KALICI NESNELERİ DESTEKLEYEN
GENİŞLETİLEBİLİR BİR SİSTEM TASARIMI

DOKTORA TEZİ
Y. Müh. Erdal KEMİKLİ
504920011012

Tezin Enstitüye Verildiği Tarih : 26 Şubat 2002
Tezin Savunulduğu Tarih: 30 Ocak 2004

Tez Danışmanı: Prof. Dr. Nadia ERDOĞAN
Diğer Jüri Üyeleri Prof. Dr. A. Emre HARMANCI (İTÜ.)
                 Prof. Dr. Oğuz TOSUN (BÜ.)
                 Prof. Dr. Oya KALIPSIZ (YTÜ.)
                 Prof. Dr. Tevfik AKGÜN (YTÜ.)

OCAK 2004
DESIGN OF AN EXTENSIBLE SYSTEM SUPPORTING PERSISTENT OBJECTS

Ph.D. Thesis by
Erdal KEMİKLİ, M.Sc.
504920011012

Date of submission: 26 February 2002
Date of defence examination: 30 January 2004

1. Supervisor (Chairman): Prof. Dr. Nadia ERDOĞAN

Members of the Examining Committee

Prof. Dr. A. Emre HARMANCI (İTÜ.)
Prof. Dr. Öğuz TOSUN (BÜ.)
Prof. Dr. Oya KALIPSIZ (YTÜ.)
Prof. Dr. Tevfik AKGÜN (YTÜ.)

JANUARY 2004
PREFACE

Computer systems are one of the most complex artifacts of the human mind. The complexity of these systems are the main motivation of research and development in software development and systems architecture. New studies are crucial in improving productivity and tackling with the complexity of new systems design.

This dissertation includes a new systems development framework and programming environment. It extends the well known notion of persistency in programming into systems development and abstraction.

I want to thank my thesis advisor Professor Dr. Nadia ERDOĞAN who was supportive yet not restrictive during my thesis research. It would have been impossible to complete this study without her suggestions, guidance and support.

February, 2002

Erdal KEMIKLİ
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABBREVIATIONS</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
<tr>
<td>ÖZET</td>
<td>xi</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>xiii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. PERSISTENT TECHNOLOGY AND RELATED WORK</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Object Oriented Model</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Operating Systems</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Object Oriented Operating Systems</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Extensible Systems</td>
<td>8</td>
</tr>
<tr>
<td>2.4.1 SPIN</td>
<td>9</td>
</tr>
<tr>
<td>2.4.2 Oberon system 3</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Persistent Technology</td>
<td>12</td>
</tr>
<tr>
<td>2.5.1 The concept of persistency</td>
<td>12</td>
</tr>
<tr>
<td>2.5.2 Persistent languages</td>
<td>13</td>
</tr>
<tr>
<td>2.5.2.1 PS-Algol</td>
<td>15</td>
</tr>
<tr>
<td>2.5.2.2 E language</td>
<td>15</td>
</tr>
<tr>
<td>2.5.2.3 Persistent Smalltalk</td>
<td>16</td>
</tr>
<tr>
<td>2.5.3 Persistent systems</td>
<td>17</td>
</tr>
<tr>
<td>2.5.3.1 Stability</td>
<td>18</td>
</tr>
<tr>
<td>2.5.3.2 Address translation</td>
<td>19</td>
</tr>
<tr>
<td>2.5.3.3 Process model and protection</td>
<td>19</td>
</tr>
<tr>
<td>2.5.3.4 Operating system kernel</td>
<td>20</td>
</tr>
<tr>
<td>2.5.4 Persistent store</td>
<td>21</td>
</tr>
<tr>
<td>2.5.5 Examples of persistent operating systems</td>
<td>21</td>
</tr>
<tr>
<td>2.5.5.1 Grasshopper</td>
<td>21</td>
</tr>
<tr>
<td>2.5.5.2 SOS</td>
<td>22</td>
</tr>
<tr>
<td>2.5.5.3 Mungi</td>
<td>23</td>
</tr>
<tr>
<td>2.5.5.4 Opal</td>
<td>25</td>
</tr>
<tr>
<td>2.6 Comparison of EPS to Other Systems</td>
<td>25</td>
</tr>
<tr>
<td>3. EPS REQUIREMENTS</td>
<td>27</td>
</tr>
<tr>
<td>3.1 EPS General Architecture Concept</td>
<td>27</td>
</tr>
<tr>
<td>3.2 System Services</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Single Node Implementation</td>
<td>29</td>
</tr>
<tr>
<td>3.4 Hardware and Software Platform Requirements</td>
<td>29</td>
</tr>
<tr>
<td>3.5 Programming Interface Requirements</td>
<td>29</td>
</tr>
<tr>
<td>3.5.1 Object loading strategy</td>
<td>30</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Type compatibility</td>
</tr>
<tr>
<td>3.6</td>
<td>Security Requirements</td>
</tr>
<tr>
<td>3.7</td>
<td>Reliability Requirements</td>
</tr>
<tr>
<td>3.8</td>
<td>Performance Expectations</td>
</tr>
<tr>
<td>3.9</td>
<td>Persistent Store</td>
</tr>
<tr>
<td>3.10</td>
<td>Supported Application Types</td>
</tr>
<tr>
<td>4.</td>
<td>PROGRAMMING IN EPS</td>
</tr>
<tr>
<td>4.1</td>
<td>EPS-C Programming Language</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Persistent variable management</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Mandatory operations</td>
</tr>
<tr>
<td>4.2</td>
<td>Active Object Development</td>
</tr>
<tr>
<td>4.3</td>
<td>EPS-C Preprocessor</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Compilation and linking</td>
</tr>
<tr>
<td>4.4</td>
<td>Application Programmer's Interface</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Session primitives</td>
</tr>
<tr>
<td>4.4.2</td>
<td>General object primitives</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Table object primitives</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Communication primitives</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Capability primitives</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Active object primitives</td>
</tr>
<tr>
<td>5.</td>
<td>EPS DESIGN</td>
</tr>
<tr>
<td>5.1</td>
<td>EPS System Architecture</td>
</tr>
<tr>
<td>5.2</td>
<td>Inter-Object Communication System</td>
</tr>
<tr>
<td>5.3</td>
<td>Client Object Library</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Client object library services</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Address translation</td>
</tr>
<tr>
<td>5.4</td>
<td>Naming and Protection Server</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Naming and protection services</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Object name resolution</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Request validation and access synchronization</td>
</tr>
<tr>
<td>5.5</td>
<td>Object Server</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Data integrity</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Object store on disk</td>
</tr>
<tr>
<td>5.6</td>
<td>Active Object</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Active object library</td>
</tr>
<tr>
<td>5.7</td>
<td>EPS Security Architecture</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Capability handling</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Access control</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Heap management and garbage collection</td>
</tr>
<tr>
<td>6.</td>
<td>EPS PERFORMANCE</td>
</tr>
<tr>
<td>6.1</td>
<td>EPS Program Execution Performance</td>
</tr>
<tr>
<td>6.2</td>
<td>EPS Programmer Productivity</td>
</tr>
</tbody>
</table>
ABBREVIATIONS

AcM : Access Mode
AMs : Answer message
AOB : Active object
AOL : Active object library
CaV : Capability variable
COB : Client Object
COL : Client Object Library
DBMS : Database Management System
DBN : Database name
DON : Data Object Name
EPS : Extensible Persistent System
EPS-C : EPS flavour of C programming language
IOC : Inter-object Communication
IP : In parameter
LoV : Local variable
NPS : Naming and protection server
OBS : Object server
OODB : Object Oriented Database
OP : Out parameter
Ops : Operation
PeD : Persistent data
RM s : Request message structure
Rs : Result structure
SOB : Sender object
Tc : Time to convert object representation between disk and memory
TOB : Target object
Tm : Message transfer time
Tp : Time for process on object
Tr : Time to read an object from disk
Tw : Time to write an object to disk
Tt : Time for address translation
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Session primitives table</td>
<td>41</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Object primitives table</td>
<td>42</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Table object primitives table</td>
<td>43</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Communication primitives table</td>
<td>44</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Capability primitives table</td>
<td>45</td>
</tr>
<tr>
<td>Table 4.6</td>
<td>Active object primitives table</td>
<td>46</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>EPS file types</td>
<td>48</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Message structure</td>
<td>48</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Object structure</td>
<td>51</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Client object services</td>
<td>52</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>Type table structure</td>
<td>53</td>
</tr>
<tr>
<td>Table 5.6</td>
<td>InUse table structure</td>
<td>55</td>
</tr>
<tr>
<td>Table 5.7</td>
<td>Access rights state transitions</td>
<td>58</td>
</tr>
<tr>
<td>Table 5.8</td>
<td>Persistent object on disk</td>
<td>61</td>
</tr>
<tr>
<td>Table 5.9</td>
<td>Capability functions</td>
<td>63</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>B-tree loading times for C and EPSC programs</td>
<td>66</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>B-tree display times for C and EPSC programs</td>
<td>67</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>Conventional program execution time breakdown</td>
<td>67</td>
</tr>
<tr>
<td>Table 6.4</td>
<td>EPS client object execution time breakdown</td>
<td>68</td>
</tr>
<tr>
<td>Table 6.5</td>
<td>Programmer productivity</td>
<td>68</td>
</tr>
<tr>
<td>Table 6.6</td>
<td>Comparison of C and EPS-C program source</td>
<td>69</td>
</tr>
<tr>
<td>Table 6.7</td>
<td>Classification of object complexity</td>
<td>70</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page No</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>1.1</td>
<td>EPS model</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>EPS system concept</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Adding names to a b-tree</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>EPS system layers</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>EPS modules</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>Session architecture</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>Table object</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>Address translation</td>
<td>54</td>
</tr>
<tr>
<td>5.6</td>
<td>Message flow among NPS and other objects</td>
<td>54</td>
</tr>
<tr>
<td>5.7</td>
<td>Persistent store levels</td>
<td>59</td>
</tr>
<tr>
<td>5.8</td>
<td>Capability stores</td>
<td>63</td>
</tr>
</tbody>
</table>
KALICI NESNELERI DESTEKLEYEN GENİŞLETİLEBİLİR BİR SİSTEM

ÖZET

Yazılım, bilgisayar sistemlerinin en önemli parçası haline gelmesine rağmen, yazılım üretimi kronik bir verimsizlik krizinin sıkarısı içinde. Bu probleme çözüm olarak toplam kalite yönetimi ve tümleşik yazılım geliştirme ortamlarının da içinde bulunduğu çok sayılara yaklaşıma dikkate alınmaktadır.

İşletim sistemlerinin tasarmında uzun süreli ortada olan bir konu genel amaçlıktır. Genel amaçlı sistemler çok değişik uygulamaları çalıştırabilirken, belirli uygulama tipleri için uygun değillerdir. Sistemi programlama yol ile genişletmek mümkün olamsa rağmen, genellikle çok kolay bir şekilde yapılamaz.

Bu araştırmının temel hedeф programçısı verimliliği problemini teknik bir çözümle azaltacak genişletilebilir ve uyarlanabilir bir sistem tanımlamaktır. Ortaya çıkan sistem değişik uygulama türlerini destekleyebilecek genişletilebilir bir sistemın temelini oluşturmaktaşıdır.


Kalıcı nesneler yaşam süreleri kendilerini yaratan programlardan daha uzun olan ve bu nedenle programlardan gerçek dünyayı daha iyi modellemesini sağlayacak, ve programların bu modeli olusturmak için gerekli kodları yazarın zorunluluğunu ortadan kaldırarak programci verimini artıran bir soytulamadır. KGS modelinin iki temel soytulmasından birisi olan pasif nesneler, sadece veri içeren kalıcı nesnelerdir. Diğer yandan, aktif nesneler ise sadece veri değil metodlar da içerir ve sunucu niteliğindedir. Aktif nesneler sistemin genişletilebilir olmasını sağlayan sunuculardır, diğer çağdaş işletim sistemlerine benzer şekilde sistem servislerinin kullanıcı düzeyinde prosesler ile sağlanmasına olanak verilir. KGS aktif nesnelerin genişirilmesi ve yönetilmesi konusunda gelişmiş bir destek sağlamaraktır.

Yetenek (Capability) tabanlı güvenilir mimarilerde henüz tam olarak adreslenmemiş bir konu daha önce dağıtılan yeteklerin iptalıdır. Bugüne kadar dağıtılan bir hakkın herkese geri alınması konusunda bazı yöntemler önerilmesine rağmen dağıtılan hakların bazı istekçilerden geri alınırken bazılarında birkaçması gerektiren seçimi hak iptal pek çok sistemde bir sorun olarak kalmıştır. KGS bir yeteneğinin sabahının bu yetenekleri tamamen iptal etmesine, kısmen iptal etmesine veya verilmiş erişim haklarını kısıtlamasına olanak sağlamaktır. Bunlara ek olarak KGS'te yeteneklerin programlar arasında naklı için yeni bir yöntem tanımlanmıştır.

KGS senkronizasyon ve ölümcül kilitlemenin önlenmesi için programcı kontrolünde esnek bir yaklaşım önermektedir. Senkronizasyon yaklaşımı endüstride yaygın olarak kullanılan ilişkin veri tabanlarına benzemektedir. Ölümcül kilitlemenin önlenmesi için programçının uygulamalardaki ihtiyaçları karşılayacak esneklikte olan kilitlemenin dereceli veya bir kerede çözümesi seçenekleri verilmiştir. İleride yapılacak çalışmalara temel oluşturacak şekilde dağıtılmış bir işlem mekanizması da sağlanmıştır.

KGS, nesneler arasındaki haberleşme hizmeti, müşteri nesne kütüphanesi, aktif nesne kütüphanesi, isimlendirme ve güvenlik sunucusu, nesne sunucusu ve bir önderleyici ile Linux işletim sistemi üzerinde gerçekleşmiştir.
DESIGN OF AN EXTENSIBLE SYSTEM SUPPORTING PERSISTENT OBJECTS

SUMMARY

Although software has become the most important component of computer systems, software production is suffering from a chronic crisis of unproductivity. Many different approaches including the total quality management and integrated software development environments are considered in response to this problem.

A long standing issue in operating systems design has been the notion of generality. While general purpose systems can run quite different type of applications, in many cases these systems are not suitable for certain types of operations. While extending the system through programming is possible, it is usually not a very trivial task.

The primary goal of this research is to define an extensible and tailorable computing system model which will attack the programmer productivity issue from the technical side. The resulting system is suitable to be used as a base for an extensible system for different types of application domains.

Extensible Persistent System (EPS) suggests a new model for extensible systems based on a unifying view of persistency. Data and processes are viewed as persistent objects, and handled through a uniform interface. The system definition can be developed recursively using the notions of active, and passive objects. We believe this simple yet powerful view of the system modeling will support application developers and system programmers in understanding and extending the system.

Moreover, the resulting extended system will not need extra system administration tasks and complex configuration management. Despite of its revolutionary nature, EPS is easy to learn and adapt since it is built on UNIX operating system.

Persistent objects are objects whose life span are longer than their creator programs, therefore provide a better abstraction for modeling the real world by computer programs, and save programmers from writing code for supplying this model manually. Passive objects, one of the two basic abstractions of EPS, are persistent data only objects. Active object, on the other hand include not only data but also methods and act as servers. Active objects are servers that provide extensibility capability to EPS, they provide system services at user level, they provide system services as user level services, quite similar to other modern operating systems. EPS, provides an comprehensive support for the development and management of active objects.

An open issue for security architectures based on capabilities was the cancellation of once distributed capabilities. While some methods of complete cancellation is proposed, selective cancellation which means to remove access rights of some processes while leaving others valid was a problem in many systems. Another problem in access rights cancellation is the restriction of some rights while leaving others. EPS, lets a capability owner to remove a capability completely, or partially or restrict the once given access rights selectively. In EPS we also tried to apply a unique solution for transfer of capabilities from one process to another.
In a conventional system, variable names and scopes are checked by the compiler according to the language rules, which resolves name conflict problem in the compile phase. Also, since the variable names are visible to the programmer, there is little chance of making mistakes or losing a variable. Another problem with a persistent system arises because of the long life span of persistent objects and their wide scope in contradiction to conventional systems. A persistent object identifier, which is expected to be unique, may have already been used before by another programmer. The process of finding a unique identifier can easily become a tedious job, even impossible. This issue is attacked by a naming structure consisting from two parts; owner object name, and persistent object name. The name resolution algorithm based on this naming structure is a flexible one giving the programmer highest freedom possible without jeopardizing the object safety.

EPS also suggests a flexible, programmer controlled approach to synchronization and deadlock prevention. The synchronization approach is similar to that of relational database management systems, which seems to be a successful solution used widely in the industry. For the deadlock prevention, a choice of gradually or one-step lock release is supplied to the programmer which shall be flexible enough to satisfy different types of application development needs. A distributed transaction mechanism is also provided, that can be a basis for further research.

EPS is implemented by following components; inter-object communication service (IOC), client object library (COL), active object library (AOL), naming and protection server (NPS), object server (OBS), and a preprocessor on Linux operating system.
1. INTRODUCTION

The real utilization of a computer hardware is made possible by the computer software. The most fundamental software of a computer is the operating system, which controls all the computer's resources and provides a base upon which the application programs can be written. A computer is a complex system consisting of many subsystems. Developing programs for such a system handling all the details would be an extremely difficult job if such details were not covered by the operating system support.

In the early years of computers, it became apparent that in some way programmers had to be shielded from the complexity of the underlying hardware (Tanenbaum, 1992). Hence, developments in the hardware architectures almost always were followed by developments in the software layers of the computer systems.

In the first generation computers of vacuum tubes in 1940s, the computer was designed, built, programmed, operated and maintained by the same group of people. All the programming was done in absolute machine language by means of plugboards, and there was no concept of operating systems. The second generation computers were built from transistors were mainly batch systems. The batch jobs were executed in sequence by the first instances of operating systems. Third generation of computers is marked by the use of integrated circuits. The complex and powerful architectures of these computers supply a considerable computing power which could be shared among many simultaneous users. These systems are known as timesharing systems and made considerable contributions to the elaborate features of today's operating systems. The fourth generation of computer architectures used large scale integration technology and the computing power is shared through large networks. This last generation brought new concepts into the operating systems such as user friendly interfaces and issues about networked computers.

With every generation of architecture, operating systems improved with new features and facilities. The developments in operating systems are oriented from three sources of influence. The need of better usability, better utilisation of advanced hardware and better support for software development.
By 1960s a new term was coined in the software industry to define the status of software development activities: "software crisis" (Wayt, 1994). By the time being, it was perceived that software was becoming a larger and larger part of most businesses. However, the quality of the software was poor, and management of the costs and schedules of software development was difficult. In the autumn of 1968 the NATO Science Committee convened some 50 top computer scientists, programmers and managers in the software industry to a meeting to plot a course out of what had come to be known as the software crisis.

Several approaches were adopted to address the problem. Some concentrated on the technical aspects, developing new languages such as Ada and new methodologies such as Yourdon, while the others tried to apply general engineering and total quality management principles to the computer systems development area (Taft and Duff, 1997, Yourdon, 1989).

In a conventional system, programming languages provide very good support for transient data. Data with longer life spans can be supported by a DBMS (Database Management System), or a file system. Data with longer life span than its creator program can be defined as persistent data. The concept of persistency on the other hand suggests that data in a system should be able to persist (survive) for as long as that data is required. As a result, persistent systems provide a uniform abstraction for data management. This uniform abstraction removes the task of converting data model for file to memory or vice versa therefore save programmers from considerable amount of program development task. A research by designers of PS-Algol showed that %30 of program code consists of this load/save/convert operations. In many cases, support for persistency is provided at the programming language level such as PS-Algol (Cockshot et al, 1984, Atkinson et al, 1983). Recently, projects to add persistence to current object oriented programming is very wide spread (Lewis et al, 2000, Atkinson and Jordan, 2000). The persistency support at the programming language level has two drawbacks (Kemikli and Erdogan, 1997); operating systems may not provide necessary support for the implementation, and efforts are duplicated for every new persistent language implementation (Kemikli and Erdogan, 1998). These reasons motivate another approach; implementation of persistency at the operating system level.

Most operating systems are forced to balance generality and specialization. While a general system is suitable for many types of applications with some effort, specialized systems run a specific type of application with less problems. An
extensible system is one that can be changed dynamically to meet the needs of an application (Bershad et al, 1995).

The primary goal of this research is to define a persistent, extensible and tailor able computing system model which attacks the programmer productivity issue from the technical side. This model consists of two basic abstractions. Passive object is data with persistency property. Active object is persistent data and methods, and acts as a server that extends the original system behaviour. New approaches are investigated in the design of object access and synchronization, such as multiple access policies and distributed transactions. The resulting system is suitable to be used as a base for an extendible system with new server functionality.

The Extensible Persistent System (EPS) will supply facilities that will ease developers to extend the functionality of the system with better modelling based on passive/active object abstraction, and simpler programming with persistence support and active object development facilities (Kemikli and Erdogan, 1999, Kemikli and Erdogan, 2002, Kemikli and Erdogan, 2004). Moreover, the resulting extended system will not need extra system administration tasks and complex configuration management. Facilities to develop and manage new behaviors of the system is the basic reason of calling EPS an extensible system. Therefore despite its revolutionary nature as a unified application program development and system model with a new approach to system abstraction, EPS is easy to learn and adapt since it resembles to the most common computing environment of today, UNIX.

The first constraint of the research was that, the computing platform is a centralized system. While the number of research projects on distributed systems is increasing very fast, a clear distinction between this research’s goals and distributed system research was necessary.

The implementation of the model is to be developed using an existing platform. While this choice will ease the implementation process, it will also help to define a more streamlined end product with the existing systems. In this sense, the Linux operating system and the C programming language are chosen as the development platform for the implemented model.

The system philosophy is to supply programmers an easy to learn and easy to extend system with uniform abstractions, give them control over the system and provide them options since they are the critical entities of living computer environments.
The model of extensible persistent system is based on the experience in several different fields of research. The first area of related research is on operating systems. EPS itself being a new component of an operating system depends heavily on the previous work on this field. Another interrelated discipline is programming languages and abstraction models in general, and object-oriented paradigm in particular. The research on persistency in the computer systems is the last but the most important discipline to be mentioned in the list of related research.

EPS model is based on principles of these mentioned disciplines to large extent. While unique combination of principles and ideas used in the model is one contribution of this research, the solutions to certain problems such as advanced synchronization control, advanced support for development of active objects, capability control, which were not fully addressed in the previous persistent systems is another contribution. On the other hand, practical results of the persistent model in computer programming is aimed to be the third major contribution of this research.

<table>
<thead>
<tr>
<th>EPS-C Language and Preprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Object Library</td>
</tr>
<tr>
<td>Client Object Manager</td>
</tr>
<tr>
<td>Naming and Protection Server</td>
</tr>
<tr>
<td>Object Server</td>
</tr>
<tr>
<td>Inter-object Communication</td>
</tr>
</tbody>
</table>

**Figure 1.1 EPS model**

The EPS model developed in this thesis is composed of several components in different layers (figure 1.1). On the highest level, a new programming language, EPS-C which is an extension of the existing ANSI C programming language is defined. Extending an already available language will flatten the learning curve of software developers. The programs written in EPS-C are compiled with the help of the EPS-C preprocessor. During the linking phase a runtime library (Client Object Library- COL) which is responsible of conducting process level operations and hiding the system complexity from the users by supplying a well defined interface
composed of system primitives is linked to the program. Active object library (AOL) supplies basic active object functions ready to developers, is linked to active objects during the link phase. Naming and Protection Server (NPS) is a server process running just above the operating system kernel. It resolves persistent object names into persistent ids, protects objects against unauthorized access, and manages the synchronization of access to the objects. Object server (OBS) is another server component of the system responsible of the movement of objects between long and short term storage devices. The last system component is the inter-object communication (IOC), which supports the whole model through a high level object communication interface based on the well known IPC paradigm of UNIX. Communication primitives are designed for both synchronous and asynchronous communication. This functionality is implemented as a run-time library and linked to every process with the need of communication.

The content of this thesis and the details of the sections are summarized below:

Section two briefly describes the related research areas and a survey of related research activities.

The third section defines the requirements of a new extensible persistent system. The system concept, programming requirements, functional specification, security requirements, and platform requirements are fully explained in this section.

The fourth section explains the programming model in EPS, with detailed design features, program development scenarios, and sample program codes.

The fifth section explains the design of the system to satisfy the requirements described in section three. It includes the assumptions used in the design, EPS system architecture with the server components, data structures, and core system algorithms.

Sixth section discusses programmer productivity and program execution performance of EPS compared to conventional systems.

In the seventh section, contributions of this thesis are summarized, and proposals for future work are discussed.

There are four appendices; Appendix A lists important data structures of EPS, Appendix B gives list and content of source files in the EPS implementation, Appendix C lists four short sample application program codes, and Appendix D explains basic system algorithms.
2. PERSISTENT TECHNOLOGY AND RELATED WORK

In this chapter, some of the related work with significant contributions to this research are explained. After a brief review of related technologies such as object oriented model, object oriented operating systems, and extensible operating systems, two major categories of related work is discussed; persistent languages, and persistent systems.

2.1 Object Oriented Model

An object is defined with its state, behavior and identity (Booch, 1989). The state of the object consists of its attributes and the values of these attributes. The behavior of the object is its reaction to state transitions and external messages. The identity of the object is the unique property that distinguishes an object among the others.

There are two types of relationship among objects. Usage relation defines the client server organization. Including relation is the way objects become components of other objects. Objects with the same structure and behavior are called as a class. Every object is an instance of its class. The most common and strongest relation among the classes is the inheritance. Inheritance is the sharing of other classes structure and behavior, and it can be from a single or multiple sources.

The object notion in the computer science appeared in multiple areas concurrently in mid 1970’s. The development of the object technology is related to several developments:

- Advances in the computer architecture (such as the hardware support to operating systems).
- Advances in programming languages.
- Improvements in programming techniques (modular programming, information hiding).
- Improvements in the database models (Entity - relationship models).
- Research in artificial intelligence.
- Advances in philosophy and cognitive sciences.
The object model consists of several principles. The first principle, abstraction is used as the major method of dealing with complexity. Abstraction defines the necessary attributes of the object that are used to distinguish an object from others by an observer. Another principle, information hiding is hiding the details of an object. Abstraction and information hiding are interrelated concepts. While abstraction deals with the external view of the object, information hiding prevents clients to intervene the details of an object when they ask for an attribute externally visible to them. Modularity is the ability to decompose a system into loosely or tightly coupled smaller parts. This property helps to reduce the complexity of a system and improves the understandability. Hierarchy is the operation of sorting or ranking of abstractions. Persistency is the existence of an object even if the creator of that object has disappeared. While some of these principles exist on all object-oriented systems, some of them do not.

2.2 Operating Systems

Recent research on the operating systems area is focused to different needs and trends. Consequently, research projects with very different foci are conducted by different research communities.

One of the recent trends in operating system design is the notion of microkernels and assigning most of system functions to user level server processes (Boykin and LoVerso, 1990). The most influential microkernel, Mach is one of the giants in the operating systems research community (Rashid, 1989). Originally started at CMU, Mach has become the basis for many research systems.

A crucial user need for fast and reliable systems with guaranteed response characteristics derived the development of real-time operating systems. These type of systems are used in the control of robotics equipment and for other applications of embedded systems where predictable temporal performance is a requirement. As a result of this need, many recent operating systems started to supply real-time processing capabilities.

A natural descendent of object-oriented languages is object-oriented operating systems. Designs based on objects is the distinctive nature of these type of operating systems. More information on object-oriented operating systems is covered in the next section.

The improvements in hardware architectures and networking capabilities, lead the way for operating systems that support parallel and distributed architectures. The
primary motive of the research is the fact that modern workstation operating systems do not provide support for efficient distributed program execution in an environment shared with sequential applications.

2.3 Object Oriented Operating Systems

Two different models can be used in the design of an operating system: Process model, and object model (Goscinski, 1991). The basic difference between these two models is in the organization of functional units and synchronization. On the method level, these two models are considered to be equivalent.

The process model is based on processes and messages. All activities of the system are performed by processes. Processes communicate and coordinate via messages. A typical characteristic of the process model is the existence of mechanisms for process handling.

Object oriented operating systems encapsulate the services and resources into objects. Objects resemble to variables of abstract data types. Type of an object is its definition. This definition is visible to the user and defines the operations on the object. The clients access the system services through method calls to the related object. Thus, objects replace the server processes of the process based operating systems. Operating system objects can be active or passive. While an active object can modify the state of itself or other objects, a passive object can change state only when there is an operation of active objects.

While major problem in object oriented operating systems is slowness due to their inefficient mechanisms, object oriented systems simplify the programming task (Almes et al., 1985). Particularly, in parallel and distributed systems object abstraction protects the programmer from complexity of the systems and shortens development time.

2.4 Extensible Systems

Recent advances in software and hardware are pushing traditional operating systems beyond their limits. Very often this is because these new applications and technologies were not, and in some cases, could not be foreseen at system design time. To accommodate these ever-growing demands, one solution is to allow applications to customize operating systems throughout their life-times.

Conventional operating systems are forced to balance generality and specialization. While a general system can run many programs, only some programs have good
performance. Existing system structures are not suitable for easy specialization, often requiring a considerable amount of programming even for a small change in the system characteristics. Moreover, when a change is implemented to improve the performance for one type application, other applications often perform even worse. An extensible system is one that can be changed dynamically to meet the needs of an application (Bershad et al., 1995). Efforts to build extensible systems have demonstrated a three-way tension between extensibility, safety and performance. While some researchers designed extensible systems based on microkernel architectures, others rely on little languages to safely extend the operating system interface through the use of interpreted code that runs in the kernel.

2.4.1 SPIN

The SPIN operating system has been under development at the University of Washington since 1993 (Bershad et al, 1995). SPIN is an operating system that blurs the distinction between kernels and applications. Applications traditionally live in user-level address spaces, separated from kernel resources and services by an expensive protection boundary. With SPIN, applications can specialize the kernel by dynamically linking new code into the running system. Kernel extensions can add new kernel services, replace default policies, or simply migrate application functionality into the kernel address space. Sensitive kernel interfaces are secured via a restricted linker and the type-safe properties of the Modula-3 programming language. The result is a flexible operating system that helps applications run fast but doesn't crash.

SPIN combines research in systems, languages, and compilers to achieve the three fundamental goals of modern operating systems:

- **Flexibility:** Arbitrary users may customize SPIN by writing and installing new kernel code. User-defined extensions are linked into the kernel's address space and dynamically integrated with the executing system.

- **Safety:** Ignorant and malicious kernel extensions are isolated from critical kernel interfaces via restricted dynamic linking. Restrictions are enforced using the type-safe properties of Modula-3, the programming language in which SPIN and its extensions are written.

- **Performance:** Application-specific extensions access system resources and services with low latency since there are no expensive protection boundaries within the kernel.
SPIN is designed around a core set of services including threads, virtual memory primitives, device drivers, and the extension mechanism. These modules are the foundation for the rest of the system and are required for the machine to boot. Additional features, such as file systems and networking, are implemented in SPIN extensions and dynamically installed into the system while it is running.

SPIN and its extensions are written in Modula-3, a type-safe programming language. Modula-3 offers modern language features such as objects, garbage collection, and threads.

EPS has the similar goals to SPIN on flexibility and safety, aiming to be a system which can be tailored easily but safely. Contrary to SPIN, in EPS system extensions are implemented as modular user level services instead of the kernel modifications.

2.4.2 Oberon system 3

Oberon is simultaneously the name of a programming language and of a modern operating system (Mössenböck and Wirth 1991). The Oberon project was started at the Swiss Federal Institute of Technology in Zürich (ETHZ) in 1985 by Niklaus Wirth and Jürg Gutknecht. It was originally targeted towards in-house built hardware (Ceres workstation, based on the National Semiconductors 32000 processor family). Later, the decision was made to port the system to popular computer hardware, where it would run natively or on top of the operating system of the host. Today, Oberon is available for many computer platforms.

In 1991, Jürg Gutknecht and his group continued the development towards System 3. The goal was to exploit the inherent potential and features of Oberon to a much larger degree, upgrade the system by a concept of composable and persistent objects, complement the textual user interface by a graphical companion and provide support for the ubiquitous network. In 1995, the first official Oberon System 3 release was finished. Since then, the system has been constantly improved and extended. In 1997, the Release 2.2 including a large palette of applications was published together with a comprehensive hypertext-based documentation (Knasmüller, 1997).

The original Oberon system is a single-threaded, single-user, co-operative multi-tasking operating system that runs on bare hardware or on top of a hosted operating system as a single-window application. Oberon System 3 is an extended version that has intrinsic support for persistent objects and for building graphical user interfaces. It presents itself as a hierarchy of modules, many of which export one or
several powerful abstract data types. Application modules simply reuse these data types and do not have to care about their implementation at all.

Highlight of Oberon System 3 can be listed as advanced textual user interface, integrated object support in the kernel, object autonomy and persistence, fully hierarchical composability, powerful GUI framework gadgets, and extensibility on different Levels.

All System 3 objects are persistent by nature. System 3 supports persistence by a binding mechanism that allows objects to be bound to an object library.

Openness and extensibility were key goals of project Oberon from the beginning. With System 3 we can distinguish extensibility on three different levels:

The lowest level corresponds to the simplest case. It comprises the creation of composite objects, i.e. of user interfaces and documents from existing components. This level is accessible to programmers as well as to end-users. It merely requires some familiarity with (depending on the desired method) either the interactive composition tools and the inspector or the description language.

The next level is programmed use of existing gadgets and GUIs, in particular for adding "glue logic" to components and for input and output. This level is supported by a rich procedural interface and in particular by the modules Attributes, Links and GadgetsInOut. No object-oriented programming is needed on this level.

The third level involves developing new components. Here, one should distinguish between two kinds of components: elementary gadgets that do not contain any further gadgets and container gadgets that are able to manage other gadgets. In this context, "developing" means extending the type and message handler code of an existing gadget or defining a new type and programming a new message handler. To support this activity, the System 3 release provides code skeletons, in other words templates that can easily be modified and tailored to individual needs. It should be mentioned that object-oriented programming is essential in this case. In particular, the full basic message protocol must be implemented to allow the new object (type) to plug into the "software bus".

EPS differs from Oberon System 3 in some aspects; Oberon System 3 is based on a proprietary research language, EPS is based on existing open systems. An Oberon System 3 programmer has to gain complete new set of skills including the Oberon language and different programming styles for different levels contrary to the uniform model and relatively small skill set required for EPS.
2.5 Persistent Technology

Considerable amount of research has been conducted to define models of persistent environments ranging from special purpose programming languages to operating systems. We will first explain the concept of persistency, and later describe the related work in the persistent systems area with some of the selected examples.

2.5.1 The concept of persistency

A program creates and manipulates a large amount of data throughout its execution. Each item of data will have a different lifetime. Six broad categories can be described for the lifetime of data (Atkinson et al., 1983):

- Transient results in expression evaluation
- Variables involved in procedure activation (parameters and local variables)
- Global variables and dynamic variables
- Data that exists between executions of a program
- Data that exists between versions of a program
- Data that outlives a program

The first three categories are transient data. This is defined as data that ceases to exist beyond the lifetime of the creating process. The other three are non-transient, or persistent data. Typically, programming languages provide excellent, integrated support for the first three categories of transient data. The other three categories can be supported by a DBMS (Database Management System), or a file system.

The idea behind persistence is simple (Alan Dearle et al., 1992): all data in a system should be able to persist (survive) for as long as that data is required. In this sense persistent systems provide a uniform abstraction over storage, whereas in contemporary systems long lived data is treated in a fundamentally different manner from transient data.

PS-Algol coined the term “orthogonal persistence”, that is, the possibility that any object can be made to persist, independent of its type or the way it is used in the program (Atkinson et al., 1982). Systems which provide orthogonal persistence treat all data identically as persistent objects. This leads to a requirement that objects must be addressed uniformly and moved between long and short term storage in a manner that is transparent to the application programmer.
There are certain advantages of the persistent systems over the transient ones (Sajeev and Hurst, 1992):

- Several programs or different runs of the same program that use data with the same structure do not have to build the structure from scratch every time.
- The programmer is saved from task of writing extra code for I/O operations to transfer the data to a file and then transfer it back into the memory.
- Since data objects can persist, a program that uses the data doesn’t have to run to completion once it starts; execution can be temporarily distributed.
- Since procedures and processes are also first-class objects, they can be reused with the same ease as data structures.

In a conventional operating system two abstractions of data access and storage are provided: virtual memory and files. While we can directly access data in virtual memory, data in files can be accessed using system calls. A persistent system which differs from conventional systems has a different abstraction of data storage environment, the persistent data model. A persistent system unifies virtual memory and the file system to provide a single abstraction of data storage and access. So two services of a conventional operating system is satisfied by one integrated service.

2.5.2 Persistent languages

Most of the early research was focused on persistent programming environments. These research programs generally add the persistency property to existing programming languages such as C++ (E), and S-ALGOL (PS-ALGOL) (Richardson and Carey, 1989).

There are four major issues in designing a persistent programming language in addition to the issues related to a conventional language:

1. Indicating object persistence
2. Addressing persistent object
3. Synchronization of object access
4. Object Type compatibility

The general approach in indicating object persistence is to assume, by default, that every object is transient and then provide facilities for expressing object persistence. This option is based on the following observations:
• Programmers are likely to design the structure of their persistent objects with care due to their long life spans; mentioning that they are persistent adds little difficulty to the program design.

• Transient variables, like index variables and temporary value holders, are more common than persistent objects.

• Reverse assumption will be more destructive and harder to debug in case of programming errors.

Traditional programming languages restrict the life span of an object to the execution time of the scope in which the corresponding variable exists. Therefore, the compiler can calculate the data object’s relative address at the point of the variable declaration, and any referenced variable can be uniquely translated to its address. This is not easy for persistent objects. A persistent object must be uniquely accessible throughout its lifetime, which may span not only the runtime of a single program but also the runtime of several temporarily distributed programs. Capability usage is one of the possible solutions (Mullender and Tanenbaum, 1986).

Concurrent access to persistent objects must be synchronized to protect the object integrity. One common policy is many readers/one writer synchronization.

Programming languages normally employ two categories of type equivalence: the name equivalence of Pascal and structure equivalence of Ada. In the name equivalence scheme, two variables are type compatible if they are of the same type name. In a persistent programming environment, this means that a data object created of one type can be assigned to a variable only of that type name. Structural equivalence, although not considered ideal by some programming language designers, is suitable in persistent programming environments. Every persistent object stores information regarding the structure of its type. Compatibility is checked at runtime by comparing the structure of the corresponding variable in the program with that of the persistent object.

Name based type checking creates two problems; the exact name of the type must be known to every program using a persistent object. Moreover, future programs may use the same type name for an entirely different structure, in which case name equivalence will not satisfy type correctness. On the other hand, structural equivalence is apparently slower and prevents compile time type checking.
2.5.2.1 PS-Algol

PS-Algol was the first attempt at producing a system to support persistence (Cockshot et al, 1984, Atkinson et al, 1983). The designers of the language observed that 30% of the code in a typical program simply moves data between disk and memory, therefore they decided to apply the principles of persistence to an existing programming language with minimal changes. Thus programmers would be faced with the normal task of mastering the programming language but would have the facility of persistence with little or no effort. PS-Algol is implemented as a number of functional extensions to S-Algol. The added functions basically are: open database, commit, close database, and some functions for table operations.

The transfer of persistent data from long term to short term memory is conducted automatically when a reference to a persistent object is made. The data is written back to disk when the commit function is called. Type checking is also performed by the system. When a pointer is dereferenced to yield a value the system must check that the pointer points to a structure with the correct field name. The field name and the structure incarnation must carry around type information to enable this checking to be performed.

The underlying persistent store mechanism (POMS), is responsible of storing the persistent objects. It allows programmers using PS-Algol to write programs without being aware of the distinction between RAM and disk. Any data structure that they build up on the heap can be automatically transferred to disk at the end of a program and then brought back to RAM, allowing to view data manipulation in a uniform conceptual abstraction. Thus there is a uniform view of the entire store available to the programmer.

All objects in PS-Algol carry self identifying type information. While this ensures that objects created by one program are properly type checked in another, there is a tradeoff: the added type information requires space, and type checking must be dynamic. Another drawback concern is buffer space. Since the run-time system reads entire objects, applications requiring very large or very many objects may suffer poor performance.

2.5.2.2 E language

The original motivation for E was to provide programming constructs tailored for implementing a database management system (Richardson, 1989). E is an extension of C++ providing generator classes, iterators, and persistent objects. C++ provided a good starting point with its class structuring features and its expanding
popularity as a systems programming language. Generic classes were added for their utility both in defining database container types, such as sets and indices, as well as in expressing generic operators, such as select and join. Iterators were added as a useful programming construct in general, and as a mechanism for structuring database queries in particular. Both generators and iterators in E were inspired by those in Common Object Lisp (Steele, 1990).

E represents a synthesis of ideas and advances from both programming language and database communities. E is distinguished from its predecessors in being a systems level implementation language rather than a modelling or prototyping language. The E language is based on the EXODUS storage manager as the persistent object store (Hanson et al, 1993).

2.5.2.3 Persistent Smalltalk

Smalltalk and its concepts were originally developed in the 1970s at the Xerox Parc, Palo Alto Research Center (Kay, 1993). The Smalltalk-80 system (the latest in a line of Smalltalk development) was the first commercial Smalltalk system. ST-80 became a commercial product in the late 1980s when it was distributed by ParcPlace Systems.

Smalltalk is an object-oriented programming language integrated with a richly endowed multi-windowed development environment. Smalltalk's integrated development environment has a fun-to-use-factor not previously available in languages requiring edit-compile-link-debug steps. Changes are instantly saved and new editions can be tested quickly.

Developers define object classes in their applications to model the real world. These object classes are organized in an inheritance tree to enable making new objects just like other ones with minor changes. Inheritance reduces code bulk and can enable component reuse when properly engineered. In this way, developers are not limited to a small number of data types like integer, float, string, and decimal.

The fundamental way to indicate that something should happen in Smalltalk is to send a message to an object. For example, an application is typically a collection of new and extended base classes that comes to life when sent a run message. Everything is an object in Smalltalk, which gives developers great flexibility. Smalltalk systems come with extensive class libraries to build upon. Smalltalk has a small number of keywords and is relatively easy to learn compared to C++ or Ada.
The Mneme project at the University of Massachusetts had its primary goal the investigation of techniques for integrating programming languages with database features in order to support cooperative data-intensive applications (Eliot et al. 1990). In particular, the project was concerned with the implementation of efficient object-oriented persistent programming languages based on the Mneme object store. Two languages were chosen to add persistence: Smalltalk and Modula-3.

The Mneme object store was intended to support a variety of existing persistent and database programming languages and therefore to support a variety of type, inheritance, and invocation mechanisms. Thus, Mneme provides only a very primitive object model. All objects reside in Mneme files and the object identifiers stored in an object always refer to objects in the same file.

Persistent Smalltalk provides transparent orthogonal persistence to applications and rely on object faulting. The implementation was carried out by modifying the Smalltalk virtual machine to provide the illusion of a large persistent heap. When the modified virtual machine detects an attempt to access a non-resident object it raises an object fault that causes the required object to be mapped.

2.5.3 Persistent systems

Since 1978 a large number of researchers have constructed systems which support persistence. However implementing the abstraction of persistent data at the programming language level suffers from two drawbacks. The first of these is that the host operating system was not designed to support persistence; therefore the operating system interface does not usually provide abstractions sympathetic to a persistent language implementation. The consequence of this is that the language designer is usually forced to implement a persistent abstract machine above the operating system abstractions, with a corresponding loss of efficiency.

The second problem with this approach is that every persistent language implements its own persistent abstract machine duplicating much of the functionality found inside the operating system and other language implementations. Often these different implementations are entirely incompatible with each other, prohibiting interactions between programs written in different languages. Therefore, the abstraction of persistence should be implemented by the operating system. Such an approach to operating system design could be as revolutionary as virtual memory in terms of advantages to user level applications.

The four major components of an operating system are memory management, file system, input-output and process management. The nature of these four
components is different in persistent systems. In a persistent system, the functionality of the file system and memory management are replaced by the persistent store (Dasgupta et al., 1991). Therefore, an operating system designed to support persistence will have a different structure from a conventional operating system and will provide a different set of facilities.

The principal requirements of such an operating system can be summarized as follows:

The major requirement is support for persistent objects as the basic abstraction. Persistent objects consist of data and relationships with other persistent objects; the system must therefore provide a mechanism for supporting the creation and maintenance of these objects and relationships. This mechanism should be based upon a uniform addressing scheme used by all processes to access objects. That is, all processes share a single logical address space. This is essential if orthogonal persistence property is aimed.

Processes must be integrated with the object space in such a way that process state is itself contained within persistent objects. The importance of this is that processes themselves become persistent.

Although the persistent store is uniform, there is still a requirement to be able to restrict access to objects for the same reasons that file systems contain access control mechanisms. Any operating system supporting persistence must therefore provide some protection mechanism.

An operating system that provides these facilities is a "persistent operating system".

2.5.3.1 Stability

In a persistent system there may be arbitrary cross references between objects and thus the loss of a single object can result in total system failure. In this sense the problem of recovery within a persistent store is much more closely related to recovery in database systems.

Early persistent stores such as POMS and the CPOMS were constructed using conventional file systems with no special features (Cockshot et al., 1984). Since the underlying file systems offered no explicit support for persistence, and for stability in particular, techniques similar to those developed for database systems were used. The persistent store was implemented as a series of databases against which a program could apply transactions. As described above, such databases corresponded to individual files as provided by the operating system.
2.5.3.2 Address translation

Two address spaces are managed in persistent systems: a local process address space in which objects may be directly accessed by machine instructions and a persistent address space; objects should be transparently moved from one to the other on demand. This requires software address translation between local address spaces and the persistent address space; this software address translation can never be made as efficient as hardware address translation. However, the impact of the cost of address translation in persistent systems is not clear due to the lack of sufficient measurement. Another approach is to utilize paging mechanisms which are more efficient because they make use of hardware address translation such as Cricket object store (Shekita and Zwilling, 1990).

Three major ways of software address translation is defined in the literature (Vaughan and Dearle, 1992):

- Dynamically translate from PID to a virtual address on each dereference.
- Make an object’s virtual address coincident with its persistent identifier.
- Perform a once only translation from a persistent identifier to virtual address, overwriting the copy of the persistent identifier in the virtual address space with a virtual address so that all subsequent dereferences incur no translation penalty. This option is known as pointer swizzling.

2.5.3.3 Process model and protection

File level protection is of little use in a persistent system where all data is represented as objects directly addressable by processes. Therefore, all protection must be provided at the process level.

Originally, operating systems for hardware platforms supporting paged virtual memory, provided heavy weight processes, each running in its own address space and associated with a particular user. Such processes communicate with each other using various IPC mechanisms ranging from stream based systems (pipes and sockets) to shared memory and signals. More recent operating systems, have added support for lightweight processes called threads, which operate within a single shared address space. All threads have access to the entire address space and it is therefore possible for one thread to corrupt the data of another. When a thread requests a new segment to be mapped into its address space it immediately becomes accessible by all other threads sharing that address space.
There appear to be two different problems associated with process models of existing operating systems. The first of these is support for a single shared address space; this is solved by the light weight thread model. The second problem is support for protection at the process level; this is a more difficult problem and there are four different solutions:

- The use of type secure languages and trusted system components such as compilers which strictly enforce protection rules.

- To provide store level protection. However, store-level protection requires some architectural support.

- To implement multiple persistent address spaces. Adopting this approach gives a coarse grain of protection. Processes either have total access to the address space or none at all.

- To associate a page protection list with each thread; in such a scheme, the page protection map is changed on a context switch. In this manner, the threads would share a single address space but may have their access restricted on a per page basis. This would result in performance penalties on machines with virtually addressed caches and would increase the overhead of a thread context switch but it provides finer grain control over accesses.

2.5.3.4 Operating system kernel

Traditionally the operating system kernel is almost entirely composed of temporary data structures. These data structures have been regarded as structures for which it makes no sense to reason about their persistence. Kernel instances have been regarded as conceptually immortal and the bootstrap sequence has evolved to recover from those cases when this assumption is invalid. The benefits of persistence for applications programming are equally applicable to kernels. If the kernel is itself made persistent, then utilities like those mentioned above would require no special code in order to preserve their states between system invocations.

The kernel is the protector of process state; this state resides in the saved register contents for a process and in other kernel structures such as scheduler queues, open file descriptors and network connection structures. The information is held within the kernel primarily for security reasons. If processes are to be persistent objects, it is essential that the execution state of a process reside within the persistent store. If the kernel is itself persistent, that is all the data structures
maintained by the kernel are persistent, then the process state information may be held within the kernel and be persistent.

There are two goals in the operating system persistent store design: It should be able to stabilize the kernel-state of an entity independently from that of other entities. Secondary goals are that it should be efficient, since it has the ability to affect the performance of the entire system, and that it should be not intrude on code that uses it; writing an operating system kernel is already difficult enough.

2.5.4 Persistent store

Persistent store, an essential feature of a persistent programming environment or object-oriented database is a store capable of supporting complex objects. A common feature of persistent systems is that the persistent stores supporting them are both resilient and stable. Stability is the ability of a system to be consistently checkpointed on a secure medium so that computation may resume from that point at some future time. If a system is resilient then it can safely resume computation after an unexpected system crash such as a power failure. It is possible to have stability without resilience. For example, with some programming systems, state may be preserved (in a file) using a save command but consistency may be lost in the event of a system crash. Resiliency requires that the persistent store evolves from one consistent state to another atomically. That is, in the event of a system failure, all the changes are either recorded or the system recovers to the previous stable state.

2.5.5 Examples of persistent operating systems

Several notable examples of persistent operating systems are briefly discussed to provide a basis for EPS. Dearle et al, 2000 gives a comparison and general discussion of persistent operating systems.

2.5.5.1 Grasshopper

The Grasshopper operating system provides explicit support for orthogonal persistence (Dearle et al., 1993). The operating system kernel itself is persistent. In Grasshopper, many aspects of persistence are implemented by user-level software. Nevertheless, there is a small amount of data that must remain in the kernel in order to limit the damage caused by malicious or malfunctioning programs.

Grasshopper provides three basic abstractions: containers, loci, and capabilities. Containers are the single abstraction of data storage and access. They are passive,
persistent entities that provide a means for storing and accessing data. Loci are an abstraction of sequential execution. Each locus is basically a set of registers and some other system-related data such as priority and resource usage. Grasshopper is an object-thread-based system. This means that a locus need not be tied to its host container for its entire lifetime. Instead, it may move to another container by invoking it. Grasshopper uses segregated capabilities - they are system-protected entities that may only be accessed through system calls. In contrast to password capability systems, systems using segregated capabilities can keep a record of exact references to a particular entity, thus allowing deletion of unreferenced entities and any kernel data structures associated with them.

In addition to the three basic abstractions, Grasshopper provides a facility, called mapping, to share data and code extremely efficiently by allowing parts of one container to become directly accessible in another container. Mapping allows, for example, instances of abstract data types to share code rather than having their own separate copy. Mapping in Grasshopper differs from similar features in other systems such as Mach and Chorus in a number of important ways (Habert and Mosseri, 1990). First, mappings may overlap each other - the last mapping made at a particular address overrides any others. Second, mappings are not restricted to one level. Third, in addition to the usual case where a mapping is perceived by all loci in a container, a mapping may be made private to one particular locus. Such mappings are called locus-private mappings and take precedence over any global mappings. This technique both simplifies multi-threaded programming and provides a useful security mechanism unavailable in conventional systems.

EPS and Grasshopper share several similarities; both use capabilities for security, and object sharing is based on levels of access. On the other hand, EPS and Grasshopper have entirely different system models. While Grasshopper is based on object-thread abstraction, EPS has a recursive definition of a system with two types of persistent objects in two states. Moreover in contrast to the orthogonal persistency in Grasshopper, EPS has a non-orthogonal persistency support.

2.5.5.2 SOS

SOS is an object oriented system (Shapiro, 1991, Shapiro, 1990, Shapiro et al, 1989a, Shapiro et al 1989b). It is prototyped on top of UNIX.

SOS is a research program to build a distributed operating system, where all interfaces and communication are based on objects. SOS provides support for arbitrary user-defined objects, including object creation, destruction, migration, etc.
The Object Storage Service (OSS) manages the physical storage of typed and composite objects on disks in a generic way, and handles generic aspects of object persistency. Integrated with the migration mechanism, it gives the illusion of a single level store. Once stored, objects become permanent; their store representation can never be deleted. To be permanent, an object must be of type permObject, from which can be derived user-defined types. A permanent object can be composed of several segments. In order to be taken in account by the storage service, those segments must be referenced by special permanent pointers.

SOS is built using its own mechanism: all the SOS system services are implemented as fragmented objects with local proxy interfaces. Basic SOS components are Acquaintance service, Communication service, Storage service and the Name service.

Applications on SOS are designed according to the proxy principle (Shapiro, 1986). Services are composed of three kinds of elementary objects: servers, proxies and providers. The server is an object which is able to serve requests. The proxy is a local elementary object, which represents the service. Each client which wants to access a service, must have a proxy of this service in its context. The provider is in charge of providing proxies on client requests.

2.5.5.3 Mungi

Mungi is an experimental persistent operating system developed in The University of New South Wales (Heiser, 1998).

The design of Mungi is based on the following principles: A single, flat virtual address space, orthogonal persistence, and a strong but unintrusive protection model.

The single address space incorporates all memory objects on all nodes in a distributed system. There are no user-visible memory hierarchies, no file system. Any memory object is potentially persistent, i.e. can outlive its creator process. All objects are uniquely identified by their virtual address, which is a 64-bit number. Any object is potentially accessible by any process, provided the process is authorised to do so. Sharing is trivially achieved by exchanging addresses (and passwords, if required).

As there is no file system, all the secondary memory in the system is nothing but paging store. The address of an object does per se not give any indication on where the object is located, and there is really no such concept as "location" at the user
level. If a process references a non-resident object, the kernel will obtain a copy and store it in local primary memory. This is not different from paging in traditional virtual memory systems, except that a page may be obtained from disk or across the network.

Memory is allocated in (page aligned) chunks called "objects". An object is the unit of protection. No other structure on objects is assumed by the system, but higher software layers are free to impose structure.

Protection is based upon password capabilities, address-password pairs. Any holder of a valid capability to an object can access that object. Capabilities can be passed around freely, they are not system objects, but protected by sparsity. Objects do not need to be explicitly attached to a process: if the user deposits their capabilities in a system-maintained data structure, the system will transparently attach the object once it is accessed. This allows execution of programs that have no knowledge of the operation of the protection system.

A few practical points about design are:

- There are, of course, directory services to map UNIX-style path names onto object addresses. These are, however, user-level services, the system doesn't care about them.

- While all objects are persistent in principle, any new object created is entered into a "kill" list. When the process exits, the kernel deallocates all objects pointed to by this list. The process can at any time remove any object from this list and thus make it truly persistent.

- While all objects are globally visible in principle, objects are not necessarily entered immediately into the appropriate system data structures to make them globally known. This is determined by an object attribute, called "sharable": if this is not set at object creation time, the object is not guaranteed to be globally known. The object's owner can set this bit at any time and thus guarantee global accessibility of the object. Usage of this attribute allows fast allocation and deallocation of objects that are unlikely to be shared.

Mungi supports a single address space shared by all objects. EPS, on the other hand, applies principle of encapsulation and every object is under control of one active object. Contrary to Mungi, in EPS object sharing is possible through multiple copies and central synchronization.
2.5.5.4 Opal

The Opal project is exploring a new operating system structure, tuned to the needs of complex applications, such as CAD/CAM, where a number of cooperating programs manipulate a large shared persistent database of objects (Chase, 1995). In Opal, all code and data exists with in a single, huge, shared address space. The single address space enhances sharing and cooperation, because addresses have a unique (for all time) interpretation. Thus, pointer-based data structures can be directly communicated and shared between programs at any time, and can be stored directly on secondary storage without the need for translation.

Protection in Opal is independent of the single address space; each Opal thread executes within a protection domain that defines which virtual pages it has the right to access. The rights to access a page can be easily transmitted from one process to another. The result is a much more flexible protection structure, permitting different (and dynamically changing) protection options depending on the trust relationship between cooperating parties. Researchers believe that this organization can improve both the structure and performance of complex, cooperating applications.

An Opal prototype has been built for the DEC Alpha platform on top of the Mach operating system.

2.6 Comparison of EPS to Other Systems

In this section, we tried to emphasize the differences of EPS with other systems and research efforts briefly. As a conclusion, similarities, differences and the heritage of related research shall be summarized.

EPS philosophy is quite different from other extensible systems, since most of these systems have concentrated on the modification of core operating system services, whereas EPS concentrates on extending the existing capabilities of system without modifying the basic characteristics, and suggests a new model for system extension by user level modules.

EPS, as its name suggests, applies the principles of a persistent model. While this model does not support an orthogonal persistency as PS-Algol does, it uses many of the former research data as requirements and design input for this new system. By not supporting a pure orthogonality, we were able to include a distributed transaction capability to EPS, which allows the programmer to realize a transaction on several objects at the same time.
We addressed some of the uncovered issues by other persistent systems, such as problems in capability based naming and protection.

Majority of the research studies are suggesting their own system model, and EPS is no exception. The EPS system model based on persistent objects notion is unique, and its simplicity and uniformity is superior in representing the real world than other experimental system models.
3. EPS REQUIREMENTS

EPS is aimed to be a system of easy programming and use, and is supposed to work on existing architectures and operating systems. The system shall be modular, extensible, portable, support persistent objects and give freedom of choice to programmers in the controlling the system behaviour. In this section, requirements of such a system are explained.

3.1 EPS General Architecture Concept

EPS shall have a multi-tier architecture (figure 3.1) that can be extended naturally based on the needs. The core services such as object loading will be provided by the kernel. Object naming, protection and synchronization will be supplied by the third level. Active objects, objects with data and operations, are extensions to the system is the next level in the hierarchy.

![Diagram of EPS system concept]

*Figure 3.1 EPS system concept*

On the highest level, user processes reside where they can receive services from the lower levels. Thus the system services shall be realized by separate server processes at user level, in other words as active objects. Programmers will be able to extend the capabilities of the system by writing new server processes. While this is not a new concept, the ease of developing new servers and facilities for persistency and inter-object communication will be the differentiating factors.
This modular system shall be based on an existing UNIX implementation. This choice of platform will make the new system easy to adapt and use by the already existing programmers experienced on UNIX. On the other hand it will enable us to focus on the area of research instead of investing effort on details like device drivers. The system language is chosen to be the C programming language because of its natural integration to UNIX operating system.

Satisfaction of architectural requirements are explained in section 5.

3.2 System Services

Current operating systems do not provide an appropriate platform for building persistent environments. This leads to the conclusion that an operating system especially designed to support persistence is required. Using current operating systems, the implementers of a persistent environment must manage the address translation tasks. A persistent operating system will provide an abstraction consistent with our requirements as a fundamental building block.

EPS, being a persistent system would be capable of providing all the functionality of traditional systems. The research issues are those same issues which compromise the implementation of persistent systems when conventional operating systems are used as a platform, namely: addressing, resilience, process management and protection. On the other hand, EPS will support programmers and designers in new application development, through the facilities for persistent object support and inter-object communication capability which will enable programmers to use simple primitives for requests from other objects. Persistent objects can be passive - that is data only or active-meaning to include both data and methods thus ability to change state and satisfy other objects' requests. EPS will transparently manage the loading of both types of objects automatically meaning that an active object, which in fact is a kind of server process, can be loaded on demand without any programmer or operator intervention.

The synchronization of the data access will be handled by one of the EPS servers, and communication among objects, often a tricky issue in the development of system extensions will be handled by high level system primitives. These service are explained in sections 5.2 and 5.4

Another system service shall be a special compiler that will process the persistent flavor of the C language, EPS-C. EPS-C preprocessor shall also be able to handle
the program code written in standard C programming language. EPS-C is explained in section 4.

3.3 Single Node Implementation

EPS shall be designed and implemented as a single node system. Although recent trends in operating systems research is directed toward parallel and distributed systems, since primary research goals of the EPS project does not include the investigation of multi-processor utilization nor networked environments, it is logical to concentrate on the research area.

A single node implementation will enable us to utilise centralized solutions for the planned system, such as security and synchronization.

3.4 Hardware and Software Platform Requirements

The major hardware constraint is that the EPS should run on conventional architectures. Effects of this constraint are that on most current architectures addresses are a maximum of 32 bits long, there is no hardware support for object protection, and the only memory management hardware available is based on fixed sized pages. Another architectural constraint is that there shall be no need for a protected storage area because a constraint on this feature will restrict the number of capabilities.

The hardware of choice with the above mentioned properties is standard PC hardware with Intel processor family. The choice has no specific considerations other than the wide availability of this type of systems. Moreover the researcher believes that the type of hardware platform is not-relevant to the results of this research.

3.5 Programming Interface Requirements

Programming under EPS will not be very different than a conventional system. The well-known programming languages will be used with few but critical additions to them. This will enable programmers to develop new systems based on EPS with a short learning period. Consequently, this harmony with the existing systems will facilitate the widespread use of EPS. Programming environment is explained in section 4.
3.5.1 Object loading strategy

The object loading strategy which has an important impact on the overall performance has two possible choices: explicit loading or implicit loading. An explicit loading scheme is one in which the store provides a command to retrieve an object. This method is clumsy without language support. It is also inefficient since a check must be made on every access to see if the object is in physical memory. An implicit loading scheme uses memory management hardware to detect accesses to non-resident pages and does not intrude on code using the store. The disadvantage of using implicit loading is that it implies that stabilization must be performed on page-sized units since memory management hardware can only detect modification of pages rather than individual entities. This is not a problem in systems where the entire store is stabilized simultaneously, making page boundaries irrelevant.

In EPS, loading of persistent objects shall be handled by explicit function calls. While this looks like an extra task to programmer, we shall be aware of the fact that the programmer shall declare the binding between the persistent object and the database in any case if we do not want to have a single store system with orthogonal persistence. The decision of explicit binding, hence non-orthogonality property, is based on performance and usability concerns. Another fact is, most of the extra programming effort in non-persistent environments is spent in I/O operations and data conversions which are successfully addressed in non-orthogonal systems as well as orthogonal systems. In EPS, programmer will decide the data loading time. He will have the choice of loading data at the program start automatically or make an explicit call for loading. This gives flexibility to design programs for different purposes and characteristics. A static server program may benefit from loading all the parameters while starting, and it may be acceptable to have a delay on the activating phase of the server. On the other hand, a program which needs different data for different types of events may benefit from loading data dynamically when it is necessary. Implementation of object loading in EPS is explained in 5.2.

3.5.2 Type compatibility

There are two approaches in variable type checking. Static type checking is conducted during compilation, before program execution. Static checking allows assertions to be made and even proved about a computation before it is executed. Therefore, it provides a level of safety within the system. On the other hand, dynamic type checking which is conducted during program execution is sometimes
necessary for binding variables created after the compilation time of the program, and gives more flexibility to programmers.

EPS will support both approaches. EPS-C preprocessor does the type checking during the compilation phase. It is also possible to delay the type checking to the execution phase, to enable programmers develop code for currently non-existent persistent objects. In this case, preprocessor creates new objects at compile time. Persistent object type management is explained in section 4.

3.6 Security Requirements

There are two basic mechanisms to access objects: capabilities and access lists. Capabilities are non-replicable tickets which give authorization to their owner to access an object. Capabilities can be used to name, protect and define the operations on an object. General characteristics of capabilities are:

- If a process owns a capability, then it has the access right to the related object.
- Capabilities allow sharing of the objects.
- Capability should be non-replicable and non-decipherable. A system that lets users to create capabilities is useless.

Several types of capability mechanisms have been developed:

Encrypted sign capabilities: In this approach, a capability consists of two major parts; object id and access rights. A signature field derived from the previous two fields is appended as the trailer of the capability. Signature, id and access rights fields of a valid capability should be in correlation. A modification on the id or access rights without the modification of signature will invalidate the capability. To prevent users to find the relation among these three, capability fields are encrypted together. Encryption operation is conducted by the kernel which also owns the encryption key. Encrypted capabilities can be kept in unprotected memory area. Capability shall be decrypted by the kernel to check its validity.

LLNL (Lawrence Livermore National Laboratory) capability: This capability implementation is composed of server address, attributes, unique local identification and extra information fields (Donnelley, 1981). This last field stores extra data to prevent the unique local identification from being copied. This approach does not include a solution for capability transfer.

Amoeba capability: A capability in Amoeba is 128 bit long and consists of four fields; server port, object number, rights, control field (Mullender et al, 1990). When a new
object is created by a server process, the capability for this object is returned to the client process.

Capabilities are managed by user processes and protected by the encryption method. When a new capability is created the rights field is joined by a 48 bit random number and the result is put into the control field. This data structure is kept in the server process. When there is a request for an object, the capability sent by the client and the original capability is compared. Capabilities are stored in directories, which are also objects.

Signature Capability: This capability consists of two parts; a unique object id and a signature. This signature field is very long in order to prevent others to estimate it. In this approach, since a capability is produced especially for a client an encryption operation is not necessary. The major problem with this technique is that the sharing of objects is difficult. For every client of an object a new capability has to be created.

Very little work has been done on the removal of already distributed capabilities. Amoeba system with the most advanced features can remove a capability by modifying the random number of the capability, but partially removing the capabilities is not possible.

The EPS system shall provide the following capability features:

- It shall support resource sharing. By passing a capability to a new object the access rights of the related object shall be transferred.

- While capabilities are transferred, it shall be possible to restrict the access rights. A client shall be able to transfer a restricted copy of a capability.

- Formerly distributed capabilities shall be removed from some of the clients or some of the rights in a capability shall be taken back.

- It shall be possible to remove a formerly given capability completely.

The mechanism to support these features shall have the following architectural constraint: There shall be no need for a protected storage area which is a hardware support for access restriction, since a constraint on this feature will restrict the number of objects.

Security mechanism in EPS is supplied by Naming and Protection Server, and it is explained in section 5.4.
3.7 Reliability Requirements

One of the major issues in data storage systems is preservation of consistency. While there is no problem in normal operation of programs, an abnormal termination in the processing may result in partial update of data. Database management systems solve this problem through transaction based programming model. Modifications in one transaction are either committed or rolled back at the end of the transaction ensuring data integrity in the system.

EPS shall supply a similar system which will allow programmers to group related data modification operations together and decide to proceed or to take back the system to its prior state. Each modification operation in EPS shall be implicitly a part of a transaction. Programmers will declare state transition decisions through explicit commit and rollback commands. Transaction management and resilience mechanisms are explained in section 5.

3.8 Performance Expectations

It is difficult to define numerical goals for the performance of a complex experimental system. Since the primary focus of this research is not performance related, we rather have goals for the type of data which will be supported with an acceptable performance. EPS is not expected to have comparable performance for conventional database type data management, it will rather have better performance for complex information representations such as CAD/CAM data. On the other hand small pieces of data shall be supported appropriately. Consequently EPS shall support small and medium size grained objects with complex relations such as linked data structures and pointers with an acceptable performance.

3.9 Persistent Store

The first persistent languages such as PS-Algol and Smalltalk have one persistent store for one program. Every data object in the program is saved automatically in this single store, and loaded as a whole at the initialization phase of the program. This approach is known as orthogonal persistence where the programmer need not know or declare whether a data object is to persist or not.

While the notion of transparency to the programmer may seem favorable, this type of approach has caused certain problems; for large chunks of data loading/saving of store is very slow, or even impossible if the size of the store has grown larger than
the available virtual memory. Moreover, a single store approach makes it necessary
to use type-safe languages because a run-time memory protection is not possible.
In single store approach, garbage collection becomes a major concern. Since data
for temporary storage is automatically saved in a single store orthogonal system,
determination of useless data and removing them from the store is a mandatory
task. Transfer of data from one persistent store to another raises certain difficulties
such as; determination of large number data objects to transfer, and keeping track of
closure property, that is objects with links to each other have to be copied together.
A single store makes it difficult for the programmer in reasoning about the program
and bugs. The effects of faulty code can easily destroy many parts of the store
because there is no intra store limits and protection.
A generic architecture for such a persistent store shall address a list of key issues:

- Identification of persistent objects reachability. EPS assumes that object’s
management for allocation and security are handled by owner objects, either
active or client, and there is no garbage collection service, so a persistent object
always lives unless it is explicitly destroyed by somebody. Therefore this issue is
not addressed in EPS.

- Properties of objects and their interconnections; the store must be able to
coherently store non-pointer and pointer data (i.e. inter-object references). EPS
supports both pointer and non-pointer data.

- Scale of the object store; the object store should provide a conceptually infinite
store. Virtual memory system and file systems will simulate such an
environment.

- Provision of stability; the system should be able recover from unexpected
failures and return the system to a recent self-consistent state. This is similar to
the reliability requirements of OODBs. EPS uses shadow object and shadow
files technique to prevent an object from getting into an inconsistent state. Multi
object consistency however is risked by unexpected system failures.
Implementation of these techniques are explained in section 5.3 and 5.5.

3.10 Supported Application Types

EPS will be a general purpose system with special features of persistency and
extensibility. The built-in capabilities of the system will make the system favourable
for the developers of different types of application.
The primary type of application can be server systems for different purposes. With extensibility capabilities of EPS it is possible to tailor the system for certain types of applications and usage. It can also be used for applications dealing with complex information representations such as CAD/CAM systems and artificial intelligence with its support to persistent representation of complex data types.
4. PROGRAMMING IN EPS

Programming in EPS is quite similar to programming under UNIX with some exceptions. The first difference is the support for persistent data. Another important aspect is the transparent loading of server processes (active objects) as required. The third major difference is the support of an inter-object communication, which is easy to use yet powerful and flexible. The last difference is the support for active object development, which is easy and prevents programmer from making errors.

An EPS-C program is compiled by EPS-C preprocessor and the C compiler. Object code is linked to COL in addition to other required libraries. A typical EPS-C program starts with the declaration of persistent variables. One of the first few lines of executable code shall include a call to EposInit() function, which will initialize the persistent process environment by loading the persistent objects. Once the intended operations are conducted, the program is exited by calling the EpsClose function which will synchronize data and release resources.

4.1 EPS-C Programming Language

The primary programming language is the C programming language with some extensions. This extended syntax is called as EPS-C to distinguish it from the standard C programming language. This implementation will allow application developers to use the C programming language alone if they prefer. A sample program code is shown in figure 4.1, and more example EPS-C programs are available in Appendix-C.

Main enhancement to C language is the addition of a reserved keyword, "$", to declare the persistent nature of variables and some standard EPS functions as a library.

The loading of persistent objects are handled either by explicit function calls or the necessary statements are inserted into program code by EPS-C preprocessor. While explicit function calls look like an extra task to programmer, we should be aware of the fact that the programmer shall declare the binding between the persistent object and the database in any case. These two options let programmers
decide on the loading time and give flexibility to design programs for different purposes and with different characteristics.

```c
#include "epsotypes.h"
#include "epsmsg.h"
#include "epslib.h"
#include "epscob.h"

$FileType pName;
$TableType pPrTable;
capabilityType cName;
EpsResult result;
char ad[30];
char parameter[1000];
SessionType session;

int main()
{
  /* 1*/ result = EpsInit($session, "btree", cName);
  if (result.code) exit(1);

  /* 2*/ LoadPersistentObject($session, parameter, cName, ACMRW);

  scanf("%s",ad);
  result = AdEkle(&pName,ad);
  if (result.No == 0)
    /* 3*/ ObjectCommit($session, "pName", OPCOR);
  Else
    /* 4*/ ObjectRollback($session, "pName", OPROR)
  }
/* 5*/ EpsClose($session, OPROR);
```

Figure 4.1 Adding names to a b-tree

If we examine the program in figure 4.1; at step 1 (line with /* 1*/), EpsInit function is called to initialize the EPS session for program "btree" with "cName" capability. Step 2, loads a persistent object "pName" to "session" with "cName" capability. Step 3 and 4 either saves or rollbacks modified "pName" object. Step 5 closes the local EPS session and rollbacks transaction.

4.1.1 Persistent variable management

Since we do not assume all variables to be persistent in EPS, an extra notation was necessary for the declaration of persistent variables. We chose to use a single character "$" at the start of a line of code to declare the persistency property of a data object as seen in the example below.

```
Siint testVar;
```

Persistent variable types can be any simple C variable type or structures defined as types. Structures should be one level structures consisting of simple C types, and
every type definition of persistent variable shall be stored in a separate file with the same name. Type definition file is defined by programmer during program development. As a part of EPS design philosophy there are no specific types file formats used. Instead, already available type declaration capability is used for the type checking facility of passive persistent objects. Persistent type handling is explained in EPS-C preprocessor section.

For a complex data structure such as a linked list or linked tree, declaration of the root pointer variable as a persistent object is sufficient. In the loading or saving phases, all the addresses reachable by the persistent root will be processed, since it is loaded in contiguous memory.

Persistent variables are considered to have global scope in the program they are defined. Hence it is not possible to have to persistent variable declarations in the same program even though they are in different scopes with regard to C programming language standards.

4.1.2 Mandatory operations

Every EPS-C program shall start with the call of EpsInit function. This function will first load the program table. Program table is a predefined type of persistent object (structure is explained in section 5.3) used by every EPS program. This persistent object is used to store state data of a program. It can also be used to store small grained simple type data.

Every EPS-C program shall be terminated by a call to EpsClose function, which closes and frees the session structure together with loaded objects. The modified objects are either committed or rolledback depending on the programmer request.

4.2 Active Object Development

Active objects development is supported by an active object library. This library implements a main server loop to receive and handle client requests. This library also includes standard commit and rollback functions. Active objects are part of EPS and, in a sense, they are client objects with regard to their relationship with NPS and OBS.

Programmer shall follow 4 steps to complete the development of a functional server ready to run:

1. Write a function for each message type server accepts. This function shall satisfy the demand of client sending this specific message.
2. Fill the entries of a static function pointer array with these functions:

```c
int (* f[MAXFUNCTION]) (char *, int) = {AobCommit, AobRollback, NULL, NULL, NULL, NULL, AddItem, GetItem};
```

In this example, AddItem and GetItem functions are developed by the application programmer. The first six locations of the array is used by AobServer function, and shall be left unchanged.

3. Write a main function that initializes an EPS session (EpsInit) and then calls AobServer function (AobServer(f)):

```c
...
EpsInit(&session,"warehouse",ICapF);
AobServer(f);
EpsClose(&session,COMMIT);
...
```

4. Compile and link program with active object library (epsaob.c) and client object library (epscoab.c)

Now we have an active object which responds to request messages from client objects by executing the corresponding server function. An example active object is available in Appendix C:

### 4.3 EPS-C Preprocessor

EPS-C is implemented through a preprocessor which is used together with the standard C compiler and client object library which will be linked to every EPS program. After the EPS-C preprocessing phase, the C preprocessor and compiler run to process the program code. EPS-C program files have ".epsc" extension, after EPS-C preprocessing, a source code file with ".c" extension is produced to be manipulated with standard C compiler and linker.

On the first pass, the preprocessor parses the program code and replaces the persistent declarations with normal C variable declarations. By the end of first pass, a persistent object table is filled, in which persistent variable types, sizes and counts are recorded. For complex persistent types such as structures, the type and size of the persistent variable is determined by parsing its related include file content. In EPS every complex persistent type shall be declared in a separate include (.h) file with that complex type's name. An example will be useful to clarify the situation:

For a variable declared in program:
$TestType testVar;

An include file “TestType.h” should be supplied by programmer. The include file’s content is:

    Typedef struct
    {
        char key[20];
        char data[20];
    }TestType;

EPS-C preprocessor and runtime libraries use information in this include file to handle persistent variable properly.

Instead of defining a new notation for persistent object type declaration, we chose to extend the meaning of available notation. This is a good example of EPS philosophy, keeping the system simple and similar to existing ones, and at the same time, enhancing it with new capabilities and a new programming model.

On the second pass, using the persistent object table filled in the first pass, it inserts function calls to load persistent variables into program source, if requested as a preprocessor option.

The preprocessor can accept option “-l” to control the loading time of persistent data. With this option, all persistent objects are loaded automatically at the start up of a program. On the other hand, if this compiler option is not used, automatic persistent object loading is disabled and programmer can control object loading time at run time through using LoadPersistentObject function.

Option “-cc” directs the preprocessor to create program capability file, if it was not created before. More information on capabilities is available in section 5.7.

Option “-co” directs the preprocessor to create objects on disk, if it was not created before.

An example preprocessor syntax:

    epscpp warehouse -l

4.3.1 Compilation and linking

EPS-C is implemented through a preprocessor which is used together with the standart C compiler and a client object library (COL) linked to every EPS program.
After the EPS-C preprocessing phase, the C preprocessor and compiler run to process the program code.

In the linking phase, COL is linked to every EPS-C program. This library not only includes the application programming interface functions but also contains functions to handle persistent variables.

4.4 Application Programmer's Interface

Application programming interface consists of a number of system primitives which can be used in application programs. These primitives are implemented in Client Object Library and help to hide the complexities of EPS.

4.4.1 Session primitives

Session primitives are functions related to session management. Input/output parameters can be found in table 4.1.

```
ResultType EpsInit(SessionType *, char prgName[FILenamesize], FILE* ICapFile): Allocates memory for the heap, loads program table and creates typecheck and allocation lists.
```

```
ResultType EpsClose(SessionType *session, int operation): Closes the session and commits or rollbacks the objects in session.
```

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>In Parameters</th>
<th>Out Parameters</th>
<th>In/Out Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EpsInit</td>
<td>PrgName</td>
<td></td>
<td>session</td>
</tr>
<tr>
<td>EpsClose</td>
<td>Session, operation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4.2 General object primitives

General object primitives are functions related to object management. Input/output parameters can be found in table 4.2.

```
ResultType LoadPersistentObject(SessionType *session, char** pVar,
CapabilityType caV,int acM): Load the object to the heap, bind it to the local variable.
```

```
ResultType SavePersistentObject(SessionType *session, char** pVar,
CapabilityType caV,int operation): Commits the modifications of this program
```
session except the program table. Also sends commit signals to active objects which were requested to make modifications to their data.

`ReturnType ReleasePersistentObject(SessionType *session, char** pVar, CapabilityType caV, int operation);` Releases the heap and commits/rollbacks the program table;

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>In Parameters</th>
<th>Out Parameters</th>
<th>In/Out Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadPersistentObject</td>
<td>Capability, access mode</td>
<td></td>
<td>Session, parameter,</td>
</tr>
<tr>
<td>SavePersistentObject</td>
<td>Parameter, capability, operation</td>
<td></td>
<td>Session</td>
</tr>
<tr>
<td>ReleasePersistentObject</td>
<td>Parameter, capability, operation</td>
<td></td>
<td>Session</td>
</tr>
<tr>
<td>EpsCommit</td>
<td>Operation</td>
<td></td>
<td>Session</td>
</tr>
<tr>
<td>EpsRollback</td>
<td>Operation</td>
<td></td>
<td>Session</td>
</tr>
<tr>
<td>ObjectCommit</td>
<td>Object Name</td>
<td></td>
<td>Session</td>
</tr>
<tr>
<td>ObjectRollback</td>
<td>Object Name</td>
<td></td>
<td>Session</td>
</tr>
<tr>
<td>CheckObjectByID</td>
<td>Session, PID</td>
<td>MemoryObject</td>
<td></td>
</tr>
<tr>
<td>CheckObjectByName</td>
<td>Session, PID</td>
<td>MemoryObject</td>
<td></td>
</tr>
<tr>
<td>Pmalloc</td>
<td>Session, objectName, count</td>
<td>Pointer to new allocation</td>
<td></td>
</tr>
<tr>
<td>Pfree</td>
<td>Session, objectName</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`ReturnType EpsCommit(SessionType *session, int operation):` Commits the modifications of this program session. Also sends commit signals to active objects, which requested to have modifications made to their data.

`ReturnType EpsRollback(SessionType *session, int operation):` Rollbacks the modifications of this program session. Also sends rollback signals to active objects, which requested to have modifications made to their data.
**ResultType ObjectCommit(SessionType *, char *objectName[NAMESIZE], int operation):** Commits the modifications of an object. Also sends commit signals to active objects which requested to have modifications made to their data.

**ResultType ObjectRollback(SessionType *, char *objectName[NAMESIZE], int operation):** Rollbacks modifications of an object. Also sends rollback signals to active objects which were requested to make modifications to their data.

**MemoryObject *CheckObjectByID(SessionType *session, PIDType pid):** Find an object in the session by its Persistent ID (PID).

**MemoryObject *CheckObjectByName(SessionType *session, char *objectName[NAMESIZE]):** Find an object in the session by its name.

**char *PMalloc(SessionType *session, char *objectName[NAMESIZE], int count):** Allocate heap memory for persistent object.

**void PFree(SessionType *session, char *objectName[NAMESIZE]):** Free heap memory of persistent object.

### 4.4.3 Table object primitives

Table object primitives are functions related to table object management. Input/output parameters can be found in table 4.3.

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>In Parameters</th>
<th>Out Parameters</th>
<th>In/Out Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>TableRead</td>
<td>Session, tableName, key</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>TableWrite</td>
<td>Session, tableName, key, data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TableDelete</td>
<td>Session, tableName, key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TableAppend</td>
<td>Session, tableName, key, data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**int TableRead(SessionType *session, char *tableName[NAMESIZE], char key[TABLEKEYSIZE], char *data[TABLEDATASIZE]):** Retrieves an object from a table object.
int TableWrite(SessionType *session, char tableName[NAMESIZE], char key[TABLEKEYSIZE], char data[TABLEDATASIZE]): Adds an object to a table object.

int TableDelete(SessionType *session, char tableName[NAMESIZE], char key[TABLEKEYSIZE]): Removes an object from a table object.

int TableAppend(SessionType *session, char tableName[NAMESIZE], char key[TABLEKEYSIZE], char data[TABLEDATASIZE]): Append new rows to the table object.

4.4.4 Communication primitives

Communication primitives are functions related to inter-object communication. Input/output parameters can be found in table 4.4.

int SendMessage(PIDType objectId, PIDType senderObjId, int operation, CapabilityType capability, char *parameter, ResultType result): Sends a message to a client or active object or NPS (Naming and Protection Server). SendMessage is asynchronous, program does not have to wait for any result after SendMessage call.

int ReceiveMessage(PIDType objectId, PIDType *senderObjId, int *operation, CapabilityType *capability, char *parameter, ResultType *result): Receives a message from a client or active object or NPS. ReceiveMessage is synchronous, program has to wait for a result on ReceiveMessage call.

int Request(CapabilityType caV, int operation, char *parameter, ResultType *result): Requests an operation from an active object, waits for the answer and returns the EpsResult structure.

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>In Parameters</th>
<th>Out Parameters</th>
<th>In/Out Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendMessage</td>
<td>ObjectId, senderObjectId, operation, capability, parameter, result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReceiveMessage</td>
<td>ObjectId, senderObjectId,</td>
<td>operation,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>capability,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>parameter,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>result</td>
<td></td>
</tr>
<tr>
<td>Request</td>
<td>Capability, operation</td>
<td>result</td>
<td>parameter</td>
</tr>
</tbody>
</table>
4.4.5 Capability primitives

Capability primitives are functions related to capability management. Input/output parameters can be found in table 4.5. Additional functions that manage capability store are explained in section 5.7.1 – “Capability Handling”.

\[
\text{ReturnType RequestCapability(FILE* capF, char objectName[NAMESIZE], CapabilityType *ICap): Request a capability from Naming and Protection Server.}
\]

\[
\text{ReturnType AobCapabilityTransfer(char objectName[NAMESIZE], CapabilityType ICap, char limitation[10]): Transfers a capability to another client object.}
\]

\[
\text{ReturnType CancelRights(char objectName[NAMESIZE], CapabilityType ICap, char, char cancellation[10]): Limits or removes access permissions to a persistent object.}
\]

\[
\text{ReturnType CancelCapability(char objectName[NAMESIZE], CapabilityType ICap, char, char cancellation[10]): Cancel already distributed capability for all/group/other owner objects.}
\]

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>In Parameters</th>
<th>Out Parameters</th>
<th>In/Out Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequestCapability</td>
<td>Capability file, objectName</td>
<td>Capability</td>
<td></td>
</tr>
<tr>
<td>AobCapabilityTransfer</td>
<td>ObjectName, capability, limitation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CancelRights</td>
<td>ObjectName, capability, cancellation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CancelCapability</td>
<td>ObjectName, capability, cancellation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4.6 Active object primitives

Active object primitives are functions related to the implementation of active objects. Input/output parameters can be found in table 4.6.

\[
\text{int AobServer(int (*f[MAXFUNCTION]))(char*, int): Main active object function that implements a client request message poll and satisfy loop.}
\]

\[
\text{int AobCommit(char*, int): Commit an object in the active object session.}
\]

\[
\text{int AobRollback(char*, int): Rollback an object in the active object session.}
\]
<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>In Parameters</th>
<th>Out Parameters</th>
<th>In/Out Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>AobServer</td>
<td>Function Array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AobCommit</td>
<td>Parameter, operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AobRollback</td>
<td>Parameter, operation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5. EPS DESIGN

In this section, design of a new system satisfying the requirements explained in the previous sections is detailed. The design of the Extensible Persistent System is developed with certain constraints derived from the requirements. The new system is to be built on an existing operating system. Because of its availability, open source policy, and facilities for development, the Linux operating system is chosen as the platform.

5.1 EPS System Architecture

EPS has a multi-tier architecture (figure 5.1) that is designed to extend naturally based on the needs. The basic system is composed of three basic components; Naming and Protection Server (NPS); Object Server (OBS) and Client Object library (COL). An underlying messaging facility, inter-object communication interconnects these three modules. While these modules constitute the base system, further user

![EPS system layers](image)
needs can be satisfied by extending the system via active objects (AOB). Basic system algorithms and interactions among servers are explained in detail in Appendix-D. Other functionality is either built upon these basic algorithms or are very similar to them.

Different types of objects with specific representations in EPS are listed in table 5.1

<table>
<thead>
<tr>
<th>Description</th>
<th>File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPS program file</td>
<td>*.epsc</td>
</tr>
<tr>
<td>Preprocessed EPS program file</td>
<td>*.c</td>
</tr>
<tr>
<td>Object Type definition file</td>
<td>*.h</td>
</tr>
<tr>
<td>Capability store</td>
<td>*.cap</td>
</tr>
<tr>
<td>Object file on disk</td>
<td>*.dat</td>
</tr>
</tbody>
</table>

5.2 Inter-Object Communication System

For the communication between objects, a high level interface matching with the general requirements of the system is provided. Inter-object communication is the bond among different objects/modules in EPS (figure 5.2). The primitives are designed to support object to object synchronous (Request) and asynchronous (Send) communication, and the choice is left to the programmer.

<table>
<thead>
<tr>
<th>Message Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender Object</td>
<td>Persistent Identifier of the sender object.</td>
</tr>
<tr>
<td>Receiver Object</td>
<td>Persistent Identifier of the receiver object.</td>
</tr>
<tr>
<td>Operation</td>
<td>Requested operation</td>
</tr>
<tr>
<td>Capability</td>
<td>Capability ticket for the operation</td>
</tr>
<tr>
<td>Data</td>
<td>Any data to be transferred.</td>
</tr>
<tr>
<td>Result</td>
<td>Message result.</td>
</tr>
</tbody>
</table>
Message structure (table 5.2) is based on object communication paradigm hiding the IPC details from programmer. We believe both types of communications have their merits and can be used according to needs of the application. These primitives are explaines in detail in section 4. Inter-object communication subsystem is based on Unix IPC mechanisms.

Primitives of this high level interface are used both in systems development and application development, so they are available to application programmers as well.
5.3 Client Object Library

Client Object Library (COL) which is used by every EPS user program and active object has facilities to hide the underlying complexity of the system. It is linked to every program. Some of the system primitives are implemented fully or partially in the COL. The services of COL also include the local (in-process) implementation of synchronization and address translation facilities, which are transparent to the application programmer. The remote parts of the services are requested from EPS servers and active objects through the inter-object communication subsystem.

![Diagram showing the session architecture](image)

Figure 5.3 Session architecture

COL creates and manages a client session (figure 5.3), which is a term to define local management scope of EPS and also the storage area for loaded persistent objects. In this sense, it is the primary memory part of the EPS persistent store. When the client object is loaded, in other words a client process is activated, a new session is initialized in the process address space. Loaded persistent objects, are stored and handled in the client session without the intervention of programmer. These services are mainly: add a new object to the session, remove an object from session, find and object in the session, address translation, and commit/roll back object or objects in the session. The level of control over the persistent object management is a decision of the programmer. This control is explained in section 4.

Main data structures used in the Client Object Library, session and object are explained in Appendix A. Theoretically, COL is the part of EPS which needs to be ported to be used for the support of different programming languages.

A default persistent object called “program table” (figure 5.4) is available to every program, and the necessary functions to handle this persistent object are inserted automatically into the program code during the linking phase. Program table is a specific example of persistent table object, used in EPS implementation. Programmers can create and use new table objects in their applications. Program table object is specific to each program, and includes standard fields such as
'variable name - key', 'variable value - data'. There are extra functions to retrieve and modify the values in the program table, which are explained in section 4. This table object is planned to be used as the practical storage area for every program.

<table>
<thead>
<tr>
<th>key1</th>
<th>data1</th>
</tr>
</thead>
<tbody>
<tr>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>keyN</td>
<td>dataN</td>
</tr>
</tbody>
</table>

**Figure 5.4 Table object**

Session object is used to hold information on the current process such as the active object id, and loaded passive objects. Session object actually consists of two parallel linked lists (Figure 5.3). Parallel nodes of the session store the original and modified objects, that lets the system to realize commit and rollback requests.

### Table 5.3 Object structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersistentID</td>
<td>Persistent identifier of the object</td>
</tr>
<tr>
<td>ObjectName</td>
<td>Object Name</td>
</tr>
<tr>
<td>DataLength</td>
<td>Current length of data in object</td>
</tr>
<tr>
<td>MaxDataLength</td>
<td>Maximum length of data</td>
</tr>
<tr>
<td>ObjectSize</td>
<td>Total object size</td>
</tr>
<tr>
<td>Data</td>
<td>Address of data</td>
</tr>
<tr>
<td>ObjectStatus</td>
<td>Object status; Not modified, modified, deleted or active</td>
</tr>
<tr>
<td>CopyStatus</td>
<td>Shadow or original object</td>
</tr>
<tr>
<td>ShadowObject</td>
<td>Address of shadow or original object status</td>
</tr>
<tr>
<td>ObjectType</td>
<td>Simple or complex type name</td>
</tr>
<tr>
<td>DataFileName</td>
<td>Name of the data file on disk</td>
</tr>
<tr>
<td>Capability</td>
<td>Address of its related capability</td>
</tr>
<tr>
<td>ModifiedAt</td>
<td>Last modification time</td>
</tr>
<tr>
<td>ModifiedBy</td>
<td>Last modification by</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic redundancy code</td>
</tr>
<tr>
<td>PrevObject</td>
<td>Address of previous object in linked list</td>
</tr>
<tr>
<td>NextObject</td>
<td>Address of next object in linked list</td>
</tr>
</tbody>
</table>

Client Object Library also manages the local capability store, which is the storage area for previously obtained capabilities. When a request for an object is to be
made, client object first checks if it already has the capability for that object in its local capability store. If it is not available, then it requests this capability from NPS.

Object structure is the description of passive objects in the primary memory (table 5.3). This structure is the main reference point for a loaded passive object.

### 5.3.1 Client object library services

A list of services made available to programmers is listed in table 5.4. Primitives implementing these services are explained in section 4.

<table>
<thead>
<tr>
<th>Service Description</th>
<th>Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize the local environment</td>
<td>EpsInit</td>
</tr>
<tr>
<td>Close the local environment</td>
<td>EpsClose</td>
</tr>
<tr>
<td>Load a persistent object to the local environment</td>
<td>LoadPersistentObject</td>
</tr>
<tr>
<td>Release a loaded object from the local environment</td>
<td>ReleasePersistentObject</td>
</tr>
<tr>
<td>Commit the modifications on an object</td>
<td>ObjectCommit</td>
</tr>
<tr>
<td>Rollback the modifications on an object</td>
<td>ObjectRollback</td>
</tr>
<tr>
<td>Commit the modifications in the session</td>
<td>EpsCommit</td>
</tr>
<tr>
<td>Rollback the modifications in the session</td>
<td>EpsRollback</td>
</tr>
<tr>
<td>Read a data row from a table object</td>
<td>TableRead</td>
</tr>
<tr>
<td>Write a data row to a table object</td>
<td>TableWrite</td>
</tr>
<tr>
<td>Append a row to table</td>
<td>TableAppend</td>
</tr>
<tr>
<td>Delete a data row from a table object</td>
<td>TableDelete</td>
</tr>
<tr>
<td>Request a method from an active object</td>
<td>Request</td>
</tr>
<tr>
<td>Request a capability from NPS</td>
<td>RequestCapability</td>
</tr>
<tr>
<td>Find an object in the session</td>
<td>CheckObjectByLD, CheckObjectByName</td>
</tr>
<tr>
<td>Memory allocation and release for persistent objects</td>
<td>PMalloc, PFree</td>
</tr>
<tr>
<td>Send (async) and receive (sync) message to another EPS object (active object, NPS, OBS, client object)</td>
<td>SendMessage, ReceiveMessage</td>
</tr>
<tr>
<td>Request (sync) execution of an active object method</td>
<td>Request</td>
</tr>
</tbody>
</table>

### 5.3.2 Adress translation

Persistent objects loaded into the primary memory by the Object Server (OBS), and transferred to local memory via inter-object communication subsystem need one more crucial operation to be useful; conversion of persistent pointer addresses into local memory addresses. While data sections of persistent variables (basic or
structured) does not need any specific post-load operation, pointers in the variables
are converted into local pointer addresses.

Address conversion is conducted by COL and based upon the persistent object
type, and realized immediately at load time instead of waiting for an access to that
object. In some designs, lazy conversion is preferred to speed up object loading, but
in EPS we chose slower loading and fast access since programmer already has
control over the loading time of an object and can make his/her choice based on
performance and other considerations. Once replaced, the pointer values are valid
to be used in the process, and need to be converted back into persistent pointers
only when they are written back to the disk saving the system from conversion
overhead for every access to the persistent object. A reverse address translation is
applied to persistent object just before it is written back into the persistent store.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of the structure field</td>
</tr>
<tr>
<td>Type</td>
<td>Basic type of the field (e.g. char, int ...)</td>
</tr>
<tr>
<td>Pointer</td>
<td>Is field a pointer or not</td>
</tr>
<tr>
<td>Size</td>
<td>Field size</td>
</tr>
<tr>
<td>Count</td>
<td>Field count (1 for a variable, N for a N element array)</td>
</tr>
</tbody>
</table>

Address translation task consists of two major steps;

- The first task is understanding the structure of the object. This is done
  through type parsing of object’s type definition (.h) file. This is a standard C
  include file with object type defined as a C type. As a result, an object
  structure table with field, field type, pointer information and size is
  constructed (table 5.5). An example type definition file test.h is given here:

```c
typedef struct {
    char arr1[10];
    int *p1;
    int num1[10];
}TestObjectType;
```

- On the second phase, object structure table is used to transfer pointer fields
  of the object on disk. If for instance, root object is at address 1000, then
  pointer fields are relocated based on this starting address. Continuous
  address range is supported by PMalloc and PFree functions.

An example of address translation for a two node linked list is in figure 5.5

53
5.4 Naming and Protection Server

Naming and Protection Server (NPS) is implemented in the form of a server process. It runs as a background process. This choice makes it possible to include an indefinite size object table to perform some of the functions. NPS is responsible of the security and synchronization of object access. Each request for object access is received and handled by NPS (Figure 5.6).

InUse table (table 5.6), which itself is a persistent object, is used to keep the records of all currently used/loaded objects. There is only one copy of InUse table, so all object access control and synchronization is controlled with this information safely without any further investigation by objects other than NPS. It also holds the entire capability information. Since the number of loaded objects cannot be anticipated, its size is dynamically adjustable.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersistentID</td>
<td>Persistent object identifier</td>
</tr>
<tr>
<td>ClientPersistentID</td>
<td>Persistent identifier of client object using this persistent object</td>
</tr>
<tr>
<td>OwnerCapabilityID</td>
<td>Capability ID of owner client object</td>
</tr>
<tr>
<td>GroupCapabilityID</td>
<td>Capability ID of group, client object belongs to</td>
</tr>
<tr>
<td>OtherCapabilityID</td>
<td>Capability ID of for other client objects</td>
</tr>
<tr>
<td>OwnerCapability</td>
<td>Address of owner client object capability</td>
</tr>
<tr>
<td>GroupCapability</td>
<td>Address of group client object capability</td>
</tr>
<tr>
<td>OtherCapability</td>
<td>Address of other client object capability</td>
</tr>
<tr>
<td>ObjectName</td>
<td>Persistent object name</td>
</tr>
<tr>
<td>InUse</td>
<td>Object’s in use status</td>
</tr>
<tr>
<td>UsedBy</td>
<td>ID of client using this object</td>
</tr>
<tr>
<td>AccessLevel</td>
<td>Current access level</td>
</tr>
<tr>
<td>StartTime</td>
<td>Start time of current write level access</td>
</tr>
<tr>
<td>MaxUseTime</td>
<td>Maximum time allowed to use this object</td>
</tr>
<tr>
<td>EncryptionKey</td>
<td>Encryption value to check request validity</td>
</tr>
<tr>
<td>ObjectType</td>
<td>Persistent object’s type</td>
</tr>
<tr>
<td>RestrictedObjects</td>
<td>List of restricted client Ids</td>
</tr>
<tr>
<td>RestrictedObjectLevels</td>
<td>List of restriction levels (Access mode levels)</td>
</tr>
<tr>
<td>ObjectStatus</td>
<td>Is this object “Not modified”, “Modified” or “Deleted”</td>
</tr>
</tbody>
</table>

Requests to NPS are sent and received via inter-object communication subsystem. Communication primitives are explained in detail in Section 4.4.

5.4.1 Naming and protection services

NPS Services to other objects are listed below. Codes on the left hand-side are used to define the services in NPS. Request messages for active object services are also handled by NPS, but message values are defined in active object and there is no special treatment for different active object requests, other than access rights control.

ACMRO : Request a passive object for reading only. Other objects can access it to modify.

ACMSA : Request a passive object for reading, but do not let other objects to modify this object.

ACMRW : Request a passive object for reading and modifying

ACMRD : Request a passive object for reading and deleting
CAPRO : Request a capability
OPAIU : Make a new entry to the InUse table
OPREL : Release object from InUse table so that it can be accessed by other objects
OPCOM : Request a commit for passive object. NPS checks for validity of the request and then passes this request to OBS.
OPCOR : Request a commit and release for passive object. NPS checks for validity of the request and then passes this request to OBS.
OPROL : Request a rollback for passive object. NPS checks for validity of the request and then passes this request to OBS.
OPROR : Request a rollback and release for passive object. NPS checks for validity of the request and then passes this request to OBS.
CAPNE : Request a new capability.

5.4.2 Object name resolution

Persistent object names consist of two parts: creator object name and persistent object name. While creator object name is the creator program name, persistent object name is the persistent variable name declared in the program. Objects are created initially by EPS-C preprocessor (details in section 4.3). This two level naming scheme helps to distinguish the persistent objects created by a particular active object. It also reduces the probability of collision in persistent object names. Normally, long life span and very wide scope of persistent objects increase the probability of using the same name for different objects.

When a request for an object is made with its persistent object name, it is NPS's job to convert this name which is meaningful to humans, to the persistent identifier which is a long integer and a more natural representation for the computer. An example for this representation is given to clarify the design: Assume that our program name is warehouse.epsc, and a persistent variable is declared as below

... $int stockCount ...

Then, the Persistent Object Name is warehouse.stockCount, persistent ID can be any long integer given by EPS-C preprocessor such as 10000.

NPS searches for a requested persistent object’s name in the inUse table. NPS first searches for the full name with owner and object name. If it fails to have a full match, it then tries to match the object name only. This approach will prevent object
name collision by different processes. In the example above, a search for stockCount shall be conducted first with its full name “warehouse.stockCount”, if this search fails, a new search for the name “stockCount” is performed.

Object name resolution task is conducted when an object is to be requested but its capability is not available in the program’s local capability store. In this case capability for this object is requested from NPS, which in turn looks for it in NPS capability store with the algorithm explained later in this section, and returns this capability to the requesting client object. If, capability is available in local capability store, the same search algorithm is used to find it locally.

5.4.3 Request validation and access synchronization

The target object name in every request for loading a static object or for an operation from an active object is first converted to Persistent IDentification (PID) by matching the requested target object name with the existing capabilities, and then evaluated by NPS. Validation is done in four steps:

1. Find the record for the target object in the inUse table. If it is not available ask OBS to load it from the secondary memory.
2. Control if requesting object has restricted access
3. Check the access level of the requested object to see if it allows for the requested operation.
4. Check the validity of the capability by verifying the encryption.

If a request is validated, then a record is inserted into the inUse table for the persistent object. Requested operation is forwarded to OBS if necessary. For instance, a Load Passive Object request is sent to OBS to be realized. In this case, the result of the load operation is sent to the client by OBS (figure 5.2 explains the message flow). A client process can access multiple persistent objects simultaneously. The access control and protection mechanisms will be used for the entire persistent object. A finer grained control which would enable client processes to control a subset of an object such as a part of a tree structure is not supported.

The possible state transitions for passive object access in inUse table are shown in table 5.7. Each column of this table explains allowed new access modes for each access status of an object in InUse Table. The access modes are:

**NO** : No access, meaning that object is not used by any client. In this case, all new access requests are granted.

**RO** : Read only, object is used by one or more clients in read only mode.

**SA** : Shared access, object is used at least by one client with shared access permission, in this case, it is not possible to access the object in readwrite
mode.

**RW**: Read/write, object is used by one client with read/write permission.

**RWD**: Read/write/delete, object is used by one client with read/write/delete permission.

<table>
<thead>
<tr>
<th>NO→RO</th>
<th>RO→NO</th>
<th>SA→NO</th>
<th>RW→NO</th>
<th>RWD→NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO→SA</td>
<td>RO→SA</td>
<td>SA→RO</td>
<td>RW→RO</td>
<td>RWD→RO</td>
</tr>
<tr>
<td>NO→RW</td>
<td>RO→RW</td>
<td>SA→SA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO→RWD</td>
<td>RO→RWD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7 Access rights state transitions

Synchronization has a different context for passive (data only) and active (server) objects. An active object can answer only one request at a time, so the synchronization problem is naturally solved. A passive object can be shared among multiple processes, therefore access synchronization has to be explicitly managed. When a request for a passive object is validated, the access level of the inUse row is set by values in Table 5.7. When a new request comes for the same object, the current access level and state transition rules are used to decide if it is possible to satisfy this request. If it is not accessible, than NPS sends a rejection message to requesting client object. Client object tells back when it is done with the requested object, so the access level is reset.

This approach has one drawback, deadlocks. When two objects request objects already accessed by each other, this scheme will cause a deadlock. Instead of deadlock resolution, we chose to implement an extra feature to prevent deadlocks. When COB accesses a new object, it tells the expected completion time of the request in seconds, and it is recorded in the inUse table. NPS resets the access level to zero or decreases the level of access by one 1, in a periodic control loop on its idle time. These two methods can be used together to have a better control and better performance.

### 5.5 Object Server

Object Server (OBS), which can be considered as a part of the operating system kernel due to its functionality, is the persistent object store of EPS. Physically it is implemented as a separate module and runs as a background process. The Object Server handles the storage and retrieval of passive and active objects. While active objects consist of data and methods, a passive object contains data but does not have methods to modify it.
Since OBS is part of the operating system kernel, it is easier to coordinate it with other kernel services while loading objects from secondary storage area. When a request for a load operation is received, OBS first decides if the target is a passive or an active object, then loads it as appropriate. Since OBS accepts requests only from NPS, it need not to worry about synchronization and access authorization issues.

The basic service supplied by OBS are:

**ACMRO**: Load passive object into the primary memory. Access modes in NPS is not used in OBS.

**OPCOM**: Commit passive object. Stores the modified persistent object on disk.

**OPLOA**: Load active object as a process into the primary memory.

The persistent store architecture is two leveled (figure 5.7). A two level store has a clear distinction between primary and secondary memory. This decision is based on the flexibility of two level stores. On the other hand, one level stores are fast but have restricted capabilities, such as lack of transaction control. With a two level store architecture the general concept of persistent object can be inspected in three levels.

![Persistent Object Heap - Session](image)

<table>
<thead>
<tr>
<th>Persistent Object Heap - Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stable Store - OBS + Object File</td>
</tr>
<tr>
<td>Non-volatile Store - Linux File System + Harddisk</td>
</tr>
</tbody>
</table>

**Figure 5.7 Persistent store levels**

Highest level is the stable heap of persistent objects in client object session. This is the layer presented to the persistent programming language or object-oriented database. It allows for the creation, manipulation and management of complex objects. The granularity of access to this layer is object-based. This layer is implemented in the client object via client object library, as explained in the section 5.3.

Under the stable heap, there is stable store, which is implemented via structured files and OBS services. This layer provides a contiguous range of virtual addresses that are always in a self-consistent state. The lowest level, non-volatile store is an
interface to a persistent physical storage mechanism (i.e. a file system API as a high level interface or a low level disk drive API).

5.5.1 Data integrity

Consistency of multiple objects in memory is maintained with a "shadow object" mechanism. The objects are kept in the memory in two forms; original and modified. In addition to this status information, InUse table is used to keep track of the object. When a "commit" statement is issued, the modified versions are copied into the original versions and all heap values with modified flag set are written to the file system. Therefore it is possible to maintain consistency of interrelated local passive objects.

Service requested from an active objects is another important issue. When a client object requests a modification from an active object as part of its transaction, the system automatically sends rollback /commit requests to the active object. Active object library offers basic commit and rollback services, but issues like controlling remodification of a modified but not committed object are expected to be handled by programmer.

Shadow files technique is used to reduce the risk of creating an inconsistent database. In shadowing, the modified object file is written back to the disk into a different file first. After the file write is complete it is copied into the original file. Since it brings a certain performance penalty, this function is made optional and specified when starting Object Server (OBS).

The capability of maintaining consistency among multiple objects is an open issue to be discussed. At a glance, currently implemented mechanism seems to be useful, but on the other hand, it still lacks the assurance of data integrity among distributed objects.

5.5.2 Object store on disk

Persistent objects are stored on the disk for long term storage. The header structure of a static passive persistent object on disk is shown in table 5.8. The detailed object structure is explained in appendix A.

Persistent objects are loaded into the memory by OBS. A passive object loaded into the process memory is represented by its root object. Root object addressed by persistent object in program is the one variable accessible directly from the program instructions. Other related objects are linked to the root object via pointers. Client
processes will get the physical address of the root persistent object, and then reach the other objects using pointers.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersistentID</td>
<td>Persistent ID of the object</td>
</tr>
<tr>
<td>ObjectName</td>
<td>Persistent object name</td>
</tr>
<tr>
<td>ObjectLength</td>
<td>Length of persistent object</td>
</tr>
<tr>
<td>ObjectType</td>
<td>Basic or complex object type</td>
</tr>
<tr>
<td>ModifiedAt</td>
<td>Modification time</td>
</tr>
<tr>
<td>ModifiedBy</td>
<td>Modified by which object</td>
</tr>
<tr>
<td>DataFileName</td>
<td>File name of persistent object on disk</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic redundancy code</td>
</tr>
</tbody>
</table>

Table 5.8 Persistent object on disk

Each persistent object group includes a persistent root. Each persistent object group is kept in different file and includes a homogeneous structure inside its file. These groups are called as persistent databases.

The type information of the persistent objects is kept inside the persistent root. While the simple types such as integer or character are defined directly in the persistent root header, complex user defined types are referenced in the header and kept as type definition include files separately as explained in section 5.3.2.

5.6 Active Object

Active Objects (AOB) are not part of the core EPS. However we expect to see a considerable number of AOB to be developed in the future which will benefit from advantages of EPS. In short, active objects are persistent server processes which themselves can act as clients objects to other active objects and EPS.

An important aspect of active objects is their support by the EPS. EPS transparently loads active objects whenever they are needed. The access control for active objects is handled by NPS, similar to passive objects. Active objects also use the COL library and the inter-object messaging system.

Some methods are mandatory if a program is to be named as an active object:

- Rollback operation
- Commit operation
- Execute local method
In addition to these mandatory operations, an interface which accepts requests from other objects is a natural part of an active object. Standard interface of an active object is implemented by the active object library to ease programmer’s task of server development. This library supplies messaging and synchronization service, and a framework for server processing cycle. Development of new active objects is explained in section 4.2.

5.6.1 Active object library

Active Object Library supplies a framework for server programs. AobServer function implements a loop to listen and accept client object requests. Functions supplied by the Active Object Library are;

- **AobServer**: Provides a request handler loop. Redirects requests according to their content.
- **AobCommit**: Commit data modified by the active object.
- **AobRollback**: Rollback active object data

5.7 EPS Security Architecture

EPS security is based on capabilities. Security issues are handled by NPS as explained in the former section.

5.7.1 Capability handling

In EPS, access control to the objects relies on capabilities. EPS capability structure includes three IDs used for access rights limitation, persistent object id and name, rights given by this capability, and fields to determine if this capability is transferrable or not. EPS capability structure is explained in detail in Appendix A. Each capability presented by the requesting process will be checked if its clientObjectld field matches the presenter, or it has the special "can be used by every object" value in the clientObjectld field and the capability field “transfer” is true. Available functions to handle capabilities are listed in table 5.9.

Every client object has its local capability store as well as the global capability store owned by NPS (figure 5.8).

Capability store is a special type of file with a hash structure, therefore a single access is sufficient to retrieve a capability, if we have the capability ID.
If the server wants to give the transfer right of a capability, it assigns a special value to the clientObjectld field and set the transfer field to true.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequestCapability</td>
<td>Request a capability from Naming and Protection server</td>
</tr>
<tr>
<td>CapabilityTransfer</td>
<td>Transfers a capability to another active object.</td>
</tr>
<tr>
<td>RightsCancel</td>
<td>Limits or removes access permissions to a persistent object</td>
</tr>
<tr>
<td>LoadCapability</td>
<td>Load a capability from capability store</td>
</tr>
<tr>
<td>LoadCapByName</td>
<td>Load a capability from capability store, by its name</td>
</tr>
<tr>
<td>AddCapability</td>
<td>Add a capability to capability store</td>
</tr>
<tr>
<td>RemoveCapability</td>
<td>Remove a capability from capability store</td>
</tr>
<tr>
<td>ExtendCapability</td>
<td>Extend the size of capability store</td>
</tr>
<tr>
<td>ReturnCapID</td>
<td>Return an unused (new) capability ID</td>
</tr>
</tbody>
</table>

With a clientObjectld field set and a false transfer field capability, a client process owning the capability can only use it for itself but can not transfer it. These types of capabilities are transferred through the naming and protection server requests.

The server always checks to see if target id is valid or not, so that an unauthorized copy of the capability can not be used to gain access to the related resource.
Capability objects can be stored in two different locations: local capability stores for each client, and a common capability store managed by NPS. A process can always load the capabilities in its program table while it can only retrieve capabilities in public tables if it has the required capability for them. Once the object is activated—that is loaded into memory, the capability is also loaded into the InUse table.

Encryption value = Encryption Function(ObjectId, Rights, Encryption key)

Encryption functions are predetermined one way functions, which will only be known by a related server process. Assessing the function and the value will do encryption control.

Using this technique, capabilities can be passed to other objects if access rights are not modified. A modification in rights in the transfer makes the intervention of the NPS necessary.

5.7.2 Access control

Access control mechanism is central and realized by the NPS. While this is not the fastest method it enables to resolve many of the security-related issues, and it is assumed that it will not be used frequently. Access control mechanism will be used only to load the persistent roots and during access to methods in other active objects.

All rights cancellation: All three ID numbers (owner, group, other) in the inUse table are modified. No client object with the already distributed capabilities can access the object.

Selective cancellation: Some of the three ID umbers are modified. Clients using these numbers become incapable of accessing the object.

Access restriction with levels: Object ID is put into the relevant field of inUse table. This particular object is denied access to a certain access level denied in the restriction field.

5.7.3 Heap management and garbage collection

Each client process has its own persistent heap. The EPS provides facilities such as allocation, reallocation and release to manage this memory area. There is no garbage collection facility for the persistent heap in memory since programmers themselves decide which objects are persistent. On the other hand, garbage collection in persistent heaps is a major area of concern in orthogonal systems where all objects are considered to be persistent automatically.
Garbage collection is not supported for the persistent objects on disk either. The ability to find and delete a persistent object when it not used anymore is an open issue to be discussed.
6. EPS PERFORMANCE

EPS performance can be investigated on two aspects; execution performance and developer productivity.

6.1 EPS Program Execution Performance

The basic difference of an EPS program from a conventional program is its use of external services for persistent object management. Another issue of performance is the comparison of time consumption in address translation versus memory-disk representation translation. EPS is not an orthogonal system, therefore it does not have a considerable performance penalty since it manages only programmer declared persistent objects.

Run-time performance of EPS can be investigated in two phases; object load/save operations, and already-loaded object manipulation. While object loading times shall differ with conventional systems (formulated in table 6.4), object manipulation times shall not be different than conventional programs.

Table 6.1 shows object loading times of a b-tree with epsc and c programs on three platforms. While results in older systems shows significant performance penalty, improvement in Pentium IV computer is due to the improved disk caching mechanisms, and always running Object Server module. Hence, with improved hardware capabilities of today, EPS like systems are becoming feasible more than ever.

<table>
<thead>
<tr>
<th>Program</th>
<th>Pentium (ms)</th>
<th>Pentium II (ms)</th>
<th>Pentium IV (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demo1.epsc</td>
<td>3508</td>
<td>797</td>
<td>199</td>
</tr>
<tr>
<td>Demo11.c</td>
<td>1907</td>
<td>352</td>
<td>208</td>
</tr>
</tbody>
</table>
When we analyse the object manipulation times of the two programs, it is observed that there is no significant performance difference between two programs as seen in table 6.2.

<table>
<thead>
<tr>
<th>Program</th>
<th>Pentium (ms)</th>
<th>Pentium IV (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demo1.epsc</td>
<td>4584</td>
<td>95</td>
</tr>
<tr>
<td>Demo11.c</td>
<td>4603</td>
<td>109</td>
</tr>
</tbody>
</table>

To compare theoretical execution times of an EPS and a conventional program, we will describe a scenario: Let's assume that we have a simple program that reads an object, processes it and then writes it back to the disk. We assume that our program already has the appropriate rights to read and write this object. Time components that will take place in calculation of total execution time are:

- $T_m$: Message transfer time
- $T_r$: Time to read an object from disk
- $T_{rc}$: Time to read a capability from disk
- $T_w$: Time to write an object to disk
- $T_{c}$: Time to convert object representation between disk and memory
- $T_t$: Time for address translation
- $T_p$: Time for processing an object

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read object from disk</td>
<td>$T_r$</td>
</tr>
<tr>
<td>Convert into memory representation</td>
<td>$T_c$</td>
</tr>
<tr>
<td>Process</td>
<td>$T_p$</td>
</tr>
<tr>
<td>Convert into disk representation</td>
<td>$T_c$</td>
</tr>
<tr>
<td>Write object to disk</td>
<td>$T_w$</td>
</tr>
<tr>
<td><strong>Total Time</strong></td>
<td>$T_r + 2T_c + T_p + T_w$</td>
</tr>
</tbody>
</table>

Table 6.3 and 6.4 give total execution times of the conventional and EPS programs, respectively, in terms of the basic time components. When the two tables 6.3 and 6.4 are compared, it is observed that total execution times of both programs have $T_r$, $T_w$ and $T_p$ in common. The primary difference is in message transfer times and object conversion versus address translation times. If we neglect message transfer time, as message transfer occurs at kernel level, the time spent to convert object representation in a conventional C program and the time EPS uses for address translation are the most important factors that account for execution performance.
Table 6.4 EPS Client object execution time breakdown

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Capability</td>
<td>Trc</td>
</tr>
<tr>
<td>Load Object</td>
<td>Tm+Tm+Tm+Tr+Tt</td>
</tr>
<tr>
<td>Send Load Request to NPS</td>
<td>Tm</td>
</tr>
<tr>
<td>Send Load Request to OBS</td>
<td>Tm</td>
</tr>
<tr>
<td>Read object from disk (OBS)</td>
<td>Tr</td>
</tr>
<tr>
<td>Receive object</td>
<td>Tm</td>
</tr>
<tr>
<td>Address translation</td>
<td>Tt</td>
</tr>
<tr>
<td>Process</td>
<td>Tp</td>
</tr>
<tr>
<td>Commit</td>
<td>Tm+Tm+Tw+Tt</td>
</tr>
<tr>
<td>Address translation</td>
<td>Tt</td>
</tr>
<tr>
<td>Send Commit Request to NPS</td>
<td>Tm</td>
</tr>
<tr>
<td>Send Commit Request to OBS</td>
<td>Tm</td>
</tr>
<tr>
<td>Write object to disk (OBS)</td>
<td>Tw</td>
</tr>
<tr>
<td>Total Time</td>
<td>5<em>Tm + 2</em>Tt + 2*Tr + Trc + Tw + Tp</td>
</tr>
</tbody>
</table>

At this point, we can argue that address translation can be more efficient and is not affected by the complexity of the objects as opposed to conversion operation which can be very complex depending on the object complexity. Consequently, together with total programmer control over persistency property, EPS programs can have acceptable execution performance when compared to conventional programs. Table 6.1 shows experimental results supporting this hypothesis.

6.2 EPS Programmer Productivity

Programmer productivity has been a subject of interest for many years and different methods such as correlating programmer productivity to lines of code and function point counts has been developed (Albrecht and Gaffney, 1983, Matson et al, 1994). Productivity is estimated statistically by Jones (1986) based on function points and source line of code (SLOC) (Ragland, 1994). Table 6.5 lists productivity values for some popular programming languages.

Table 6.5 Programmer productivity

<table>
<thead>
<tr>
<th>Language</th>
<th>Productivity Average Per Staff Month (SLOC/Month)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>750 – 1500</td>
</tr>
<tr>
<td>Pascal</td>
<td>455 – 910</td>
</tr>
<tr>
<td>Ada</td>
<td>710 – 1420</td>
</tr>
</tbody>
</table>
To investigate the advantage (if any) of program development in EPS over conventional Linux environment, a simple program for the following scenario is developed. Let’s assume that a scientist is collecting some data everyday, for 10 days. He wants to save these data daily in a file so that at the end, he can quickly analyse data. Two programs, “tableproc.c” and “tableproeps.epsc” (source codes are listed in appendix C) to satisfy this requirement are written in C programming language and EPS-C, respectively. Two programs do exactly the same task:

- load a table of data from secondary storage
- display table content
- prompt for new data
- save modified table to secondary storage

<table>
<thead>
<tr>
<th>Table 6.6 Comparison of C and EPS-C program source</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Language</td>
</tr>
<tr>
<td>Source Line of Code</td>
</tr>
</tbody>
</table>

The resulting source line of codes are compared in table 6.6. EPS-C code is significantly (27%) shorter when compared to C code performing the same task. The percentage of difference is similar to observations of developers of PS-Algol that we have mentioned in section 1 (Cockshot et al, 1984). The percentage difference of the number of source lines of code can be accepted as the measure of the advantage gained by using EPS.

Code sections and relationship of different types of sections to the level of gain is an interesting issue. If we can model this relationship, than it is possible to inspect program requirements and decide whether it is better to use EPS-C or not, considering run time and development productivity tradeoff.

Two factors influence directly the advantage gained in the persistent environment:

- Number of persistent objects (N)
- Complexity of persistent objects (C)

We can formulate the gain function as:

\[ \text{Gain} = f(N, C) \quad (6.1) \]

Object complexity, is the non countable item of the formula, so we shall have a classification for complexity. A simple yet useful complexity classification is suggested in table 6.7.
### Table 6.7 Classification of object complexity

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Complexity Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple C Type</td>
<td>1</td>
</tr>
<tr>
<td>Array Type</td>
<td>2</td>
</tr>
<tr>
<td>Multidimensional Array Type</td>
<td>3</td>
</tr>
<tr>
<td>Linked List (all types)</td>
<td>4</td>
</tr>
<tr>
<td>Tree (all types)</td>
<td>5</td>
</tr>
<tr>
<td>More Complex</td>
<td>6</td>
</tr>
</tbody>
</table>

This model and formula (6.1) shall be tested with a number of programs to derive an applicable formula for the estimation of gain of EPS-C over conventional C programs, at the requirements and design phases. However result of this formula can not be accepted as a fact but only an estimation. Precision of the formula will depend on the number of experiments conducted and data collected.

On the other hand, potential gain of converting an existing conventional program can be calculated as percentage of source line of code in internal processing sections to source line of code in input/output processing sections (SLOC.IP/SLOC.IO). Therefore, formula 6.2 calculates advantage of EPS program as a percentage difference.

\[
Gain = \frac{SLOC.IP}{SLOC.IO}
\]

(6.2)
7. CONTRIBUTIONS AND FUTURE WORK

In this section contributions of this thesis study, and the future research directions are discussed. The outcomes of this thesis study can be discussed in three parts; novel solutions to some existing problems, a different model for understanding and developing systems, and being a platform for future research.

As explained in the second and third sections, some issues in the development of persistent and extensible systems were not addressed properly up to now. An open issue for capability based security architectures was the cancellation of once distributed capabilities. While some methods of complete cancellation is proposed, selective cancellation which means to remove access rights of some processes while leaving others valid was a problem in many systems. Another problem in access rights cancellation is the restriction of some rights while leaving others. This is a case when we want to cancel the write permission once distributed to other clients leaving only the read permission. EPS, allows a capability owner to remove a capability completely, or partially or restrict the once given access rights selectively. The transfer of capabilities from one process to another was the last issue, which EPS tried to resolve.

In a conventional system, variable names and scopes are checked by the compiler according to the language rules, which resolves name conflict problem in the compile phase. Also, since the variable names are visible to the programmer, there is little chance of making mistakes or losing a variable. Another problem with a persistent system arises because of the long life span of persistent objects and their wide scope in contradiction to conventional systems. A persistent object identifier, which is expected to be unique, may have already been used before by another programmer. The process of finding a unique identifier can easily become a tedious job, even impossible. This issue is attacked by a naming structure consisting from two parts; owner object name, and persistent object name. A request with only the persistent object name is resolved using the second part of names. If more than one persistent object with the same name exists, type checking is conducted and the object with the same type is returned. If more than one object with the same name and right type exists, an error is returned.
EPS suggests a flexible, programmer controlled approach to synchronization and deadlock prevention. The synchronization approach is similar to that of relational database management systems that seems to be a successful solution used widely in the industry. For the deadlock prevention, a choice of gradually or one-step lock release is supplied to the programmer which shall be flexible to satisfy different types of application development needs.

EPS suggests a new model for extensible systems based on a unifying view of persistency. Data and processes are viewed as persistent objects, and handled through a uniform interface. The system definition can be developed recursively using the notions of active, passive, and static objects. We believe this simple yet powerful view of the system modeling will support application developers and system programmers in understanding and extending the system.

Active object development is supported through a standard interface and a library of functions. EPS support enables programmers develop active objects with little effort unlike conventional server program development tasks.

The better modeling together with the high level programming facilities such as persistent object support and inter-object communication enables applications programmers to develop extensions to the system easily and without creating risks of security.

The last set of contributions in this thesis is being a platform for future research on investigating the effects of an extensible persistent system on programmer productivity.

Since persistent object creation is under programmer control, we were able to ignore the job of garbage collection to some extend. While this will not cause a problem in the primary memory, the management of objects in the secondary memory is a future research issue.

While EPS has facilities to promote system integrity at the programming level, the system does not have a protection on system integrity of multi object modifications in the case of a system crash. In this case, while some of the objects may be modified, others may remain unmodified, leaving the system in an undefined state. Multi object integration architecture shall be redesigned to overcome this potential problem.

An integrated development and system management environment will be very useful to speed up experimental studies.
There is a need for experiments to compare the programmer productivity in EPS versus a conventional system. The programmer productivity gain model defined in section 6 needs comprehensive experiments in order to become an applicable formula.

We shall also investigate the suitability of the system for different types of applications practically, by carrying out structured performance tests, of which theoretical foundation is explained in section 6.
REFERENCES


APPENDIX A MAJOR EPS DATA STRUCTURES

Persistent Object in Memory
struct MemoryObjectStruct
{
    PIDType persistentlD;
    char *objectName[NAMESIZE];
    long dataLength;
    long maxDataLength;
    int objectSize;
    char *data;
    char *objectStatus;
    char *sessionStatus;
    char *copyStatus; /* Shadow|Original*/
    struct MemoryObjectStruct *shadowObject;
    char *objectType[TYPESIZE];
    char *dataFileName[FILENAMESIZE];
    CapabilityType *capability;
    char *modifiedAt[DATESIZE];
    char *modifiedBy[USERNAMESIZE];
    char CRC[8];
    struct MemoryObjectStruct *prevObject;
    struct MemoryObjectStruct *nextObject;
};

Persistent Object on Disk
typedef struct
{
    PIDType persistentID;
    char *objectName[NAMESIZE];
    long objectLength;
    char *objectTypeStatus;
    char *objectType[TYPESIZE];
    char *modifiedAt[DATESIZE];
    char *modifiedBy[USERNAMESIZE];
    char *dataFileName[FILENAMESIZE];
    char CRC[8];
} DiskObjectType;

Session
typedef struct
{
    MemoryObjectType *objects;
    SIDType SID;
} SessionType;
typedef struct
{
    int      rNo;
    char     rText[TEXTSIZE];
}ResultType;

InUse Table
typedef struct
{
    PIDType       persistentID;
    PIDType       clientPersistentID;
    CIDType       ownerCapabilityID;
    CIDType       groupCapabilityID;
    CIDType       otherCapabilityID;
    CapabilityType ownerCapability;
    CapabilityType groupCapability;
    CapabilityType otherCapability;
    char          objectName[NAMESIZE];
    char          inUse;  /* Yes/No/Will be */
    sessionIDType usedBy;
    int           AccessLevel;
    int           startUseTime;
    int           maxUseTime;
    char          encryptionKey[8];
    char          objectType[TYPESIZE];
    PIDType       restrictedObjects[OBJECT_COUNT];
    int           restrictedObjectLevels[OBJECT_COUNT];
    char          objectStatus;  /* Not modified|Modified|Deleted*/
}InUseRowType;

Capability
typedef struct
{
    CIDType       persistentID;  /* Capability ID */
    PIDType       clientPersistentID;  /* PID of the related object */
    PIDType       capabilityID;  /* Who uses the capability */
    char          capabilityType;  /* 'O'wner/'G'roup/o'T'her */
    char          objectName[NAMESIZE];  /* Name of the related object*/
    char          rights;
    char          passiveOperation;  /* Also the access level*/
    OperType      activeOperation[MAXFUNCTION];  /* Active operations */
    char          objectType[TYPESIZE];
    int           capabilityTransfer;
    char          transferMask[2];
    char          CRC[8];
}CapabilityType;

Message
typedef struct
{
    int          operation;

Table Object
typedef struct
{
    char key[30];
    char data[100];
}TableType;
APPENDIX B EPS SOURCE FILES

CAPLIB.C

Capability library defines capability functions that used by other EPS system programs. Public functions of capability library are:

CapabilityType LoadCapability(FILE *capF,long capID);
int LoadCapByName(FILE *capF,char objectName[NAMESIZE], CapabilityType* lCap);
int AddCapability(FILE *capF,CapabilityType aCap);
int RemoveCapability(FILE *capF,long capID);
int ExtendCapability(FILE *capF, int extendSize);
int ReturnCapID(FILE *capF, CapabilityType *lCap);

EPSAOB.C

Active Object library defines functions that help to develop Active Objects. Public functions of active object library are:

int AobServer(int (*f [MAXFUNCTION])(char*,int));
int AobCommit(char*,int);
int AobRollback(char*,int);

EPSCOB.C

Client Object library defines functions used in client objects. Public functions of client object library are:

ResultType LoadPersistentObject(SessionType *session, char** pVar,
                                  CapabilityType caV,int acM);

ResultType SavePersistentObject(SessionType *session, char** pVar,
                                  CapabilityType caV,int operation);

ResultType ReleasePersistentObject(SessionType *session, char** pVar,
                                     CapabilityType caV, int operation);

ResultType EpsInit(SessionType *, char prgName[FILENAMESIZE], FILE* lCapFile);

int Request(CapabilityType caV,int operation, char *parameter, ResultType *result);
int TableRead(SessionType *session, char tableName[NAMESIZE], char key[TABLEKEYSIZE], char data[TABLEDATASIZE]);
int TableWrite(SessionType *session, char tableName[NAMESIZE], char key[TABLEKEYSIZE], char data[TABLEDATASIZE]);

int TableDelete(SessionType *session, char tableName[NAMESIZE], char key[TABLEKEYSIZE]);

int TableAppend(SessionType *session, char tableName[NAMESIZE], char key[TABLEKEYSIZE], char data[TABLEDATASIZE]);

ResultType RequestCapability(FILE* capF, char objectName[NAMESIZE], CapabilityType *ICap);

ResultType EpsClose(SessionType *session, int operation);

ResultType ObjectCommit(SessionType *, char objectName[NAMESIZE], int operation);

ResultType ObjectRollback(SessionType *, char objectName[NAMESIZE], int operation);

ResultType EpsRollback(SessionType *session, int operation);

ResultType EpsCommit(SessionType *session, int operation);

MemoryObjectType *CheckObjectByID(SessionType *session, PIDType pID);

MemoryObjectType *CheckObjectByName(SessionType *session, char oName[NAMESIZE]);

char* PMalloc(SessionType* session, char objectName[NAMESIZE], int count);

void PFree(SessionType* session, char objectName[NAMESIZE]);

EPSCONV.C

This source file includes functions to make conversion between persistent address and memory address. Public functions of epsconv are:

ResultType LocalizeVariable(char *loadedObject, char **memoryObject, int acM, long *objSize);

ResultType GlobalizeVariable(DiskObjectType *dOb, char **pObject, char *oObject);

int CalculateStructSize(TypeTableType tt[MA反复VAR_NO]);

EPSCPP.C

EPSCPP is the EPS-C preprocessor program. Command syntax for EPS-C preprocessor is:

epscpp sourceFileName <-l> <-cc>

"-l" option generates code to load object load statements

"-cc" option generates capability file for new program

"-co" option generates persistent object on disk for new program
EPSLIB.C

General utilities for EPSC programs. Public functions of this library are:
ResultType FillError(int, char *);

EPSMSG.C

Epsmsg file defines functions used in inter-object communication. Public functions of
message library are:
int SendMessage(PIDType objectld, PIDType senderObjectld, int operation,
    CapabilityType capability, char *parameter, ResultType result);

int ReceiveMessage(PIDType objectld, PIDType *senderObjectld, int *operation,
    CapabilityType *capability, char *parameter, ResultType *result);

EPSNPS.C

This source file is the implementation of Naming and Protection server. Since it
communicates with clients and Object server via inter-object communication system,
it does not include any public functions. Command syntax for Naming and Protection
server is:
epsnps &

EPSOBS.C

This source file is the implementation of Object server. Since it communicates with
clients and Naming and Protection server via inter-object communication system, it
does not include any public functions. Command syntax for Object server is:
epsobs <-s> &
"-s" option enables shadow files mode.

PARSER.C

This source file includes functions to parse a complex c type. Public functions of
parser are:
ResultType PopulateTypeTable (char TypeFile[TYPESIZE], TypeTableType
    typeTable[MAX_VAR_NO]);

CAFCREAT.C

A utility program used to inital creation of capability store.
CAPLIB.H
Capability Library structure and public function declarations.

EPSAOB.H
Active Object library structure and public function declarations.

EPSCOB.H
Client Object library structure and public function declarations.

EPSCONV.H
Address translation related structure and public function declarations.

EPSCPP.H
EPS-C preprocessor structure and function declarations.

EPSERROR.H
Standard error constant declarations.

EPSLIB.H
General library structure and public function declarations.

EPSMSG.H
Inter-object messaging structure and public function declarations.

EPSNPS.H
Naming and Protection Server structure and function declarations.

EPSOBS.H
Object Server structure and function declarations.

EPTYPES.H
Common structure and constant declarations.

TABLE.H
Table object structure declaration.
C.1. DISPLAYING NAMES IN A B-TREE

```
#include "epstypes.h"
#include "epsmsg.h"
#include "epslib.h"
#include "epsobs.h"
#include "epsnps.h"
#include "epscoib.h"
#include "epscovn.h"
#include "caplib.h"
#include "table.h"
#include <name.h>

$nameType    pName;
$tableType   pPrgTable;
SessionType  session;
capabilityType cName;
EpsResult    result;
char         ad[30];
int main()
{
    /* Initialize session */
    result = EpsIni(&session,"btree",cName);
    if (result.rNo) exit(1);

    /* Explicitly Load b-tree */
    LoadPersistentObject(&session, &parameter, cName, ACMRO);
    AdListele(parameter) ;

    /* Release EPS session, and commit changes of session objects*/
    EpsClose(&session,COMMIT);
}
```
C.2. INCREASING ITEM NUMBER IN A STORE SERVER

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "test.h"
#include "epstypes.h"
#include "epsmsg.h"
#include "epslib.h"
#include "epsobs.h"
#include "epsnps.h"
#include "epsacb.h"
#include "epsconv.h"
#include "caplib.h"
#include "table.h"
#include "epsaob.h"
#include "warehouse.h"

/* Server Function Prototypes*/
int AddItem(char *parameter, int amount);
int GetItem(char *parameter, int amount);

/* Global variables */
SessionType session;
int (* f[MAXFUNCTION]) (char *, int) = {AobCommit, AobRollback, NULL, NULL, NULL, NULL, AddItem, GetItem};
PIDType processID = 100000;
char *mainObject="warehouse";

Warehouse manager active object program

int main(int argc, char *argv[])
{
    ResultType sonuc;
    FILE *ICapF;
    CapabilityType caV;
    $StockType stock;

    /* Initialize an EPS session, Object load code is inserted automatically
     * in compilation */
    EpsInit(&session,"warehouse",ICapF);

    /* Main server function (a loop) to listen and server client requests */
    AobServer();
Release EPS session, and commit changes of session objects*/
EpsClose(&session, COMMIT);

return(0);
}

 לתת Items to the stock

int AddItem(char *parameter)
{
    int amount;
    amount = atoi(parameter);
    stock->item = stock->item + amount;
    return(amount);
} /* End of function AddItem */

Get Items from the stock

int GetItem(char *parameter)
{
    int amount;
    amount = atoi(parameter);
    stock->item = stock->item - amount;
    return(0);
    return(amount);
} /* End of function AddItem */
C.3. TABLE PROCESSING PROGRAM IN EPS-C

/* TablePro.c : Table handling demo program in EPS-C */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "epstypes.h"
#include "epsmsg.h"
#include "epslib.h"
#include "epsclib.h"
#include "epsconv.h"
#include "caplib.h"

#define MAXROW 10
#define MAXCOLUMN 3

int main(int argc, char* argv[]); void DisplayTable(int table[MAXROW][MAXCOLUMN]); void ModifyTable(int table[MAXROW][MAXCOLUMN]);

/* Table Demo main function */
int main(int argc, char* argv[])
{
    SessionType session;
    FILE *lCapF;
    $int table [MAXROW] [MAXCOLUMN];

    EpsInit ( &session, "table", lCapF);
    DisplayTable(table);
    ModifyTable(table);
    EpsClose(&session, COMMIT);

    return 0;
}

/* Display table data on screen*/
void DisplayTable(int table[MAXROW][MAXCOLUMN])
{
    int i,

    for(i=0;i<MAXROW;i++)
        printf("%d %d %d\n",table[i][0],table[i][1],table[i][2]);
}

}/* End of Function DisplayTable */
/* Modify data in table */
void ModifyTable(int table[MAXROW][MAXCOLUMN])
{
    int i,j;

    printf("Row number: ");
    scanf("%d", &i);
    printf("Column number: ");
    scanf("%d", &j);
    printf("Cell value: ");
    scanf("%d", &table[i][j]);
}

/* End of Function ModifyTable */
C.4 TABLE PROCESSING PROGRAM IN C

/* TableProc.c : Table handling demo program in C */
#include <stdlib.h>
#include <stdio.h>
define MAXROW 10
#define MAXCOLUMN 3
int main(int argc, char* argv[]);
int LoadTable(char * fileName, int table[MAXROW][MAXCOLUMN]);
int SaveTable(char * fileName, int table[MAXROW][MAXCOLUMN]);
void DisplayTable(int table[MAXROW][MAXCOLUMN]);
void ModifyTable(int table[MAXROW][MAXCOLUMN]);

int table[MAXROW][MAXCOLUMN];

/* Table Demo main function */
int main(int argc, char* argv[])
{
    LoadTable("tableproc.dat",table);
    DisplayTable(table);
    ModifyTable(table);
    SaveTable("tableproc.dat",table);
    return 0;
} /* End of Function main */

/* Load data table from disk */
int LoadTable(char * fileName, int table[MAXROW][MAXCOLUMN])
{
    int i,j;
    FILE * tr;

    tr = fopen(fileName,"r");
    if (tr == NULL)
        return -1;
    for(i=0;i<MAXROW;i++)
        for(j=0;j<MAXCOLUMN;j++)
            fread(&table[i][j],sizeof(int),1,tr);
    fclose(tr);
    return(0);
} /* End of Function LoadTable */

/* Save data table to disk */
int SaveTable(char * fileName, int table[MAXROW][MAXCOLUMN])
{
    int i,j;
    FILE * tr;
tr = fopen(fileName,"w");
if (tr == NULL)
    return -1;
for(i=0;i<MAXROW;i++)
    for(j=0;j<MAXCOLUMN;j++)
        fwrite(&table[i][j],sizeof(int),1,tr);
fclose(tr);
return(0);
} /* End of Function SaveTable */

/* Display table data on screen */
void DisplayTable(int table[MAXROW][MAXCOLUMN])
{
    int i;

    for(i=0;i<MAXROW;i++)
        printf("%d %d %d\n",table[i][0],table[i][1],table[i][2]);
} /* End of Function DisplayTable */

/* Modify data in table */
void ModifyTable(int table[MAXROW][MAXCOLUMN])
{
    int i,j;

    printf("Row number:");
    scanf("%d",&i);
    printf("Column number:");
    scanf("%d",&j);
    printf("Cell value:");
    scanf("%d",&table[i][j]);
} /* End of Function ModifyTable */
**EpsInit()**

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>malloc(heap) /*Allocate prg. Heap*/</code> Send (NPS,AcM, CaV, prgTableName, Result, Reason,LoV)</td>
<td>Receive (COB,AcM, CaV, prgTableName, Result, Reason, IP) /* load prg table*/ Search(inUse, PeD) if not found then Request(OBS, Load, CaV, PrgTableName,Result, Reason,IP) Else Send (COB,AcM, CaV, prgTableName, NotAvailable, Used,IP) End if</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receive (NPS, RMs) Receive (NPS,Load, CaV, prgTableName, Result, Reason, IP) Load (LoV, PeD) If Loaded then Send (NPS,AcM, CaV, prgTableName,</td>
<td></td>
</tr>
<tr>
<td>Receive (OBS,Load, CaV, prgTableName, Success, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add(inUse,PeD info)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive (OBS,Load, CaV, prgTableName, Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create (session parameters)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return(Ams)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**EpsClose(Commit/Rollback)**

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EpsCommit/EpsRollback</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free(heap)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return(Ams)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**EpsRollback()**

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rollback(Local Heap)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>While</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (NPS,RollBack, CaV, OP, Result, Reason, IP) (1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) Receive (AOB,Rollback, CaV, OP, Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End while</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive (COB,Rollback, CaV, OP, Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search(InUse, PeD)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If Check(CaV) then</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (COB,RollBack, CaV, OP, Fail, NotAuthorized,IP) (2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If found then</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (AOB,Rollback, CaV, OP, Result, Reason,IP) (3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request (OBS, Load, CaV, AOB, Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (AOB,Rollback, CaV, PeD,</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) Receive (NPS,Rollback, CaV, OP, Result, Reason, IP)
Rollback(persistent Object)
Send (COB,RollBack, CaV, OP, Success, Reason,IP)
<table>
<thead>
<tr>
<th>Result, Reason, IP</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>End if</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive (NPS, Load, CaV, OP, Result, Reason, IP)</td>
<td></td>
</tr>
<tr>
<td>Load (LoV, PeD)</td>
<td></td>
</tr>
<tr>
<td>If Loaded then</td>
<td></td>
</tr>
<tr>
<td>Send (NPS, Load, CaV, OP, Loaded, Reason, PeD)</td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
<tr>
<td>Send (NPS, Load, CaV, OP, NotLoaded, Reason, IP)</td>
<td></td>
</tr>
<tr>
<td>Add (inUse, PeD info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive (NPS, Rollback, CaV, OP, Result, Reason, IP)</td>
<td></td>
</tr>
<tr>
<td>Rollback(persistent Object)</td>
<td></td>
</tr>
<tr>
<td>Send (COB, RollBack, CaV, OP, Success, Reason, IP)</td>
<td></td>
</tr>
<tr>
<td>Return (Ams)</td>
<td></td>
</tr>
</tbody>
</table>
**EpsCommit()**

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit(Local Heap)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>While</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (NPS, Commit, CaV, OP,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result, Reason, IP) (1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) Receive (AOB, Commit, CaV, OP,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End while</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive (COB, Commit, CaV, OP,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search(InUse, Pd)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If Check(CaV) then</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (COB, Commit, CaV,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP, Fail, NotAuthorized,IP) (2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If found then</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (AOB, Commit, CaV, OP, Result,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reason, IP) (3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request (OBS, Load, CaV, AOB,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (AOB, Rollback, CaV, Pd)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive (NPS, Commit, CaV, OP,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit (persistent Object)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (COB, Commit, CaV, OP, Success,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Result, Reason, IP</strong> (4)</td>
<td><strong>End if</strong></td>
<td><strong>Receive (NPS, Load, CaV, OP, Result, Reason, IP)</strong></td>
<td><strong>Load (LoV, PeD)</strong></td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Send (NPS, Load, CaV, OP, Loaded, Reason, PeD)</strong></td>
<td><strong>Else</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Send (NPS, Load, CaV, OP, NotLoaded, Reason, IP)</strong></td>
</tr>
<tr>
<td><strong>Add(inUse, PeD info)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Receive (NPS, Commit, CaV, OP, Result, Reason, IP)</strong></td>
<td><strong>Commit(persistent Object)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Send (COB, Commit, CaV, OP, Success, Reason, IP)</strong></td>
</tr>
</tbody>
</table>
LoadPersistentObject(LoV, DBN, AcM, CaV)

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
</table>
| Send (NPS, Load, CaV, DBN, Result, Reason, LoV) | Receive (COB, Load, CaV, OP, Result, Reason, IP)  
Search(inUse, PeD)  
if not found then  
Send (OBS, Load, CaV, DBN, Result, Reason, IP)  
Else  
Send (COB, AcM, CaV, DBN, NotAvailable, Used, IP)  
End if | Receive (NPS, Load, CaV, DBN, Result, Reason, IP)  
Load (LoV, PeD)  
If Loaded then  
Send (NPS, Load, CaV, DBN, Success, Reason, PeD)  
End if  
Send (COB, AcM, CaV, DBN, |
<table>
<thead>
<tr>
<th>Action</th>
<th>Result, Reason, PeD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive (OBS, Load, CaV, DBN, Result, Reason, PeD) Add(inUse, PeD info)</td>
<td></td>
</tr>
<tr>
<td>Receive (OBS, Load, CaV, DBN, Result, Reason, LoV) Return(Ams)</td>
<td></td>
</tr>
</tbody>
</table>

103
### ReleasePersistentObject(LoV, Commit/Rollback/None)

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If parameter2 = Commit then</td>
<td>Commit(LoV)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Else if parameter2 = Rollback then</td>
<td>Rollback(LoV)</td>
<td>Receive (COB, Commit/Rollback, CaV, PeD, Result, Reason, IP)</td>
<td>Delete(inUse, PeD)</td>
</tr>
<tr>
<td>Request (NPS, Commit/Rollback, CaV, LoV, Result, Reason, IP)</td>
<td></td>
<td></td>
<td>Send (COB, Release, CaV, PeD, Success, Reason, PeD)</td>
</tr>
<tr>
<td>Free(LoV)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return(Ams)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client Object Library (COL)</td>
<td>Naming and Protection Server (NPS)</td>
<td>Object Server (OBS)</td>
<td>Active Object (AOB)</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------</td>
<td>---------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Send (NPS, Ops, CaV, AOB, Result, Reason, IP)</td>
<td>Receive (COB, Ops, CaV, OP, Result, Reason, IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Search(inUse, PeD)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if not found then</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Request (OBS, Load, CaV, AOB, Result, Reason, IP) (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add(inUse, AOB info)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Else</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send(AOB, Ops, CaV, OP, Result, Reason, IP)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| (1) | Receive (NPS, Load, CaV, OP, Result, Reason, IP) | | |
| | Load (AOB, PeD) | | |
| | If Loaded then | | |
| | Send(NPS, Load, CaV, OP, Success, Reason, PeD) | | |
| | Else | | |
|                           | Send(NPS, Load, CaV, OP, Fail, Reason, IP) | Receive (NPS, Ops, CaV, OP, Result, Reason, IP)  
|                           |                                           | Do Requested operation  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Send (COB, OP, CaV, OP, Result, Reason, IP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive (AOB, Ops, CaV, OP, Result, Reason, IP)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TableRead(tableName, DON, LoV)

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetTableRow(ScRefKey, tableName)</td>
<td>Return(value)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TableWrite(tableName, DON, LoV)

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PutTableRow(ScRefKey, tableName)</td>
<td>Return(value)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TableDelete(tableName, DON)

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteTableRow(ScRefKey, tableName)</td>
<td>Return(value)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CapabilityTransfer(AOB, CaV, Limit)

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request (NPS, Transfer, CaV, AOB, Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Receive (COB, Transfer, CaV, AOB  
| Result, Reason, IP)  
| Search(inUse, PeD)  
| if not found then  
| Request (OBS, Load, CaV, AOB,  
| Result, Reason, IP) (1)  
| Add(inUse, AOB info)  
| Else  
| Send (AOB, Transfer, CaV, OP,  
| Result, Reason, IP) |

| (1)  
| Receive (NPS, Load, CaV, OP, Result,  
| Reason, IP)  
| Load (AOB, PeD)  
| If Loaded then  
| Send(NPS, Load, CaV, OP, Success,  
| Reason, PeD)  
| Else  
| Send (NPS, Load, CaV, OP,  
| Fail, Reason, IP) |

| Receive (NPS, Transfer, CaV, OP,  
| Result, Reason, IP)  
| Add capability to the available capability list |
| Receive (AOB, Transfer, CaV, OP, Result, Reason, IP) |  | Send (COB, OP, CaV, OP, Result, Reason, IP) |
### RightsCancel(AOB, CaV, Parameters)

<table>
<thead>
<tr>
<th>Client Object Library (COL)</th>
<th>Naming and Protection Server (NPS)</th>
<th>Object Server (OBS)</th>
<th>Active Object (AOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request (NPS, Cancel, CaV, CaV, Result, Reason, IP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Receive (COB, Cancel, CaV, CaV Result, Reason, IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Search(InUse, PeD)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if not found then</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Request (OBS, Load, CaV, AOB, Result, Reason, IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add(InUse, AOB info)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Else</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modify (InUse, CaV, Cancel)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send (COB, Cancel, CaV, CaV, Result, Reason, IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Receive (NPS, Load, CaV, OP, Result, Reason, IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load (AOB, PeD)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If Loaded then</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send (NPS, Load, CaV, OP, Success, Reason, PeD)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td>Send (NPS, Load, CaV, OP, Fail, Reason, IP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RESUME

Erdal Kemikli was born in Yatağan, Muğla in 1967.

He has a B.Sc. degree in Computer Engineering from Boğaziçi University Istanbul, and M.Sc. Degree in Computer Science and Engineering from the Institute of Science and Technology, Ege University, Izmir. He is currently a Ph.D. candidate on Computer and Control Engineering in Institute of Science and Technology, Istanbul Technical University, Istanbul.

Throughout his career, he has worked as research assistant, programmer, software designer, software quality manager, and information systems department manager.