedMth)
              Endif
        EndFor
        CurrClass = CurrClass.superclass
  EndDo
  If not Found
        InheritedMth = -1   Method undefined
  EndIf
  Return InheritedMth
EndBegin
```

Figure 4.4 The Inheritance Algorithm

In the POServer system, some of the schema changes may affect the values of its subclasses. For example, if Employee.**salary** was dropped, the inherited method SystemsEngineer.**salary** would be invalidated. Algorithm 4.1 uses the **reference_count** defined in class Method to implement the *deferred-update technique* [30]. The deferred-update technique allows the obsolete information to be deleted at some later time so that the system can guarantee that a schema change completes quickly. The **reference_count** of Employee.**salary** is increased by 1 when the method **salary** is inherited by Systems Engineer. In contrast, the **reference_count** of Employee.**salary** is decreased by 1 when the method **salary** is overloaded or removed from Systems Engineer. The method can be cleaned out when its **reference_count** is dropped back to 0. It is interesting to note that Gemstone adopted the *immediate update approach* [10]. As the name suggests, the algorithm immediately updates all affected schemas. The drawback of this approach is that it makes a schema change potentially very time consuming because the inherited chains must be checked and maintained for each schema operation.

Meanwhile, the degree of inheritance offered by any particular implementation of an object-based system may vary from nil to full. POServer provides full inheritance since any behavior in a class can be inherited by its subclasses, simply by adding the target behavior to the set of behaviors in the receiver's Class as shown in Algorithm 4.1. In contrast to this approach, C++ imposes further constraints on inheritance. In C++

terminology, only a *virtual function* [21] as declared by the programmer may potentially be inherited or replaced in each of the derived classes. This limitation may have some efficiency benefits for the compiler but it reduces the reusability of the C++ object model.

4.3 Behavior Implementation

Polymorphism is one of the key features in an object-based system. It means that the sender of a message does not need to know the class of the receiving instance. The sender simply provides a request for a specified operation, while the receiver knows how to perform this operation. The polymorphic characteristic sometimes makes it impossible to determine at compile time which class an instance belongs to. The ambiguity about which method should be invoked can only be resolved at runtime when the message is actually sent. Thus, *dynamic binding* is a way of implementing the polymorphism characteristic. Dynamic binding is flexible, but reduces performance. If the class of each object is known at compile time, then the correct method can be determined and called directly. *Static binding* is more secure and efficient due to the method look-up algorithm being performed only once during the compilation. For the POServer system, each object has a very clear type, namely its class, so objects can be associated with any class in the system and dynamic binding will occur in all situations. Therefore, in principle, the method look-up must be carried out during execution.

```
typedef struct
{
        boolean (* equal) ();   /* basic functions provided at the root level */
        oid       (* oid) ();
        boolean (* identical) ();
} objectclass;

typedef struct
{
        boolean (* equal) ();           /* function inherited from the superclass */
        boolean (* includes) ();        /* specific functions for class Set */
        integer (* count) ();
} setclass;

objectclass * object_init(self)         /* initialization for class Object */
objectclass * self;
{
        self->equal = object_equal;     /* assign function ptrs to corresponding
                                           methods */
        self->oid    = object_oid;
        self->identical = object_identical ;
        return(self);
}

setclass * set_init(self)                       /* initialization for class Set */
setclass * self;
{
        self->equal = set_equal ;       /* function equal is overloaded */
        self->includes = set_includes;
        self->count = set_count ;
        return(self);
}

/* set objects can invoke the function set_equal by the follow syntax */

(*myset -> equal) (myset, yourset) ;
```

Figure 4.5 The Use of Function Pointers in C for Dynamic Binding

The C language supports an elegant and efficient technique for performing dynamic binding using a predefined structure containing function pointers. For example, we can define the class descriptor as a C *struct* containing all the methods defined in the class or inherited from a superclass. Basically, the class descriptor struct defines the signatures of the methods visible to a class. Another initialization function, e.g. set_init and object_init, can be set up to define and initialize the class descriptor with the proper method implementation. Later, a function call can be made that is indirectly referenced through a pointer variable by applying the function call operator to the pointer as shown in Figure 4.5.

Behavior application involves the retrieval and execution of an appropriate piece of binary code that is dependent on the receiver's class and the selector for that behavior. A cursory look at some system implementations of dynamic methods resolution and execution will be enlightening. The C++ system implementation maintains a *virtual table* [21] for every class that has at least one virtual function. Each object of a class that has virtual functions needs to maintain a pointer to that class' virtual table. A call to a virtual function is resolved according to the underlying type of object for which it is called. The proper offset into the table is computed at runtime during function dispatch. A direct jump can then be performed to get to the address of the appropriate code to be executed. In contrast, the Smalltalk-80 system maintains a structure called *method dictionary* [23] in the class description for each class. The keys in this dictionary are

message selectors and the values are the compiled form of methods. The protocol supports compiling methods, accessing the compiled and noncompiled versions of the method, and adding the association between a selector and a compiled method. In summary, sophisticated caching strategies or special dispatch tables are used to minimize the overhead and to make the method dispatch more efficient.

As for the POServer system, the methods resolution algorithm becomes pretty straightforward once inheritance Algorithm 4.1 is in place. The receiving instance is responsible for searching for and finding the appropriate method to be executed. The receiver's class can be determined fairy easily from the first pointer of the object instance or by calling ReceiverObject.**class**. The method resolution for POServer which interprets a message sent to an object therefore operates according to the algorithm shown in Figure 4.6. Algorithm 4.2 can be summarized as follows. When an object is to perform a method, its associated class selects the required method and performs it using the object's attributes and the parameters. The method is selected by finding a method that has the same name as the message. This can be achieved by invoking ReceiverCls.**lookup_behavior**(MethodSel) which returns the target method stored in class Method. If the method is not found in the receiver's class, then the superclass of the receiver's class must be examined to see if there is a signature for the passed message. This is done by calling inheritance Algorithm 4.1 defined earlier. The inheritance algorithm continues the search up the superclass chain until the behavior is found. If the

signature is unknown, -1 will be returned from Algorithm 4.1 and the error message 'unknown method' is displayed. Otherwise, it executes the target method using the parameter list by calling TargetMth.**execute**(ParmList). Finally, any values from the execution of the method are returned.

Algorithm 4.2 The Method Resolution Algorithm

```
Begin
  Input  ReceiverCls      Receiver's class
         MethodSel        Method selector
         ParmList         Parameter list
  Var    TargetMth        Target method
         Found            Search indicator
         Result           Function's result

  Found = False
  TargetMth = ReceiverCls.lookup_behavior(MethodSel)
  If TargetMth = -1   Method not found in receiver's class
        TargetMth = Call Inheritance Algorithm 4.1 with
                                  ReceiverCls, MethodSel
        If TargetMth = -1   Method undefined
              Error Handling - display message 'unknown method'
        Else
             Result=TargetMth.execute(ParmList)-refer to Algorithm 4.3
        EndIf
  Else
        Result = TargetMth.execute(ParmList)
  EndIf
  Return Result
EndBegin
```

Figure 4.6 The Method Resolution Algorithm

Since it is not possible to know where the required load modules are and what libraries the program will need at link time, C's shared library facility is used for the implementation of method execution in POServer as shown in Figure 4.7. The object TargetMth contains the library path (TargetMth.**lib_path**) of the executable code and the method selector (TargetMth.**method_name**) to be executed that are required for the program execution. Loading a library at run time is known as *explicit loading* [12] in UNIX. For performance sake, all primitive classes and the associated methods can be loaded into a *method cache* during system initialization. When a message is sent to an object, the system first sees if the object is already in the method cache. If the message has a corresponding function in the cache, it is invoked directly to handle the message. If it is not present, the library path and the method selector of the target method are used to locate the appropriate module in the shared library. A UNIX system service call, Load(&Handler, LibPath, MethodSel, &ExecutableCode), returns a pointer to the executable code that implements the message from the shared library. Program execution can be done by calling the pointer to the &ExecutableCode obtained in the Load function with the parameter list, for example, Execute(&Handler, FunctionPtr, ParmList, &Result). The **in_cache** indicator of the target method is then marked as true and the method is made available for all instances of the class. Methods subsequently loaded into the cache can be maintained by the well known caching technique known as the *least recently used* (LRU) algorithm that attempts to replace the methods that have not been referenced for an extended period, as discussed in Effelsberg and Harder [20].

Algorithm 4.3 The Method Execution Algorithm

```
Begin
  Input  TargetMth          Target method
         ParmList           Parameter list
  Var    Handler            Returned handler from operating system
services
         FunctionPtr        A pointer to the executable code
         MethodSel          Method selector for target method
         LibPath            Library path for executable code
         ExecutableCode     Pointer to executable code
         Result             Function's result

  If TargetMth.in_cache
         FunctionPtr = TargetMth.executable
  Else
         MethodSel = TargetMth.method_name
         LibPath = TargetMth.lib_path
         Load(&Handler, LibPath, MethodSel, &ExecutableCode)
         If Handler = ok
                FunctionPtr = ExecutableCode
                TargetMth.in_cache = True
         Else
                Error Handling
         EndIf
  EndIf
  Execute(&Handler, FunctionPtr, ParmList, &Result)
  Return Result
EndBegin
```

Figure 4.7 The Method Execution Algorithm

Chapter 5

The POSever Storage Manager

The primary objective of the POSever Storage Manager (SM) is to provide persistence. This is the ability to store and allow objects to survive beyond the duration of the process that created them and after the termination of the program that manipulates it. The need for persistence arises from the volatile nature and limited capacity of primary memory. Persistence often means that objects are copied from a fast and volatile primary memory to a slow and persistent secondary memory, therefore the Storage Manager must also provide efficient ways to access and manipulate objects in secondary memory. The end users should only see the logical view of the object model and should not take part in the decisions about how the physical storage is done, therefore the operations taken by the Storage Manager should be made totally transparent to end users. The Object Manager (OM) is the only user of the Storage Manager. They both communicate and interact through a set of procedural interface called the *Object Manager Interface* (OMI). The SM provides access to physical storage for objects. It manages the allocation and deallocation of pages on disk, moves pages to and from disk, finds and places objects in object buffers. The SM in fact consists of two components. The SM server deals with objects in the disk format and is responsible for maintaining physical storage for objects. The SM client resides with the OM and interacts with it to perform object transformations and object buffer management. The SM is really a page server since it deals only with

pages and does not understand the semantics of objects. Entire pages are transferred between the SM client and the SM server so the overhead on the communication link is minimized. If a proper clustering mechanism is in place, a significant fraction of the objects on each page will eventually end up being referenced by the OM. The SM client and SM server can be implemented by Mach's remote procedure call (RPC) interface. The OMI calls made by the Object Manager to the SM client are translated into RPC calls to the SM server.

5.1 The Persistence Model For POServer

In general, a persistence model defines and specifies three aspects of an object-based system: 1) What classes can be made persistent; 2) When objects become persistent; and 3) How objects acquire persistence properties. We shall look at each aspect in turn. The goal is to minimize changes or extensions to the object model to achieve persistence and to make the operation as seamless as possible. Persistent objects should be referenced and manipulated by an application in the same way as transient objects. This is in contrast to the relational systems, where only certain data types can be persistent. For example, to read some columns of a table from DB2, if the query returns a set of records then a *cursor mechanism* [42] must be defined to co-ordinate between the programming language and the DBMS as the programmer iterates through the result set, mapping one record at a time into the buffer. This is because relational systems support only a single

data structure -- records. Many database researchers now agree that persistence should be a characteristic of objects entirely orthogonal to their type. Atkinson called the notion *persistence orthogonal to type* [2].

Two dominant approaches to persistence in present OO systems are considered here. The first approach is what has been termed reachability based persistence. For example, O_2 has defined a *reachability model* of persistence [19]. Objects in O_2 are created as transient, then made persistent when referenced by a persistent object. Access to persistent objects requires no explicit read or write calls to storage manager, but needs to mark and retrieve *persistent roots*. An object needing persistence would then inherit from the *persistent class*. When an object or value becomes persistent, so do all of its components, and vice versa. O_2 supports the automatic storage of objects with all their dependants. It is interesting to note that persistence in O_2 is implemented by associating a *reference count* with each object. Therefore, no explicit deletes are required because objects are garbaged automatically when no longer referenced. Similarly, the notion of persistence is also built in Eiffel through a *database root* [39]. To become persistent, an object or a value must be attached directly or transitively to a *persistent root*. Every object *reachable* from the database root is persistent. A single root class to provide basic persistence capabilities guarantees uniform I/O semantics. From the data manipulation point of view, persistence is transparent. The second approach to persistence is based on *membership* in a *persistable collection class*. Systems like ObjectStore take this

approach which suggests that the decision about persistence be made prior to object creation. It restricts the persistence of an object by requiring it to be allocated within some *persistent container* [34] during object creation. This is in contrast to type extents where the object DBMS will automatically maintain object collection. In this approach, it is the programmer's responsibility not to leave dangling pointers to transient objects in persistent space which means referential integrity is difficult to maintain in such systems.

Atkinson's principle has a great influence on the POServer system. POServer can store instances of any class defined in the object model, not just those that can be force-fit into records. POServer objects may be transient or persistent and may be converted from persistent to transient or vice versa at will. The object descriptors (ODs) introduced in Chapter 4 are a uniform means to reference objects and values. An OD is applied for both persistent or temporary objects. Persistent objects remain in the system beyond the life of a program execution. Transient objects are newly created objects stored only in the client's memory space and disappear when the application terminates. Our implementation approach resembles database-style semantics: explicit create and delete calls. Therefore, two behaviors are implemented for the class Object, namely, **persistent** and **transient**. As an example, :employeex.**persistent** makes the receiver object employeex persistent, while :employeex.**transient** converts the receiver object employeex from persistent to transient. These two behaviors are equivalent to CREATE OBJECT and DELETE OBJECT in some object systems [22]. Also, the Storage

Manager is based on Unix file system. Objects belonging to the same class are clustered in one contiguous segment and each class is physically mapped to the underlying OS file system. This is similar to relational systems where tuples of a relation are stored in the same segment of disk pages. *Clustering* refers to storing related objects close together on secondary storage. It is a highly useful technique that can be used to minimize the I/O cost of retrieving a set of related objects. Since each class is stored in a separate file, the SM provides object I/O to disk file translation in the same way that 3GL (e.g. COBOL) provides record I/O. The :employeex.**persistent** function is thus equivalent to adding a new employee record to the Employee file while the :employeex.**transient** function deletes the employee record from the file. The default clustering mechanism helps the system to maintain the extent of persistent objects in a class. This allows sequential scanning of all objects in a class to be carried out efficiently.

However, clustering is much more difficult for OO systems than for conventional systems because there are more ways in which a set of related objects may be accessed together. Persistent objects can be accessed explicitly by global variable names or transparently when references or pointers are followed. It may be beneficial to the Object Manager to be able to access an object and its embedded objects rapidly if the entire complex object is needed. Therefore, it may be useful to cluster an object and the objects it references, even if these objects belong to different classes. Ideally, class designers should be able to control the clustering of objects within a system. This can be achieved by providing the

information for object clustering when defining the class hierarchy through some ODL syntax. The information that certain objects must be clustered with other objects in the same storage extent is then stored by the OM as metaclass information. The SM will then make use of this information to perform proper and efficient object clustering.

5.2 Logical And Physical Object Identifiers

An important design decision for POServer is the implementation of object identity. Identity is that property of an object which distinguishes each object from all others. The implementation of object identifiers has a considerable impact on how the rest of the system is implemented and on its performance. Implementation of the identity of persistent objects generally differs from that of transient objects. The object identities allocated by the OO programming languages such as C++ and Smalltalk are valid only within a single address space because they are just memory pointers, so handling reference in a shared environment is hard. In a shared object environment, the object identities must be unique within that environment. This implies that there must be a mechanism for controlling the allocation of such identities. The implementation of persistent object identity has two common solutions, based on either *physical* or *logical identifiers* [3, 29] with their respective advantages and shortcomings. Perhaps the simplest implementation of the identity of an object is the physical address of the object. The physical identifier approach equates the OID with the physical address of the

corresponding object. The physical storage address can be a disk page address and the byte offset within the page. While these physical identifiers can offer performance advantages on certain local operations since the object can be obtained directly from the OID, this comes at a high cost in flexibility. For example, physical identifiers can make schema evolution very difficult. Schema changes typically require instances to be moved, which changes their physical address and invalidates all references to the object. This means there is no location independence with this approach. In contrast, the logical identifier approach consists of allocating a system-generated and globally unique OID (i.e. a surrogate) per object. Surrogates are the most powerful technique for supporting identity because they are completely independent of any physical location. Each object of any class is associated with a globally unique surrogate when an object is instantiated. This surrogate is used to internally represent the identity of its object throughout the lifetime of the object. A logical OID is invariable and position independent, it allows transparent storage reorganization and there is no overhead due to object movement.

For the POServer system, the logical identity is used because schema evolution and object distribution are two of the four important requirements outlined. This approach gives the SM the flexibility to move and cluster objects around in secondary storage as necessary to achieve scaleable performance. All references within an application to a particular object can remain the same even if that object is moved or reorganized. The mechanism for the object identifiers in POServer is similar to that of Orion [30]. Two

types of OIDs exist in POServer. One type of object identifiers are permanent identifiers, known as *Logical Object Identifiers* (LOIDs), which is unique across multiple systems and independent of physical object location. The counterpart of the LOIDs at the physical storage level are called the *Physical Object Identifiers* (POIDs) which provide the actual location of an object on disk. A POID consists of three parts: a 1-hex partition number, a 4-hex page number and a 3-hex offset. The 3-hex offset can address up to 4 kilobytes of storage, i.e. 1 page. The 1-hex partition number allows a class to have up to 16 file partitions. The size of each file partition is calculated as 2^{16} * 4K = 256 Megabytes (M). The maximum size for all objects in a class is therefore determined by 256M * 16, which gives 4 Gigabytes of storage. The Object Manager controls the allocation of the LOIDs and ensure their uniqueness, whereas the POIDs are determined by the Storage Manager. The higher levels of POServer, including the Object Manager and the user APIs, use LOIDs to represent object references. LOIDs are visible to the application and can be passed around. Usually, end users consider LOIDs the only object identifiers in the system because they do not know or see the POIDs. The structure of a logical OID consists of a 4-byte *class identifier* (CID) and a 4-byte *instance identifier* (IID). The CID is the identifier of the class to which the object belongs and the IID is essentially a serial number to resolve identity within a class. Each IID can hold up to 2^{32} (4 billion) values. The class identifier is designed to be long enough to allow it be globally unique and the instance identifier is large enough to avoid reuse of values under any probable conditions. The following are some examples of LOIDs:

Class Class	00000001:00000001
Class Department	00000001:00001001
Class Employee	00000001:00001002
Department Application Services	00001001:00000020
Employee Dave Jones	00001002:00000020

This should be contrasted with Smalltalk's use of a 4-byte *memory pointer* [23] to implement identity for both classes and instances. The message processing may be somewhat inefficient because run-time type checking becomes expensive. When a message is sent to an object, the types of the objects referenced in an object can only be determined by actually fetching the objects and examining the class identifiers stored in them. This implies that invalid messages cause unnecessary fetching of objects. The obvious advantage of our strategy is flexibility because the POServer system can extract the CID directly from the LOID and then look up the class object to determine if the message is valid or not. Of course, the flexibility is at the expense of one table look-up per object access.

It is essential for POServer to provide an efficient mechanism for mapping the LOID of an object to the POID storage address of the object. The Class Tuple can be used for this purpose. The POServer system maintains a pair [IID, POID] in a table called the

Instance ID (IID) table. The **lookup** behavior in class Tuple can be used to search for the corresponding POID for an object. There is one IID table for each class in the system and it is referenced by the attribute **instance_table** of the class. The IID table is persistent by default. It might be very large as it contains one entry for each object in the system. It is also a highly demanded resource in a multi-user environment. Figure 5.1 shows how classes and their IID tables are represented in disk storage. The tuples in the IID tables are shown as [IID, *object_name*] for illustration purpose. The *object_name* should actually be the corresponding POID. The contents of a class in Figure 5.1 are also simplified, only the attributes **class_name** and **instance_table** are shown to emphasize the relationship between a class and its IID table. The CID for class Class is 00000001 and its **instance_table** refers to all the class objects, for example, class Department has IID 00001001 and class Employee has IID 00001002 assigned. As a result, all LOIDs for Department objects start with 00001001 such that 00001001:00000020 identifies Department Application Services. Another important point to note is that the POServer system simply stores LOIDs as part of the attributes of an object. Therefore, the LOID 00001001:00000020 is embedded in the attribute :jones.**dept** and is physically stored on disk.

Class Class and its IID table

(00000001:00000001, [**class_name**:'Class', **instance_table**:00000010:00000001])

(00000010:00000001, {[00000001, Class], [00000002, Method], [00000003, Attribute],
[00000010, Tuple], [00001001, Department], [00001002, Employee]})

Class Department and its IID table

(00000001:00001001, [**class_name**:'Department', **instance_table**:00000010:00000001])

(00000010:00001001,{[00000010,Technical Services],[00000020, Application Services],
[00000030, Operational Services]})

Class Employee and its IID table

(00000001:00001002, [**class_name**:'Department', **instance_table**:00000010:00000001])

(00000010:00001002, {[00000010, John], [00000020, Dave], [00000030, Rob],
[00000040, Jim], [00000050, Pat], [00000060, Don]})

Department Application Services

(00001001:00000020, [**name**:'Application Services', **number**:130, **location**:'5s'),
emps:{00001002:00000020, 00001002:00000050})

Employee Dave Jones

(00001001:00000020, [**name**:'Dave Jones', **dob**:'1962-01-01', **job_grade**:9,
dept:00001001:00000020)

Figure 5.1 Object storage representations and the IID tables

One fundamental operation of the OM is to retrieve the embedded object that is referred to by a source object. This is known as *dereferencing* of the embedded LOID. To dereference an embedded LOID, for example :jone.**dept**, POServer first determines whether the required object is already loaded in the Object Access Table (OAT). If it is, dereferencing simply means setting the memory pointer for :jones.**dept** to the corresponding department object descriptor (OD) in memory. Whenever the OM receives a request for an object whose LOID is not currently in the OAT, it requests the page containing that object from the SM and dereferencing involves the following steps. Figure 5.2 presents a graphical representation of the steps required to locate and retrieve an object on disk. Assuming that we are trying to perform dereferencing for :jones.**dept** 00001001:00000020 because it does not exist in the OAT.

1. First, the CID 00001001 is used to search for the target class object in the OAT. Since the CID for class Class is 00000001, therefore the LOID for class Department becomes 00000001:00001001. The search operation is then translated into OAT.**lookup**(00000001:00001001).

2. The object entry for class Department is returned in step 1. By following the object descriptor pointer the class Department object is accessed.

3. The next step is to locate the corresponding IID Table. Each class contains an IID table. The IID table, DeptIID, for the class Department can be accessed by invoking Department.**instance_table**.

4. The behavior Department.**physical_file** is used to retrieve the path for physical file Department.

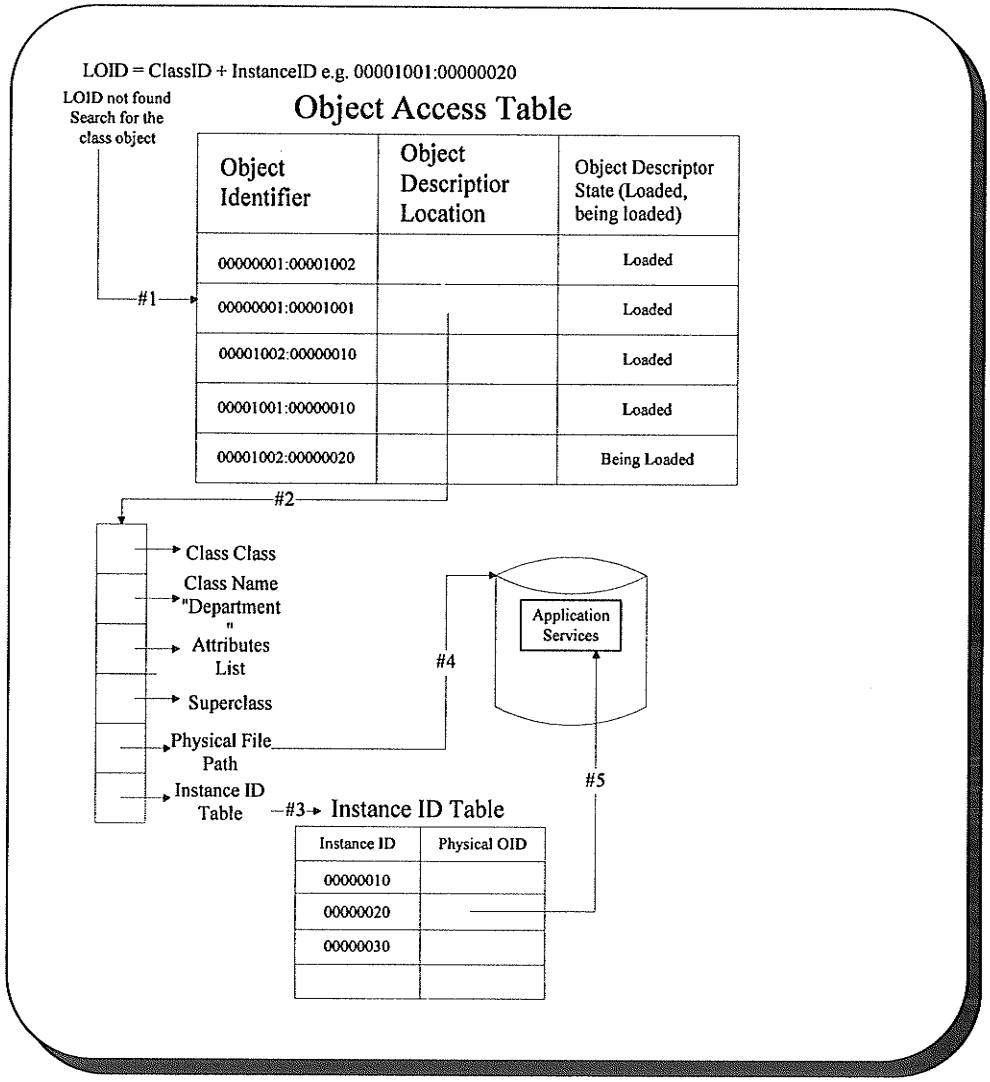

Figure 5.2 Object Dereferencing and Instance ID Table

5. This step maps the instance id 00000020 to its POID. Search the IID Table for the corresponding POID which contains the physical address, page# + offset, of the Department object. This is achieved by invoking DeptIID.**lookup**(00000020).
6. A new object entry for the department object to be fetched is inserted into the Object Access Table and the OD status is set to 'being loaded'.
7. Perform the Object Manager Interface routine SM_ReadObject(PhysicalFile, POID, &ObjectPtr) to retrieve the page containing the object from disk back to the Object Manager. The returned object pointer is simply the base pointer for the page plus the offset.
8. Once the object pointer, &ObjectPtr, is obtained from step 7, the OM then uses the object structure in the class to build the OD. The OD status is set to 'loaded' and the addressability to the OD is established accordingly.
9. Finally, the memory pointer for :jones.**dept** is set to the newly created Object Descriptor. The dereferencing is now complete and :jones.**dept.name** becomes accessible.

Class objects may be cached to optimize system performance. Note that the object descriptors are set up via direct pointers to the fetched object's data in the SM client's cache without incurring any memory allocation or copying overhead. The objects are manipulated directly on the cache page on which they reside. This eliminates the need for *pointer swizzling* [23]. The pointer swizzling technique involves expensive

conversion of the POID of an object and the converse unswizzling operation when sending a page back to the SM server. The object descriptors also supports efficiently the copy semantics of values because only a new OD needs to be created for a copy operation and the object values can be shared.

5.3 Object Manager Interface (OMI)

In this section the OMI routines are discussed. Object Manager Interface specifies the interface between the Object Manager and the SM client. The key operations include: 1. Retrieving an existing object; 2. Creating a new object; 3. Deleting an object; and 4. Updating an object. For reading an object, the OMI provides a routine, SM_ReadObject, to return a pointer to the object within a given disk page; the desired byte range can be calculated based on the object structure of the class. For updating an object, another routine, SM_WriteObject, is provided to tell SM that a subrange of the bytes in the page have been modified. When the behavior object.**persistent** is invoked, the corresponding OMI routine SM_CreateObject is executed. It saves the object into the PhysicalFile and returns its physical object identifier. Similarly, when the behavior object.**transient** is invoked, the OMI routine SM_DeleteObject is executed. It removes the object from the PhysicalFile where the key is equal to the POID. The following are the specifications for the key OMI routines:

```
SM_ReadObject(PhysicalFile, POID, &ObjectPtr)   /* returns a pointer to the object */
SM_CreateObject(PhysicalFile, ObjectString, &POID)      /* returns the physical OID */
SM_DeleteObject(PhysicalFile, POID)                     /* deletes an object */
SM_WriteObject(PhysicalFile, POID, ObjectString)        /* updates an object */
```

OMI might also include other routines to insert a sequence of bytes at a given point in the object, to append a sequence of bytes to the end of the object, and to delete a sequence of bytes from a given point in the object.

Chapter 6

Conclusion

The intent of this thesis was to demonstrate the design and implementation of a simple persistent object server and this is precisely what has been achieved. In this thesis, an architecture for a persistent object server called POServer is presented, the conceptual object model defined, and the implementation design of the key structures and functions in the system are dicussed. Throughout the examination of various implementation techniques, it is evident that building an object-based system is a very important yet very difficult task. There are many ways a persistent object server can be implemented. Two significant approaches to developing an object server were examined and compared in the thesis. The POServer system is based on a client/server architecture which is designed to make use of the very attractive remote procedure call (RPC) facility from CMU. RPC technologies have evolved to provide standard communication mechanisms and Mach's RPC can provide location transparency through the name services. Using the features provided in Mach, POServer is simple to implement as a prototype system which can offer great flexibility. The two main components of the architecture are an object manager and a storage manager. From one viewpoint, the Object Manager is the analogy of a relational manager, while the Storage Manager is the analogy of a file system.

The development of an object model and object manipulation languages has been the focus of the research on object-based systems. The POServer object model described attempts to provide the basic constructs for a fundamental object-based system. The three constructs defined are *object*, *method* and *class*. Four abstract classes exist: *primitive*, *construct*, *schema* and *user-defined*. The fundamental data types (e.g. character, boolean, real, integer, etc.) and three most common constructed classes (e.g. set, list and tuple) are included in the POServer class hierarchy. In addition, the object model defines the protocols for the most common behaviors in each class. An application programmer only need to understand the protocol specifications to use the classes effectively. The object model also demonstrates the classification mechanism and the capability of representing complex objects. It is important to note that the primitive class hierarchy can easily be extended to cover more basic data structures or to augment additional object-oriented semantic modelling concepts. Objects interact with other objects by invoking their methods. The Object Manager consists of the data structures (e.g. the *object descriptor* and the *object access table*) to support object management. POServer is much more than just a persistent storage manager, it also supports manipulation of objects through API's. The object manipulation language illustrated is an upwardly compatible object-oriented extension to the SQL relational database language borrowed from the Object Management Group (OMG).

6.1 Contributions

In general, this thesis shows and examines the techniques and algorithms required to design and implement a simple object server. The system architecture, the object model and the algorithms form the framework of the POServer system. This thesis makes an important contribution as part of the research project for building a Distributed Object-Based system in the following ways:

1. The set up of the Mach system in the DB lab is considered a significant step because it allows the team members in the project to carry out other system developments and testing.

2. The use of Mach's IPC facility to implement the POServer RPC was validated. The RPC was tested by running Mach in multi-user mode using the two DecStations. Some of the primitive classes were implemented as the code illustrated in Figure 4.5. Mach's C language interface was tested and the use of C's function pointer was proven to be very efficient.

3. Although the object model presented in this thesis includes only the primitive data types, its generalization to other structured types should be straightforward.

4. The data structures defined and the algorithms introduced form the basic foundation of an object server and enable us to take one step closer to building a full-blown Distributed Object-Based System.

6.2 Future Research

There are still many aspects of an object manager which were not addressed in this thesis and require further work. Some open problems in object management are discussed below and the discussion attempts to identify interesting problems that remain. In this section some guidelines for future research which would help enhance the POServer system are proposed:

1. Schema versioning is a problem that has existed in relational databases. With object identity, objects can be uniquely tracked throughout their lifetime. The POServer object model could easily be extended to capture historical versions because the identity is a property that can be maintained across structural and content modifications of an object.

2. Another area in which the database research community plays a vital role is distributed systems. Distributed systems enable the sharing and integration of data and resources across computer systems. The benefits of combining distributed computing and object-oriented concepts are undeniable. The study of *Distributed Object-Based Systems* [8, 27, 41] has emerged as one of the most active research fields in database systems. For POServer to work in a distributed environment, the two-part logical object identifier, [class_id.instance_id], might have to be extended to include the site identifier (e.g. [site_id.class_id.instance_id]) of the site in which the object is created. When site identifier is included, each object becomes

self-contained in the distributed environment. All the knowledge that is needed for communication is encapsulated within the object.

3. POServer needs to manipulate a lot of complex dynamic data structures, for instance, the object descriptors and object caches. Algorithms for a good garbage collector are required to reclaim space automatically for unused objects and system structures. Also, referential integrity can be provided through garbage collection.

4. For a complete and sophisticated object-based system, other architectural issues include efficient object query optimization, nested transaction management, object serializability and extended security model.

Bibliography

1. M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Computer Science Department, Carnegie Mellon University, 1986.

2. M. Atkinson and O.P. Buneman. Types and Persistence in Database Programming Languages. ACM Computing Surveys, 19(2):105-190, June 1987.

3. Bancilhon et al. The Object-Oriented Database Manifesto. Proceedings of Conference on Deductive and Object-Oriented Database, December, 1989.

4. K. Barker, M.Evans, R. McFadyen and K. Periyasami. A Formal Onthological Object-Oriented Model. Technical Report TR 92-02, Department of Computer Science, University of Manitoba, March 1992.

5. R. Baron, D. Black, W. Bolosky, J. Chew, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach Kernel Interface Manual. Computer Science Department, Carnegie Mellon University, 1987

6. A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, vol. 2, pp. 39-59, Feb. 1988..

7. D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel Operating System Architecture and Mach. Journal of Information Processing, Vol. 14(4), pp. 442-453, 1991.

8. F. Boyer, J. Cayuela, P. Chevalier and A. Freyssinet. Supporting an Object-Oriented Distributed System: Experience with UNIX, Mach, Chorus. SEDMS II. Symposium on Experiences with Distributed and Multiprocessor Systems, pp. 283-299, 1991.

9. J. Boykin, D. Kirschen, A. Langerman and S. Loverso. Programming Under Mach. Reading, MA: Adison-Wesley, 1991.

10. P. Butterworth, A. Otis and J. Stein. The GemStone Object Database Management System. Communication of the ACM, 34(10), October, 1991.

11. M.J. Carey, D. DeWitt, J. Richardson, E. Shekita. Object and File Management in the EXODUS Extensible Database System. Proceedings of the 12th International Conference on Very Large Databases, 1986.

12. K. Christian. The UNIX Operating System. New York: John Wiley & Sons, 1988.

13. P. Coad and E. Yourdon. Object-Oriented Analysis. Yourdon Press, Englewood Cliffs, NJ, 1990.

14. E.F. Codd. A Relational Model for Large Shared Data Banks. Communication ACM, 13(6), pages 377-387, October 1970.

15. G. Copeland and D.Maier. Making Smalltalk a Database System. SIGMOD'84, Proceedings of Annual Meeting, SIGMOD Record, Vol. 14, No. 2, pp. 316-325, 1984.

16. C.J. Date. An Introduction to Database Systems. Addison-Wesley, 1986.

17. O. Dahl and K. Nygaard. Simula - An Algol-Based Simulation Language. Communications of the ACM, Vol. 9, No. 9, pp. 671-678, 1966.

18. O. Deux et al. The Story of O_2. IEEE Transactions on Knowledge and Data Engineering, 2(1), March 1990.

19. O. Deux et al. The O_2 System. Communications of the ACM, 34(10), October 1991.

20. W. Effelsberg and T. Harder. Principles of Database Buffer Management. ACM Transaction Database Systems. December 1984, 9(4), 560-595.

21. M.A. Ellis and B. Stroustrup. The Annotated C++ Reference Manual. Reading, MA: Addison-Wesley. 1990.

22. D.H. Fishman et al. Overview of the Iris DBMS. In W. Kim and F. Lochovsky, editors, Object-Oriented Systems, Databases and Programming, pages 174-199. Addison-Wesley Publishing Co., Inc., Reading, Massachusetts, 1989.

23. A. Goldberg and D. Robson. Smalltalk-80: The Language And Its Implementation. Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

24. D. Golub, R. Dean, A. Forin and R. Rashid. UNIX as an Application Program. Proc. of the USENIX Summer Conf., pp 87-95, 1990.

25. L. Haas, W. Chang, G. Lohman, J. Mcpherson. Starburst Mid-Flight: As the Dust Clears. IEEE Transactions on Knowledge and Data Engineering, Vol 2. No. 1, March 1990.

26. B. Irani. Implementation of the TIGUKAT Object Model. Technical Report TR 93-10, June 1993.

27. M. Jones and R. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. In OOPSA'86 Proceedings, Portland, Oregon, Pages 67-77, 1986.

28. M.A. Ketabchi, S. Mathur, T.Risch, J. Chen. Comparative Analysis of RDBMS and OODBMS: A Case Study. IEEE Transactions on Knowledge and Data Engineering. May, 1990.

29. W. Kim. Object-Oriented Database Systems: Strengths and Weaknesses. Journal of Object-Oriented Programming, pages 21-29, July-August 1991.

30. W. Kim, N. Ballou, H.T. Chou, J.F. Garza and D. Woelk. Features of the ORION Object-Oriented Database System. In Kim Won and Lochovsky F.H., editor, Object-Oriented Concepts, Databases and Applications. ACM Press, 1989.

31. W. Kim. Object-Oriented Databases: Definition and Research Directions. IEEE Transactions on Knowledge and Data Engineering. Vol. 2. No. 3. September 1990.

32. S. Koehler. Objects in insurance. Object Magazine July-August 1992.

33. G. Krasner. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley, Reading, MA, 1983.

34. C. Lamb, G. Landis, J. Orenstein and D. Weinreb. The ObjectStore Database System. Communication of the ACM, 34(10), October 1991.

35. S.J. Leffler, M.K. Mckusick, M.J. Karels and J.S. Quarterman. The Design and Implementation of the 4.3 BSD UNIX Operating Systems Principles. Reading, MA: Addison-Wesley, 1989.

36. G.M. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to Startburst: Objects, Types, Functions, and Rules. Communications of the ACM, 34(10), October 1991.

37. M.E.S. Loomis. OODBMS: The Basics. Journal of Object-Oriented Programming, 3(1), pages 77-81, 1990.

38. G. Malan, R. Rashid, D. Golub and R. Baron. DOS as a Mach 3.0 Application. School of Computer Science. Carnegie Mellon University. 1993.

39. B. Meyer. Lessons from the design of the Eiffel libraries. Communications of the ACM, 33(9), pp. 68-88. 1990.

40. J.E.B. Moss, S. Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable Shared Object Interface. In Advances in Object-Oriented Database Systems, K.R. Dittrich, Ed. Second International Workshop on Object-Oriented Database Systems, Springer-Verlag, 1988.

41. S.J. Mullender. Interprocess Communication. In S. Mullender, editor, Distributed Systems, pages 37-65, ACM Press, 1989.

42. B. Musteata and R. Lesser. DB2 Handbook. Computer Technology Research Corp., N. Y. TLM, Inc., 1988.

43. E. Nemeth, G. Snyder and S. Seebass. UNIX System Administration Handbook. Englewood Cliffs, NJ: Prentice Hall, 1989.

44. O. Nierstrasz. A survey of Object-Oriented Concepts. In W. Kim and F.H. Lochovsky, editors, Object-Oriented Concepts, Databases and Applications. ACM Press, 1989.

45. Object Management Group. The Common Object Request Broker: Architecture and Specification Revision 1.1. Object Management Group, Framingham, MA, 1992.

46. H. Osher. Distributed Object Management. Object Magazine, September/October 1991.

47. M. T. Ozsu. and P. Valduriez. Principles of Distributed Database Systems. Englewood Cliffs, NJ: Prentice-Hall, 1990.

48. F. Rabitti and W. Kim. A Model of Authorization for Next-generation Database Systems. ACM Transactions on Database Systems, Vol. 16, No. 1, March 1991, Page 88-131.

49. M. Satyanarayanan. Distributed File Systems. In S. Mullender, editor, Distributed Systems, pages 149-183, ACM Press, 1989.

50. H. Schildt. C The Complete Reference. Berkeley, California: McGraw-Hill, 1987.

51. E. Seidewitz. Object-Oriented Programming in Smalltalk and Ada. ACM OOPSLA '87 Proceedings, October 1987.

52. S. Shafer, M. Thompson. The SUP Software Upgrade Protocol. Carnegie Mellon University. School of Computer Science. 1989.

53. J. Shirley. Guide to Writing DCE Applications. Sebastopol, California: O'Reilly & Associates, 1992.

54. B. Tay and A. Ananda. A Survey of Remote Procedure Calls. Operating Systems Review, vol 24, pp. 68-79, 1990.

55. M. Stonebraker, L.A. Rowe, and M. Hirohama. The implementation of POSTGRES. IEEE Transactions on Knowledge and Data Engineering, 2(1), March 1990.

56. A. S.Tanenbaum. Modern Operating Systems. Englewood Cliffs, NJ: Prentice Hall, 1992.

57. M.R. Thompson and R.P. Draves. Building Mach 3.0. Computer Science Department, Carnegie Mellon University, 1992

58. M.R. Thompson. Setup for Mach 3.0. Computer Science Department, Carnegie Mellon University, 1993.

59. L.R. Walmer and M.R. Thompson. A Programmer's Guide to the Mach User Environment. Computer Science Department, Carnegie Mellon University, 1988.

60. M. Zapp and K. Barker. An Architecture and Model for Transactions in Object Bases. Technical Report TR 92-09, Department of Computer Science, University of Manitoba, July 1992.