INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Persistent Object System

Ba-Nguyen Tran

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 1997

ABSTRACT

Persistent Object System

Ba-Nguyen Tran

A persistent object system (PO-system) is a system used for storing and retrieving persistent objects. Such systems already exist, but they are programming language and database dependent and, in many cases, difficult to use. We have designed and implemented a persistent object system which should be simple, inexpensive and easy to use.

A client/server networking model was chosen to implement the system to make it independent of the applications' platforms and programming languages. The implementation of the system is written in the programming languague C on the UNIX platform. PO-system provides a server called PO-server. An application program uses the Application Programming Interface (API) to communicate requests to the PO-server. In this way, the application program can obtain persistent object services without requiring any knowledge of the underlying mechanism of the PO-system.

Acknowledgments

The author wishes to express the deepest gratitude to his supervisor, Dr. Peter Grogono, whose supervision and encouragement has led this work to a successful outcome, an educational and pleasant experience for the author.

The author would like to thank his friend, Loc-Nguyen Vuong, who has participated in discussions to stimulate the interest of the project and has taken responsibility for a part of the project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Concept of Persistence and Pickling

1.1.1 Persistent Data and Persistent Objects

Data created by a program are sometimes needed when the program is not executing. We call them persistent data. Hence, persistent data are data that exist beyond the lifetime of the application programs that create or manipulate them.

In object-oriented programming, data are grouped into objects. We have a similar concept of persistence applying to objects. A persistent object is an object whose existence exceeds the lifetime of the application programs that create or manipulate it. It is important to note that we use the term "object" to refer to a logical entity that consists of data attributes and methods that can manipulate the attributes. However, when we use the term "persistent object", we refer to only the data attributes of the

object.

1.1.2 Pickling System

The term *pickling system* was used in an article written by Daniel H. Craft, [Cra93], to refer to a persistent data or object system. We (Loc-Nguyen Vuong and I) found this article very interesting, and it was the starting point of our project. In the project and my thesis, the name *Persistent Object system* (PO-system) has been used to refer to our *Pickling system.*

1.2 Outline of the Thesis

- Chapter 1, *Introduction*, contains a brief introduction to the concepts of persistence and pickling, the outline of this thesis and the motivation of the project.

- Chapter 2, *Background*, provides background for our design and implementation of a persistent object system. This chapter includes also a brief historical aspect of the evolution of the software development, the need of persistence, and the surveys of some related works.

- Chapter 3, *Solution*, describes our solution for persistent objects. This chapter contains the requirements and the designs of a persistent object system, the design of an application programming interface (API) of the system, and sample application programs that cover the most general data structures used in application programs: simple class structure, inherited class structure, linked

(acyclic) structure, and cyclic-structure. This chapter also discusses a general approach to store and retrieve linked structure and cyclic structure persistent objects in sequential storage media or data stream.

- Chapter 4, *Implementation*, describes the implementation of the system. It includes details of the communication specific choices, application communication headers, main routines of the server, the API and the sample application programs. Because the system and API was written in C programming language on UNIX platform and the sample application programs were written in C++ programming language, the content of the chapter relates to C and C++ on UNIX. Some design details are also discussed in this chapter.

- Chapter 5, *Assessment*, discusses the success, failure and limitations of the project verifying with testing results.

- Chapter 6, *Conclusion and Further Work*, summarizes my work on the project, and the advantages and disadvantages of the usage of PO-system to the end-users. Finally, a list of further work is suggested for the improvement of PO-system.

1.3 Motivation of Project

The primary goal of the project was to introduce a simple and friendly solution for the enhancement of existing programs to store and retrieve their persistent objects, and

hence to reduce software maintenance cost. I have designed and implemented most parts of the system. During the implementation and testing phases, I kept revising my design to make the product simpler, more efficient, and also to get around difficulties, or to eliminate some inconsistencies.

Although the main goal of the project was on the persistent object system, I have been concerned, equally to the system's implementation, about the usefulness of the system to its end-users. The major difficulties that an application faces while working with persistent object usually include the storage and retrieval of objects that contain pointers to other objects. Such objects are usually part of the representation of a list, tree, or other data structures that may include cycles. So part of my research was on the techniques and algorithms that an application program can easily apply in order to use the system.

Chapter 2

Background

This chapter provides background on some subjects relating to the project. It includes a brief introduction of historical aspects of software development, object-oriented programming, and client/server networking model, the need for persistence and some existing solutions.

2.1 Historical Aspect

Computer science has been evolving rapidly during the last few decades. Its applications spread widely, deeply and quickly, all over the entire spectrum of research areas and professions. The discovery of structured programming technique promised the capability of building complicated programs and reusing modules [Ses96].

However, the maintenance cost for software constructed using the structured programming paradigm is high, because a minor modification may cause the need of

revision of many modules [Pit97]. A new programming technique, *object-oriented programming*, was introduced to avoid problems of this kind. In object-oriented programming, data are encapsulated together with methods that can be applied to manipulate those data in a logical entity, called an object. Modifications or enhancements to an object-oriented program can be made at the level of objects.

Orthogonal to programming, persistence is a requirement of some software. It has nothing to do with the program's code. It is the requirement that some data of the program continue to exist after the program has terminated.

Parallel to the demand of the development of new and powerful programming languages and applications, communication between machines has become more and more important. Computer networking was born and evolved rapidly to fulfill the demand for machine communication [Mil91]. It is just another dimension of development of computer technology.

2.2 Object-Oriented Programming

Object-oriented programming techniques promise high reusability and reduce maintenance cost by combining the best ideas of structured programing with several powerful new concepts: *object encapsulation*, *inheritance* and *polymorphism* [Sch90].

2.2.1 Object Encapsulation

A class defines a scope within which the attributes of an object are declared. It provides also means for manipulating some of these attributes outside of the class. An object is an instance of a class [Gro91]. In other words, an object is a logical entity containing data attributes and methods that manipulate those attributes. Objects are natural to use to model the real world, because the real world's entities contain both data and functions that manipulate on the data. For example, an object, *student*, can have attribute *address* and a function to alter *address*; an object, *professor*, can have attribute *office* and a function to change the *office*.

2.2.2 Inheritance

In an object-oriented program, a class can be declared to inherit another class. A new class inheriting an existing class will inherit its parent class' properties plus its own. Hence object-oriented programming promotes reusability.

2.2.3 Polymorphism

Polymorphism of object-oriented programming languages allow a name to be used for different methods in classes, and depending on the *type* of the data, a specific instance assigned by that name will be invoked. This property makes the enhancement of a object-oriented program much simpler.

The polymorphic property in object-oriented programming languages is not just

function overloading, it is a consequence of dynamic binding [VC90]. Some functional languages, such as Ada, have function overloading feature, but function calls are bound at compile time. For object-oriented programming languages, dynamic binding allows objects to be bound to their functions at run time. Dynamic binding combines with polymorphism in object-oriented programming languages is a wonderful tool for software reusability and enhancements, because it allows us to build class libraries easily. For example, we can have classes `Menu1` , `Menu2` derived from class `Window`. Classes `Window`, `Menu1` and `Menu2`, each defines a method (function) `display()` of their own. A polymorphic pointer `pol_ptr` is declared as a pointer of class `Window`. Latter in the program, `pol_ptr` can be used to point to object *menu1* of `Menu1` or `menu2` of `Menu2`, and then invoke `display()`. At run time, depending on the object that `pol_ptr` is pointing to, when invoking `display()` from `pol_ptr` with the call `pol_ptr->display()`, the proper method will be invoked.

2.3 Client/Server Networking Model

Computer networking is the interprocess communication among processes running on different machines. Therefore the fundamental entity of computer networking is process [Ste90].

The client/server networking model or simply client/server model is an architectural model for distributing applications. In the client/server model, processes are divided into two groups, clients and servers. A server is a process that waits for

requests sent from clients, performs operations (services) requested, and then replies or responds to the clients. A client, on the other hand, is a process that may generate and send requests to a server.

The client/server model is useful for applications where resource-sharing or information sharing is required, such as when a printer needs to be shared among several work-stations, or a database is shared between several banking-machines.

2.4 Persistence

2.4.1 The Need for Persistence

In practice, many application programs require data to exist beyond their current executions. For example, let us say, research on the Fibonacci series is carried out by running a program with an infinite-loop of computation until some specified condition becomes true. After running the program for a few days, the system crashes for some reason. If the computational results were persistent data, then when the system is re-booted, the researcher's program can continue its search without having to start from scratch. Another example of persistent data is a banking account. When Mr. A goes to a bank to open an account and deposit $500, a teller uses the bank's program to create an account and deposit $500 for Mr. A. Thereafter, no matter what happens to the bank's system, the existence of an account with $500 balance for Mr. A in that bank has to be kept.

2.4.2 Persistent-Objects

In object-oriented programming, data are grouped into objects. Hence, persistence of data in object-oriented programs is provided on the basis of objects. A new application programming domain implies new requirements. An object may belong to a class that inherits from other classes (its parent classes) and its parent classes may have data attributes too. Furthermore, an object can have pointer attributes to another objects, which in turn may have a pointer attribute that points back to the original object and forms a cyclic structure. The storing and retrieving of objects having pointer attributes can be difficult. For example, in a linked list, object a has a pointer to object a1, object a1 has a pointer to object a2, and so on. When we want to store object a1, if we store its pointer to a2 as reference (address of a2), then when we retrieve a1, the referenced address may have been reallocated to another object. If we store a1 and the object a2 pointed by the pointer of a1, we still have the same problem with a2, because a2 has a pointer to a3, and so on. Eventually, storing an object in a linked list will result in storing the entire linked list. It is even more difficult to store and retrieve cyclic structures.

2.5 Solutions

Solutions for persistent-objects have become an interesting research topic and development. Many of them have been implemented. Here are some examples of those solutions.

2.5.1 Object-Oriented Database Systems

There exist some object-oriented database languages and systems that provides support for persistent objects. GemStone, for example, is an object-oriented database system for SmallTalk-80, ODE system and Exodus system are database systems for C++. GemStone adds to Smalltalk persistent object management facilities and other database features such as concurrency control, transactions, etc., to handle persistent objects. ODE system defines a database for database programming language O++, and O++ is an extended version of C++. Exodus has a component, E language, that is also a version of C++ programming language used for writing software to support persistent applications [Hug91].

2.5.2 Pickling System

A pickling system is a project aiming to "lightweight, easily comprehensible, 80% solution of persistent data need" . The project proposes a scheme to traverse and externalize data structures at run-time using type information available to the garbage collector [Cra93]. A data structure is externalized by executing corresponding pickling codes depending on individual data types. Figure 2.1 shows an example of such a scheme [1].

The image is an external representation of an object. It will be stored on disk or passed to a remote program. Type checking is done at compile time to ensure data

[1] This example is simplified from an example in the research paper.

```
(a) internal data

PROFILE

name    : Dan Craft
phone   : 1234567
...
```

```
(b) code executed for pickling

 pickle PROFILE

 pickle STRING (name)
 pickle INT    (phone)
```

```
(c) Image resulting from pickling
data$"Dan Craft" data$1234567 ...
```

Figure 2.1: The externalization and image resulting from the pickling system

accessed to be placed correctly as they appear for internal format. It requires also run-time type-checking to ensure that unpickling performs correctly. For abstract data types, the programmers have to augment their abstract types with two routines (methods): encode/decode for each type. In general, a pickling system requires good programming environment support, especially a type-safe language.

2.6 Our Solution

The solutions mentioned above are computer-language dependent, not easy to use, and hence not really suitable to use for the enhancement of existing applications. They are formal approaches to the solutions that are not concerned with the system's construction. On the other hand, we are interested in developing a simple persistent object system that would be easy and cheap to use for object-oriented applications. The system is neither a new database, nor requires a new language, but is a new

technique toward an inexpensive solution for persistent objects.

Chapter 3

Persistent Object System

We started with the idea of building up a persistent object system of our own, that would probably be useful and cost less for its users in terms of maintenance. The project of building up *a Persistent Object System* (PO-system)[1] is started by setting out a brief requirement list.

The project is intended for object-oriented application programmers as its primary users. There were two reasons for this choice. Firstly, we wanted to narrow down our research topic, since the project should not be too big for two master students. Secondly object-oriented application programs were much easier to enhance than non-object-oriented applications, and one of our goals was to promote an easy way to enhance existing programs to make use of our system.

[1] We called it Pickling system (PK-system) at the beginning of the project.

3.1 Requirements

1. Language and platform independent design: The design of the system should be *independent of programming languages and platforms.*

2. Type safety: The system should do type-checking somehow to guarantee *type safety* for persistent objects.

3. Efficiency: The performance of the system should be *reasonably efficient.* The resource (main memory and secondary storage medium) should not be unreasonably wasted.

4. User-friendliness: The application programmers should be able to use it with *minimum amount of learning effort.*

5. Security: The system should include some security features to protect its objects against unauthorized accesses.

6. Consistency: The system should never be in an inconsistent state. If a persistent object had been succesfully stored and followed by a successful retrieval operation of the same object, then the retrieval operation has to get back exactly the object that was stored.

7. Data orientation of the applications: The system is designed for object-oriented applications. It should mainly focus on object-oriented application users and show that object-oriented application programs can significantly benefit from the system.

3.2 Discussion of Requirements

3.2.1 Platform and Computer Language Independency

It is fine for the design of the system to be platform and computer language independent. However, for example, if a system following the design is implemented in language A on platform B, then the question is "will the implemented system be able to function on a different platform, let us say C, where the application programs are written in D ?". The Client/Server network model gives us a "YES" answer to this question. If we consider the tasks of the system as somebody's job, then his/her role in the system is the server and each running application program which requires services from the system is a client of the server. Therefore the heart of the system is the server, and the server does not care where its clients are running, and in what languages they were coded.

The client/server model introduces concurrency that not only improves the efficiency of the system but also increases the complexity of the system. The server is capable of serving several clients running concurrently. It also has to take care of concurrency concerns such as concurrency-control, data consistency, etc.

Therefore, the PO-system will have a server as the heart of the system, that provides services to its clients. The services include all services needed for persistent object storage and retrieval. The server is called *the persistent object server*, or PO-server for short.

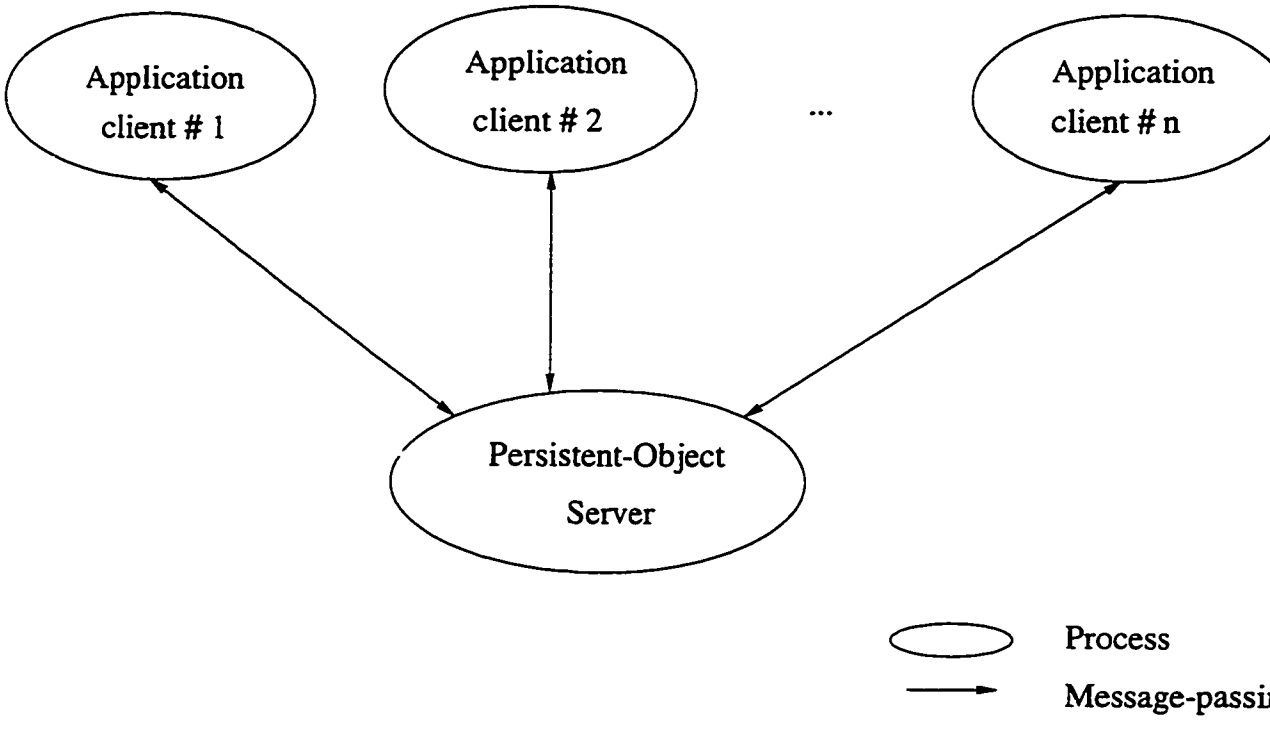


Figure 3.1: Client/server Model of PO-system.

3.2.2 Type Safety

In client/server architecture, the server and its clients may operate on different platforms, hence type checking should be done by a special client, called an *administrative client* of the system or AdminClient in short, which runs on the applications' platform and provides local service for type-checking of local clients and *clients' registrations* to the server in order to ensure the availability of the server's resources before the application clients start running. Therefore, each platform needs an administrative client.

3.2.3 Efficiency

Efficiency is measured against two criteria: speed and space. In general, algorithms can satisfy either one of them but not both. However, we are not deeply concerned about this issue, because it is not a major goal for the project. As long as we can show that the system has good mechanisms to move data forth and back between main memory and secondary storage medium, using dynamic memory allocation scheme for big chunks of data storage for space efficiency and using tables for static information storage for fast access, this requirement will be satisfied.

3.2.4 Security

The system has to preserve somehow the security of its persistent objects against unauthorized accesses and modification. In other words, PO-system has to include features that implement security checks against unauthorized accesses of persistent objects under its protection. One way to do this is to implement a security mechanism on the server as shown in Figure 3.2. This mechanism is used for:

- applying object-authorization: each object has an owner, and only the owner of an object has full-access rights to the object, other clients are restricted to read-only access through the server;
- applying encryption/decryption mechanism to avoid by-pass server accesses of objects of some clients (read directly from the storage medium); and
- applying user-password check for clients.

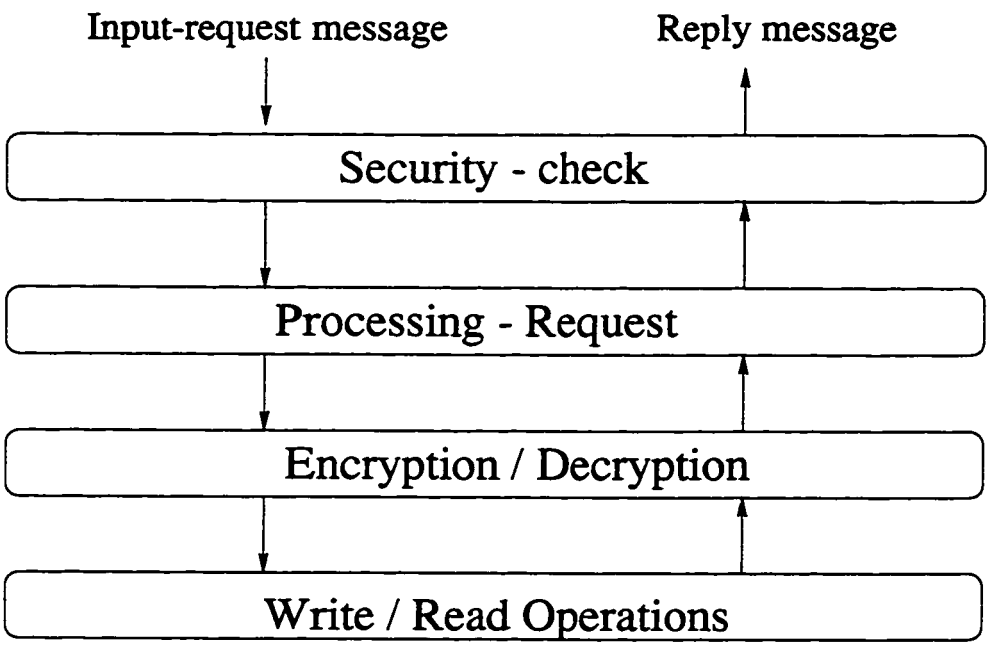


Figure 3.2: PO-server's Security.

In this way, if a client wants to access a persistent object, it cannot do so by reading the object directly from the secondary storage, because the encryption, and format of the object in there is known only to the server. This scheme enforces the accesses of persistent objects of the system to be controlled by the server.

3.2.5 Consistency

In order to deal with concurrent accesses of objects, the system has to preserve the consistency of objects under its protection.

To clarify the problem, let's take a simple example. An object named `Obj` of class `Cls`. can be accessed by program `Clt1` and program `Clt2`. `Obj` contains a `string` called `name`, and an `integer` named `number`. `Cls` has a method `get_number()` to output the object's member `number`, a method `increase_number_by(i)` to increase

number by i, and a method decrease_number_by(i) to decrease number by i. Assume the original value of Obj.number is zero. Programs Clt1 and Clt2 has been coded as following:

```
program Clt1
   ...
   b = Obj.get_number();
   ...
   Obj.increase_number_by (10);
   ...
   Obj.show_number();
```

```
program Clt2
   ...
   c = Obj.get_number();
   ...
   Obj.decrease_number_by (5);
   ...
   Obj.show_number();
```

The result expected for statement Ojb.show_number in Clt1 is $(b+10)$, and that in Clt2 is $(c-5)$. However, the results seen in Clt1 and Clt2 after running statement

`Obj.show_number()` may be different from what each individually expected if both programs run simultaneously. For example, the object is accessed by two programs in the order shown below:

step	Clt 1	Clt2
1	`b = Obj.get_number`	
2		`c = Obj.get_number`
3	`Obj.increase_number_by(10)`	
4		`Obj.decrease_number(5)`
5	`Obj.show_number()`	
6		`Obj.show_number()`

For client `Clt1`, `b` is zero, so the expected number shown by method `show_number()` is 10, but the result is 5. For client `Clt2`, `c` is zero, and hence the expected number is -5, but the real result is 5. One way to solve this problem is by `serialization` [Alm94]. However, we can serialize only two transactions or two atomic operations from two programs, but not two programs' executions, otherwise the parallelism of the system is destroyed. Unfortunately, for our case, the serialization of transactions does not give us a satisfactory solution. The example above does the same as the transaction serialization if we consider each method invocation is a transaction or an atomic operation on the object, but the results are not as expected. The reason is because our system works differently from that of a typical banking database. In

a typical transaction, a record is locked when it is accessed to ensure the consistency of the database, but for us, in order to do so, we have to serialize executions of the application programs, because we have no prior knowledge of the application programs.

For simplicity, we can solve the problem by setting out our own system restrictions:

1. Only one process is allowed to run a program at a time.

2. Each object has an owner, who is the client who creates the object.

3. Only the owner of an object is allowed to modify the object.

These restrictions should be enforced by the system itself as much as possible, not by the obedience of application users alone.

3.2.6 User-Friendliness

A simple *Application Programing Interface* (API) should be provided to minimize the application users' task as much as possible when using the system. Hence the API will be provided as a set of high level procedures to maximize information hiding and user-friendliness. A friendly menu should be provided for client-registration too.

3.2.7 Data Orientation of the Applications

Persistent data itself has nothing to do with the data orientation of the applications (object-oriented or non-object-oriented). However, the use of the persistent object system is related to data within a logical entity, which is more naturally related to

the object-oriented data model than the non-object-oriented data model. For existing big programs, involving this persistent object system relates to the extensibility of the programs on persistent data entities. This kind of extensibility is simple and natural for an object-oriented program; for a structural program, on the other hand, the extensions may require elaborate modifications.

3.3 The Partition of Project

The PO-system consists of two main components: a persistent object server and a persistent administrative client. The project also includes the implementation of a PO-system application programming interface and sample applications. The project was carried out by two master students: Loc-Nguyen Vuong and me. It was partitioned into two parts, as shown in Figure 3.3, such that each one could carried out and finised his own without waiting for the completion of the other.

3.3.1 Persistent Object Server

PO-server is the heart of the system. Its main tasks are:

- to provide service to Administrative clients;
- to provide service to Application clients;
- to manage local resources; and
- to provide security for objects under its protection.

Loc-Nguyen Vuong	Ba-Nguyen Tran (me)
Design and Implementation of: - AdminClient and - all its subcomponents.	Design and Implementation of: - PO-server, - API, - sample applications, and - sample AdminClient.

Figure 3.3: The partition of the project.

3.3.2 Persistent Object Administrative Client

AdminClient is a local platform manager for the PO-system. It has to accomplish the following main tasks:

- providing a means by which application clients can input client-registration information to the server;
- providing type-checking for local application clients; and
- sending client-registration information to the server and interpreting the return-code from the server.

3.3.3 API and Sample Applications

Application programming interface and sample applications are not part of the system. They are needed as the complement part of the project. They are:

- a set of application programming interface modules (API); and
- sample application programs that cover general cases of data structures in object-oriented programming.

3.4 Design

This section discusses only the preliminary design of the system, leaving some details in Chapter 4, where the design details are somehow affected by the implementation specific choices, such as network protocol, platform, and computer language.

3.4.1 Server

The algorithm for the main loop of the server is shown in 3.4, which consists of the following main modules:

- process_admin_request():
- process_register_object_request();
- process_retrieve_object_request();
- process_store_object_request();
- process_client_terminate_request();

```
Program POserver:

  open communication channels
  initialization of global data structures

  while NOT shutdownCommand OR
        there are application clients in execution state.

        wait for a coming message
        if the sender is  AdminClient
               check adminPassword
               if password is correct
                  reply_code = process_admin_request
               else
                  reply_code = MSG_ADMIN_PASSWORD_ERR

               send response to AdminClient
        else
               if (message_id == REGISTER_ID)
                    reply_code = process_register_object_request
               else if (message id = RETRIEVE_ID)
                    reply_code = process_retrieve_object_request
               else if (message id == STORE_ID)
                    reply_code = process_store_object_request
               else if (message id == TERMINATE_ID)
                    reply_code = process_client_terminate_request
               else
                    reply_code = UNKNOWN_MESSAGE_ID

               send response to the sender
```

Figure 3.4: General algorithm of PO-server's main-loop.

```
Program AdminClient:

  while not shut_down_command
     display menu
     get inputs
     if input_command == shut_down
        send shut_down_command to the server
        exit
     else
       /* should have done the local type-checking here*/
        send client registration to the server
        interpret reply code from the server
```

Figure 3.5: Algorithm for sample AdminClient.

3.4.2 Adminstrative Client

Figure 3.5 shows the algorithm used to implement a simple administrative client sample. This sample AdminClient does not do the local type checking and displays only a simple menu to get user's inputs. It has been provided for testing purpose only. A complete administrative client will be designed and implemented by Loc-Nguyen Vuong.

3.4.3 Application Programming Interface (API)

The application programming interface consists of the following functions. The API is computer language dependent. Here we provide a set of C language declarations of the API functions. It should be noted that the provision of this functional interface does not violate the first requirement of the system, *computer language independence*, because the main components of PO-system consists of only PO-server and

AdminClient.

```
int PK_register_obj (char *object_owner,
                     char *class_name,
                     char *object_name,
                     char *buffer,
                     int  *send_receive_length);

int PK_retrieve_obj(char *object_owner,
                     char *class_name,
                     char *object_name,
                     char *buffer,
                     int  *send_receive_length);
int PK_store_obj   (char *object_owner,
                     char *class_name,
                     char *object_name,
                     char *buffer,
                     int  *send_receive_length);
int PK_terminate     (char *cltname);
```

Before an application client can use the PO-system, it has to register itself with the PO-server through an administrative client. Before it terminates, it has to inform

the server through the `PK_terminate()` function call.

Before an application client can store an object to or retrieve an object from the server, it has to register the object with the server. The actions of registering object, storing object, and retrieving object can be done by making function calls to `PK_register_obj`, `PK_store_obj`, and `PK_retrieve_obj` respectively.

3.4.4 Sample Application Programs

The sample application programs are simple programs that output neccessary traces to illustrate the results for verifying the correctness of the system.

The general algorithm for the application programs used for testing are shown in Figure 3.6.

The `show` method is not new to object-oriented application programmers. Only the `encode` method and `decode` method are new tasks for them when making use of the PO-system. The `encode` method will write all data of an object in a buffer, in some format, and `decode` method will do the reverse. These two methods are also the only new methods for the enhancement of an object-oriented program to integrate with PO-system. Users can choose any format and algorithm for the implementation of these two methods. However, I will discuss my solutions in the next section.

In order to provide testing application programs that will cover the most general cases for object-oriented applications, we have classified objects' data structures into four categories:

1. *Simple data structure objects* are objects of a class that does not inherit from

```
Application program:

  initialize the object ( either with a constructor or not)
  invoke  object.show  to show the initial objected
  invoke object.encode to translate data of the object into a stream
                        of data in a buffer
  invoke PK_register_obj to register the object
  invoke PK_retrieve_obj to retrieve the object
  invoke object.decode to decode data in the buffer received
  invoke object.show to show the retrieved object
         compare the result with the initial one.
  ...
  make some change for the object
  invoke object.show
  invoke object.encode
  invoke PK_restore_obj  to store the object
  invoke PK_terminate
       ...
```

Figure 3.6: Algorithm for sample application programs.

any other class and do not have pointers to other objects.

2. *Inherited data structure objects* are objects of a class that inherits from at least one parent class, and do not have pointers to other objects.

3. *Linked structure objects* are objects that contain pointers to other objects, such as a linked list or tree structure.

4. *Cyclic structure objects* are objects that contain pointers to other objects, and the child objects in turn are allowed having their pointers back to the parent objects. It is the pointers back to the parent object that form the cyclic structure for the objects.

According to this classification, we need at least four sample application programs, one for each criterion. A class in each criterion should be declared and include the implementation of some methods as shown below:

1. for simple objects:

```
Declaration:
  Declaring a class including the following methods:
    show   : to display data of object of the class.
    encode : to encode the object to a stream of data.
    decode : to decode a stream of data to an object.
```

2. for inherited objects:

```
Declaration:
  Declaring a parent class to include the following methods:
    show   : show object belongs to the class.
    encode : encode the object to a stream of data.
    decode : decode a stream of data to an object.

  Declaring the child class to include methods as follows:
    show should call the show method of the parent class and then
         display the child class's specific data.
    encode should call the encode method of the parent class and
```

then encode the child class's specific data

decode should call the decode method of the parent class and then decode the child class's specific data

3. for linked structure objects:

Declaration:

Declaring a binary tree class to include the following methods:

show : to display the tree object starting from the root of the class

encode : to encode the object to a stream of data.

decode : to decode a stream of data to an object.

4. for cyclic-structure objects:

Declaration: We can construct a cyclic structured object by reusing the tree declaration and direct some pointer of some successor back to their ancestors.

Declaring a cyclic-tree class to include the following methods:

show : to display the cyclic object starting from the root of the class.

encode : to encode the cyclic object to a stream of data.

decode : to decode a stream of data to a cyclic object.

3.5 A Solution for Encoding/Decoding Methods

3.5.1 Non-Recursive Structure Objects

The encode method transforms the data of object into linear format and the decode method does the reverse. It does not matter what format we choose for the encode method, as long as we can guarantee the decode method will give us back the object with correct data.

Example: An object of some class has the following data:

```
wheels      : 4
passengers  : 6
name        : HONDA
```

The object can be matched to a buffer without data field separators as `4 6 HONDA`. It can also be matched to a buffer using commas as separators between data fields like: `4, 6, HONDA`; and so on. Depending on the object's data structure, a user can always choose an appropriate linear format.

3.5.2 Recursive Structure Objects

The major difficulty for the PO-system users is how to select an appropriate linear format for recursive-structured objects and so obtain the algorithms to implement `encode/decode` methods. Should we store the pointer to the object or the object referenced by pointer? The former is not suitable, because the address of an object is

dynamic and subject to change for different runs of a program. The latter technique has the major disadvantage of the possibility of working with a huge amount of data unneccessarily. For example, the storage of one single object at a node of a binary tree may result in the storage of the entire tree. Only the programmers know what they need, a node or a tree, and hence only the programmers can code and invoke appropriate methods. A good algorithm of traversal of a recursive structure containing cycles is needed for implementing both encode and decode methods, and an appropriate format of the storage of recursively structured objects is not trivial either.

Cyclic-Structure Traversal

The implementation of a cyclic structure traversal algorithm shown in Figure 3.7, requires the implementation of set `setOfNode`. This traversal algorithm will traverse a recursive structure and visit each node exactly once. Since the `setOfNode` here is a temporary linear data structure for the visiting check only, we should implement it as a linked-list of pointers of `node_type`, but not a copy of nodes [GS93], [GC94].

Let `N` be the number of nodes of the structure, `L` be the number of links of a node.

The statement `setOfNode = setOfNode Union with {node}` requires O(1) time, if we maintain a head pointer and a tail pointer for the linked list, and each addition of one element is added at the end of the list.

The statement if (`node->link[i] not in setOfNode`) requires O(N) time in the worst case. The function `visit_linked_struct` will be called N times. Hence

```
setOfNode = {}
int visit_linked_struct( node_type *node){
     if (node == NULL)
             return 0;
     visit( node);
     setOfNode = setOfNode Union with  {node}
     for each link i of the node
        if (node->link[i]  not in  setOfNode )
             visit_linked_struct(node->link[i])
         else
              return 0;
     return 0;
}
```

Figure 3.7: A Cyclic Traversal Algorithm

the total time complexity is

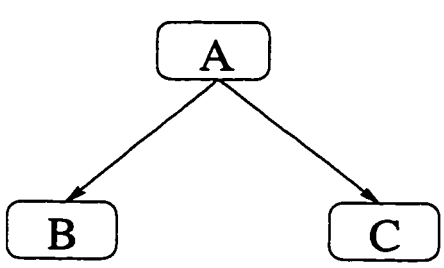
$$O(N * (O(1) + L * O(N)) = O(N^2)$$

For a recursive structure containing no cycles, we do not have to implement the `setOfNode` to check if a next node (`node->link[i]`) of a current visiting node has been visited or not, hence the algorithm has only O(N) time complexity.

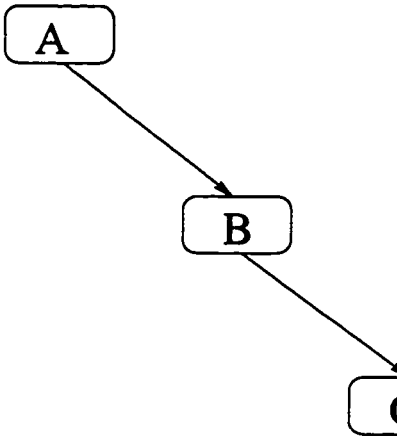
However, if it is not known whether the structure contains cycles, then checking whether the structure contains cycles will also take $O(N^2)$.

A Sequential Representation of a Recursive Structure

The sequential format of a recursive structure is quite tricky. The example shown in Figure 3.8 illustrates the point that the choice for the sequential format for a recursive structure is crucial.



Tree-1



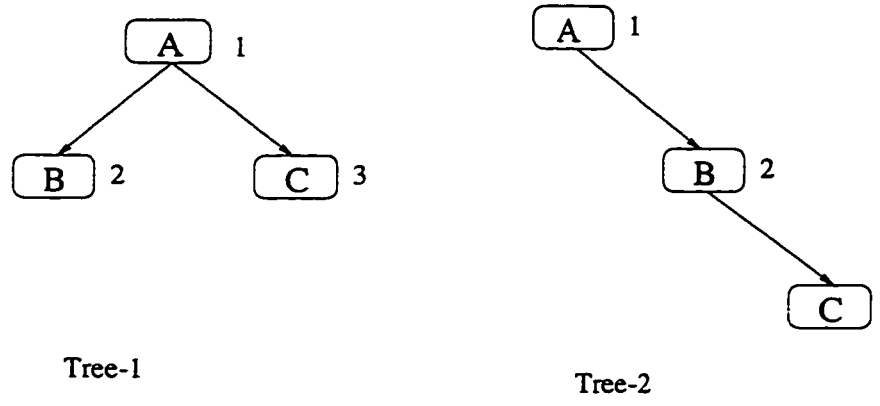
Tree-2

A SEQUENTIAL REPRESENTATION OF Tree-1 and Tree-2 : A B C

Figure 3.8: Different trees can be reproduced from sequential format data.

Tree-1 and Tree-2 do not have the same shape, although they may have the same sequential representation *'A' 'B' 'C'*. An inappropriate choice of this sequential format will lead to the existence of a deterministic encode method but not for the decode method. The equality of two structures is defined not only by the equalities of objects in one structure and those of the other (structural data equality), but also by the equalities of the pointer relations among the objects inside each structure (structural shape equality). We can define the equality of two recursive structures more formally as follows: structure *S1* is said to be equal to structure *S2* if and only if

1. for every object *a1* in *S1*, there exists an object *a2* in *S2* such that a2 is an exact copy of *a1* (except pointers in the objects, because pointers of an object in *S1* point to objects in S1, and so for pointers in an object in S2).



Tree-1

Tree-2

Tree-1 is encoded as : A 2 3 B 0 0 C 0 0
Tree-2 is encoded as : A 0 2 B 0 3 C 0 0

Figure 3.9: Trees of different shapes will be encoded differently.

2. for every object *a1* and object *b1* in *S1*, there exist object *a2* and object *b2* in *S2* such that if *a1* has a pointer to *b1*; *a2* and *b2* are the exact copies of *a1* and *b1* respectively, then *a2* has a pointer to *b2*.

The trick of the choice for an appropriate sequential representation for a recursive structure is to include the pointer-relations of objects in the structure as part of the data of the objects.

The sequential representation for a recursive structure described below is a one-to-one mapping between a recursive structure and its sequential representation. The way it distinguishes the sequential representations for tree-1 and tree-2 is to include the pointer-relations of objects in the sequential representation. Each node is given a unique label, a pointer of an object to another labeled by n will be stored as an integer n. [DD95], [MAM83]

Recalling the tree example, if we label each node with a number as shown in

Figure 3.9, we can encode Tree-1 and Tree-2 as 'A' 2 3 'B' 0 0 'C' 0 0 and 'A' 0 2 'B' 0 3 'C' 0 0 respectively.

Chapter 4

Implementation

The implementation of the persistent object system was written in the programming language C for the UNIX operating system. The heart of the system is the server. The server is supposed to be able to provide services for concurrent application clients. The most important tasks of the server are the capabilities of serving concurrent clients, providing persistent object storage management and security. The system has been designed, implemented and tested with several simple test cases. It is, nevertheless, a prototype implementation, and further work would be required to obtain a production quality version.

4.1 Implementation Specifics

4.1.1 Programming Languages

C language was chosen to implement the system and API, and C++ language was chosen to implement sample application programs. We made these choices, because the languages are available, widely used, and because we know them.

4.1.2 Network Protocol

The system uses the socket interface known as Berkeley Socket Interface available on UNIX to accomplish its communication task. Berkeley Socket Interface is a socket interface that supports the following protocols [Ste90]:

- UNIX domain protocol;
- Internet domain protocol (TCP/IP); and
- Xerox NS domain protocol (XNS).

The choice of domain does not affect the application code, because only the values of some parameters of the socket functions are different, but all socket functional prototypes remain the same. For the simplicity of the project, and the availability of the system and computer account I have had, I have chosen the *UNIX domain* protocol to implement the project, knowing that the conversion from this domain to the others (TCPI/IP or XNS) is simple and can be done quickly, if there is a need.

4.1.3 Communication Synchronization

The communication model chosen for implementing the PO-system is depicted in Figure 4.1. The message sent from a client to the server is called *a request*; and a message sent back from the server to the client who sent the request is called a *response* or *reply*. For synchronization of message passing between clients and server, a client always sends a request to the server first and then waits for a response to return from the server. The server, on the other hand, is waiting for requests. So one request will always generate one response from the server. A request is composed by a message header defined by PO-system. A response is similar. These protocols will be discussed in the next section.

The processing of a request-response pair between an application client and PO-server is slightly different from that between an AdminClient and PO-server. Figure 4.2 describes the processing of a request-response pair between AdminClient and PO-server, while Figure 4.3 is for a request-response pair between an application client and PO-server.

4.1.4 Application Protocol

Application client communication header

The application client communication header is defined as shown in Figure 4.4, where:

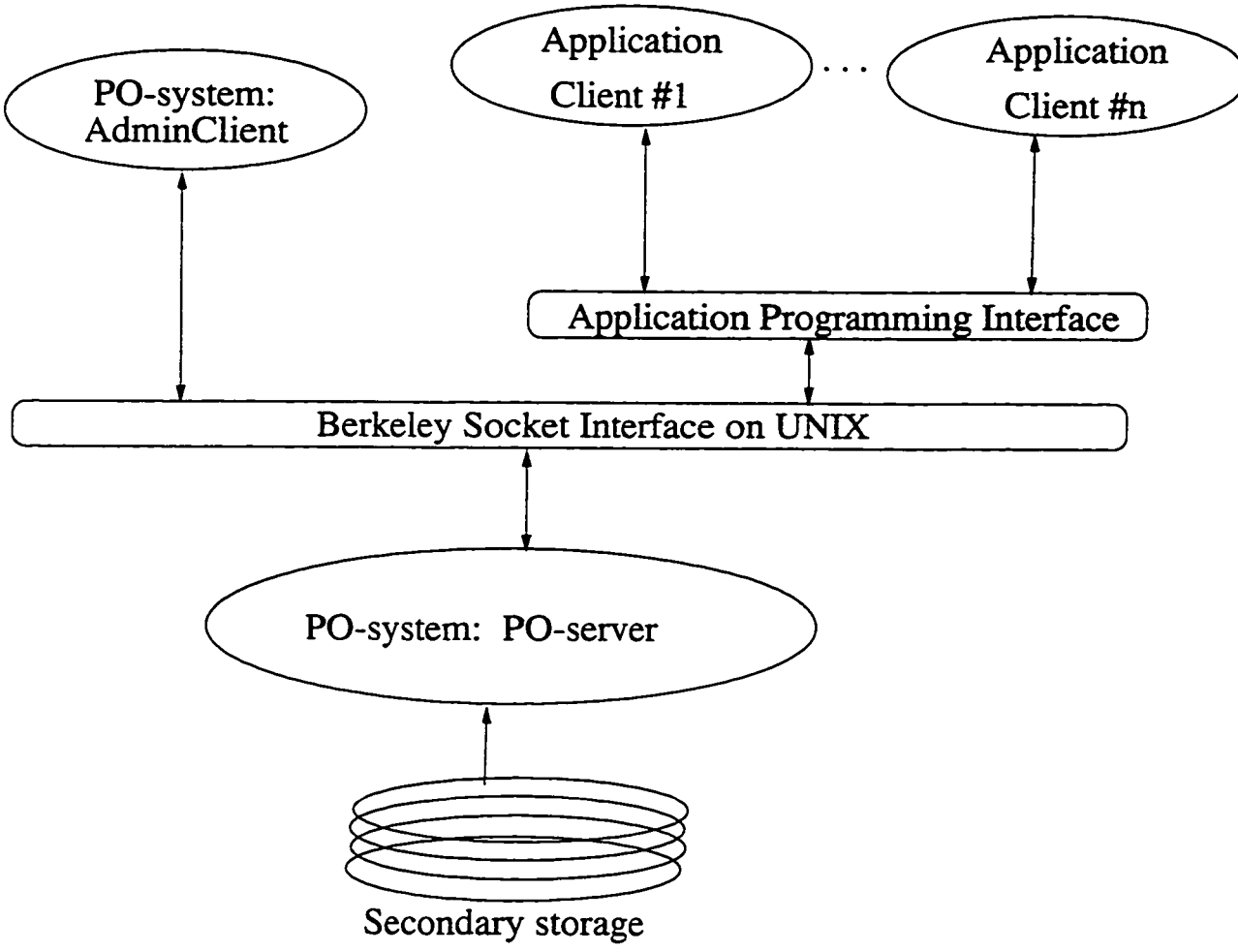
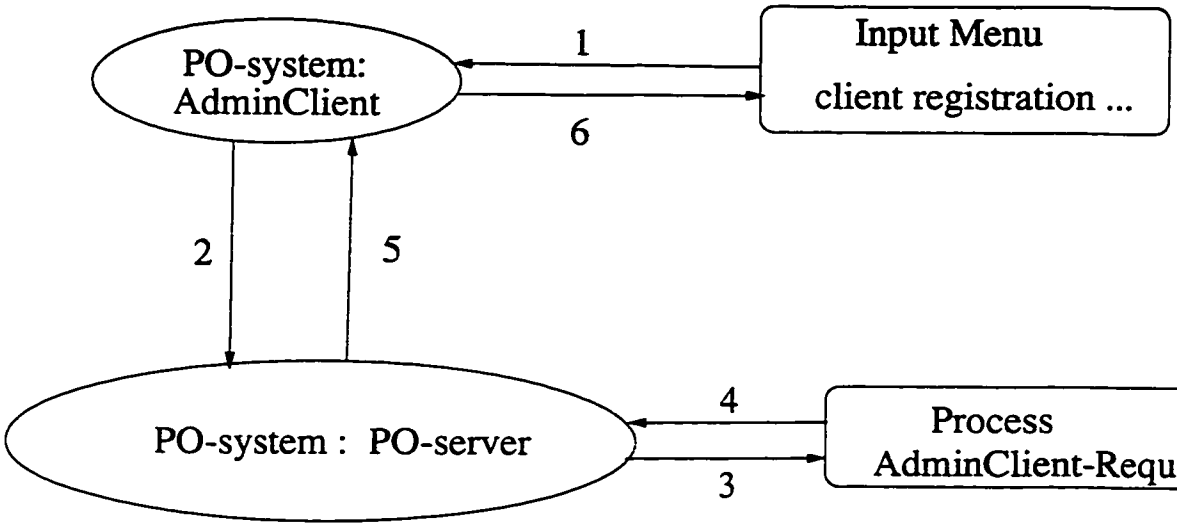


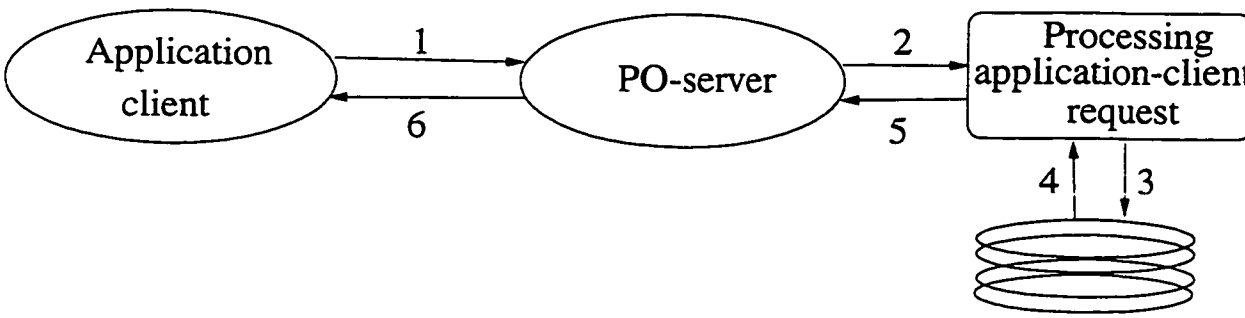
Figure 4.1: The architectural model of the PO-system.



Steps of a complete request:

1. Getting inputs
2. sending request
3. processing request
4. returning result
5. sending response
6. displaying result

Figure 4.2: A request-response for AdminClient and PO-server.



Steps of a complete application-request:

1. sending request
2. processing request
3. storing objects to disk-files
4. loading objects from disk-files
5. returning result
6. sending response

Figure 4.3: A request-response for an application client and PO-server.

```
typedef struct {
int         msgid;                          /* message id */
char        cltname[MAX_CLIENTNAME_LEN];/* APP client name*/
char        clsname[MAX_CLASSNAME_LEN]; /* class name */
char        objname[MAX_OBJNAME_LEN];   /* object name*/
char        owner[MAX_OBJNAME_LEN];     /* object's owner name*/
}ComHeaderStruct ;

typedef   union {
     struct {
          ComHeaderStruct     hd;
          int            dlen;
          char              data[MAX_DATA_BUF];
          } com;
     char    recvbuf[MAX_BUF_LEN];
} cppDataComStruct ;
```

Figure 4.4: Application communication header.

- `msgid` contains a message code [1], in reply, the server will use this field for a reply code (OK, error,..).

  A valid request `msgid` is one of the following:

  - REGISTER_CMD_ID.
  - STORE_CMD_ID.
  - RETRIEVE_CMD_ID.
  - PROCESS_TERMINATE_ID.

- `cltname` is the name of the client who send the request.

- `clsname` is the name of the object's class in the request.

[1] Message codes are defined in common utility header file comutils.h.

- `objname` is the name of the object in the request.
- `owner` is the owner of the object. If the client is also the owner of the object, then this field will be the same as *cltname*.
- `dlen` is the length of the data in field *data*.
- `data` is the data stream of the object.

Administrative Client Communication Header Structure

The administrative client communication header is shown in Figure 4.5, where:

- `msgid` is the message code in the request (only REGISTER_CMD_ID or SHUTDOWN_CMD_ID is a valid `msgid` in the request). In reply, this field contains a reply code (OK, error,..).
- `cltname` is the administrative client name.
- `xprogname` is the application client name.
- `adminPassWord` is the administrative client's password.
- `uFlag` is the running option of the application client (pickling or initialization). Pickling is the option for an application client to unpickle all its persistent objects of last run for the current run; and initialization is to override them with the current run's objects.
- `num_objs` is the total number of persistent objects (PO-objects) in the application's program.

```
typedef struct {
int         msgid;
char        cltname[MAX_CLIENTNAME_LEN];      /* admin user name*/
char        xprogname[MAX_FILENAME_LEN];      /* APP client name*/
char        adminPassWord[MAX_PASSWORD_LEN]; /* admin password*/
uFlag       u;                                /* APP running option*/

int         num_objs;                         /* APP PK objects*/
int         num_cls;                          /* APP PK classes*/
}UiComStruct;
```

Figure 4.5: AdminClient communication header.

- num_cls is the total number of persistent classes (PO-classes) in the application program.

4.2 Persistent Object Server

4.2.1 Data Structures and their Usages

The server uses one global structure, *PKserver*, and three tables, namely *Client-table*, *Class-table* and *Object-table*, to store information, manage data and control request processing.

Server Structure

This global structure PKserver is used to keep track of the current state of the server. The declaration of the structure is shown in Figure 4.6, where:

```
struct PKSERVER {
     int       num_clients;
     int       num_classes;
     int       num_objs;
     int       door;
     union{
          struct {
          ComHeaderStruct     hd;
          int           dlen;
          char          data[MAX_DATA_BUF];
          } com;
          UiComStruct         uicom;
          char          recvbuf[MAX_BUF_LEN];
     }u;
}PKserver;
```

Figure 4.6: The global structure in server program: PKserver.

- `PKserver.num_clients`: is the number of client slots reserved for the registered clients.
- `PKserver.num_classes`: is the number of class slots reserved for the registered clients.
- `Pkserver.num_objects`: is the number of object slots reserved for the registered clients.
- `PKserver.door`: is the communication state of the server (open, shutdown).
- `PKserver.u.recvbuf`: is the server's buffer (used to receive request and send reply to a client).

Client-Table

The client-table is used to keep information about the clients' registration. The declaration of the client table is shown in Figure 4.7, where:

- `cltname` is the client name.
- `send_addr` is the address of the client.
- `obj_index` is an array holding indices of objects in the object-table that is currently using by the client.
- `u` is the running flags of the client. This flag is included in a client-registration request sent from AdminClient. A user register for the client to AdminClient can make a choice for the client's running flag. The choice can be either *initialization* or *unpickling*.

  1. *Initialization* running flag informs the PO-server to override the objects of this client by the objects this client uses to do object-registration.
  2. *Unpickling* running flag informs the PO-server to load the objects this client stored at previous execution and send each one of them back to the client for the first object-retrieval request for each object.

- `num_objs` is the total persistent object in the client program.
- `num_cls` is the total persistent classes in the client program.
- `timestamp` is the time of the client registration.

```
struct CLT_TAB_SLOT {
     char       cltname[MAX_CLIENTNAME_LEN];
     char       send_addr[MAX_SOCK_ADDR_LEN];
     SFlag      u;
     int        obj_index[MAX_OBJS];
     int        num_objs;
     int        num_cls;
     time_t     timestamp;
} clt_tab[MAX_CLIENTS];
```

Figure 4.7: Client-table in server program.

Class-Table

The class-table is used to store information of classes of objects currently allocated in object-table. Each entry in the class table, Figure 4.8, has the following fields:

- `clsname` is the class name.
- `num_objs` is the number of objects in the object-table belong to this class.

Object-Table

The object-table is used to load objects from object-files or to store objects registered or stored from application clients, and other information of the objects. The object-table's declaration is shown in Figure 4.8, where:

- `objname` is the object name.
- `owner` is the owner of the object.
- `cls_index` is the index of the object's class in class table.

```
struct CLS_TAB_SLOT{
     char clsname[MAX_CLASSNAME_LEN];
     int      num_objs; /*overwritten allowed only if 0 obj */
} cls_tab[MAX_CLASSES];



struct OBJ_TAB_SLOT{
     char      objname[MAX_OBJNAME_LEN];
     char      owner[MAX_OBJNAME_LEN];
     int       cls_index; /* index to class table */
     int       num_clts;
     int       dlen;
     char      *data;
} obj_tab[MAX_OBJS];
```

Figure 4.8: Class-table and object-table in server program.

- `num_clts` is the number of clients in the client table that access this object.
- `dlen` is the length in bytes of field *data*.
- `data` is a pointer to the data stream of the object. Its size is equal to `dlen` and its space is dynamically allocated at run time.

File header structure

When receiving a process-termination message from a client, all objects owned by the client will be encrypted and then written to the file system, under a file name defined as *OBJECT_DIR/object_name.class_name.client_name*.

Each file will contain non-encrypted information header as defined in Figure 4.9, where:

```
typedef struct {
 char clsname[MAX_CLASSNAME_LEN];
 char objname[MAX_OBJNAME_LEN];
 int  length;
} file_rec_header;
```

Figure 4.9: File-header structure declaration.

- `clsname` is the class name.
- `objname` is the object name.
- `length` is the length of the data stream.

4.2.2 Security Implementation

The server does two things for security implementation: user-password checking for AdminClient's requests, encryption for object storage in files, and the restriction of *read-only* (retrieving) for objects that are not owned by the client.

User-Password Checking for AdminClient

For simplicity, the client name of the AdminClient and its password are hard-coded. This information will be checked for AdminClient requests as shown in Figure 4.10.

Encryption/Decryption

The encryption and decryption is applied on each byte of a data stream, using the following equation:

```
if client name is "ADCLT" then
   if password is not "pk" then
      security_error
   else
     security_ok
```

Figure 4.10: User-Password checking for AdminClient request.

$encryptionData = 255 - data$

Hence, the decryption process will call the same routine as the encryption's

$data = 255 - encryptionData$

It would be straight forward to replace this trivial encryption scheme with a more secure system.

Loading/Unloading

Each object eventually will be stored in a file in */OBJECT_DIR/* directory. An object's file name must uniquely identify the object. The identity of an object consists of object's owner name, object's class name, and the object name. It is important to note that the owner name is the name of the client that is the owner of the object, and the client name is chosen by the user for an application program that requires persistent object services from PO-server. A user can run concurrently more than one application program that requires persistent object services from PO-server, as long as the user selects different client names for them. Hence, choosing the object file-name as a composition of its owner name, class name and object name will surely be unique. An object *obj* of class *cls* of client (owner) *clt* will be stored in a file named

```
Loading an object:
     open the object file
        if not exist
           return error_object_not_exists
        else
           read the file header to get the data length
           allocate memory space
           read data block  with data length
           decrypt the data block
           return OK

Unload an object:
     open object file (if the file not exist, create one)
        construct file header and write to the file
        encrypt data
        write data to the file
```

Figure 4.11: Algorithm for loading and unloading an object.

: */OBJECT_DIR/obj.cls.clt.* Figure 4.11 illustrates the algorithms for loading (read) an object from a file and unloading (write) an object to a file.

Object's Authorization

Clients who wish to access an object owned by another client, are restricted to *read-only* for the object. In other words, they are subjected to the following restrictions:

- They are not allowed to do a storing request for the object.
- They are not allowed to make a registration request for the object.
- They can register themselves through the Adminclient only with option *unpickling.*

This implementation scheme is also used for concurrency control of the server.

4.2.3 Concurrency Control

The concurrency control of the server is done by serialization of the requests, object authorization, and concurrent client restriction. In the current version of PO-system, the server is a single process that queues requests and serves them in FIFO sequence. Furthermore, a stored object can be accessed by at most one writer at a time. This is sufficient to ensure consistency of the PO-system. In a more complex implementation of PO-system, with multiple server processes, it would be necessary to serialize write requests.

Serialization

All clients can run concurrently, but their requests are serialized on the server site. Only one request is served at a time by the server. The queueing is done at the socket level, so no extra effort is need for the server.

Restriction on Concurrent Clients

The reason for concurrency control for the server is data consistency. The object's authorization alone does not guarantee the multiple-write to an object in case the same program is executed twice, one immediately after the other. Furthermore, if an unauthorized client of an object is executing concurrently with the owner client of the object, then the owner client may modify the object at some point of its execution,

and may create an inconsistent view of the object for the other client if the other client does retrieving requests for the object before and after the modification of the object. The server restricts its current clients not to have the same name. One program uses only one client name, and the server does not allowed two clients of the same name to run concurrently. Let say, client `clt1` is running, for some reason, we want to run the `clt1` again, when we register for `clt1`, we get a rejection message from the server, saying that `clt1` is in running state. For the scenario of the inconsistent view of an object for two consecutive retrievals from an unauthorized client, we assume that the client should be aware that between two retrievals of an object, the object may be modified by its owner.

4.2.4 Resource Availability Control

There is a bottleneck effect for the design. Since the server is the sole process that serves all its clients, the performance can deteriorate if there are too many concurrent requests waiting for service. In order to prevent this possibility, the server limits itself to serve up to a certain number concurrent clients (MAX_CLIENTS), and to keep up to a certain number of classes (MAX_CLASSES) and objects (MAX_OBJECTS) in memory. At client registration, the total number of classes and the total number of objects are checked to guarantee these limits. Live-lock may occur because of these limitations. For example, if a client registers with max_objects set to MAX_OBJECTS, and never starts running, then none of the other clients can register. In order to prevent this, the server applies a time-out check if a client-registration failed because

```
count_cleanup_clients = 0;
time(&now);
for (i =0; i <MAX_CLIENTS; i++){
     if (clt_tab[i].executing == FALSE)
       diff_time = (double) now - (double)clt_tab[i].timestamp;
       if (diff_time  > MAX_WAITING_TIME){
        cleanup_inactive_client[i];
        count_cleanup_clients = count_cleanup_clients + 1;
       }
}
  return  count_cleanup_clients;
```

Figure 4.12: Cleanup Client algorithm.

one of the limits is reached.

When a client register request fails because of the shortage of resources (CLIENT_FULL, CLASS_FULL, OBJECT_FULL), the server will check all currently registered clients to do a time-out-cleanup as shown in Figure 4.12. If a client who is not in executing state, and its registration timestamp has passed MAX_WAITING_TIME [2], then its registration will be canceled (clean-uped). If `client_timeout_cleanup()` results in cleaning up one or more clients, then the server will resume the client registration request.

4.3 Administrative Client (AdminClient)

The implementation of the client is just a sample program used to test its counterparts, the server and application clients.

[2]Defined in header file pkser.h.

4.4 Application Programming Interface (API)

Let us recall the API prototypes defined from Chapter 3. Three functions: `PK_register_object()`, `PK_retrieve_obj()`, and `PK_store_obj()` have the same function prototypes (parameters). `PK_terminate()` can be considered to have the same prototype, except `object_owner`, `class_name`, `object_name`, `buffer` all set to NULL and `*send_receive_length` is set to zero. All API functions execute the following steps:

1. allocate send_buf.
2. construct application communication header.
3. move data from buffer to header->data.
4. set header->data_len = *send_receive_length.
5. send content in send_buf to the server.
6. wait for response from the server.
7. strip header.
8. set *send_receive_length = header->data_len.
9. move header->data to buffer for header->data_len.
10. free send_buf.
11. return header->msgid.

4.5 Sample Applications

There are four sample application programs each for one criterion of the data structure of the objects. Appendix A includes a document, named *QuickStart and Test Scenarios*, as the demonstration document of the PO-system.

4.5.1 Simple Class Application Program

Program `prog1.cpp` declares a class as follows:

```
class PK_road_vehicle {
     int wheels;
     int passengers;
     char *name;
public:
     void set_wheels(int num) { wheels = num ;}
     int get_wheels(void)     {return wheels; }
     void set_pass(int num)   {passengers = num;}
     int get_pass(void)       {return passengers;}
     void set_name(char *name2) {
             name = new char [strlen(name2) +1];
             strcpy(name, name2); }
     char  *get_name(void) { return name; }
     void encode_send(char *buf);
```

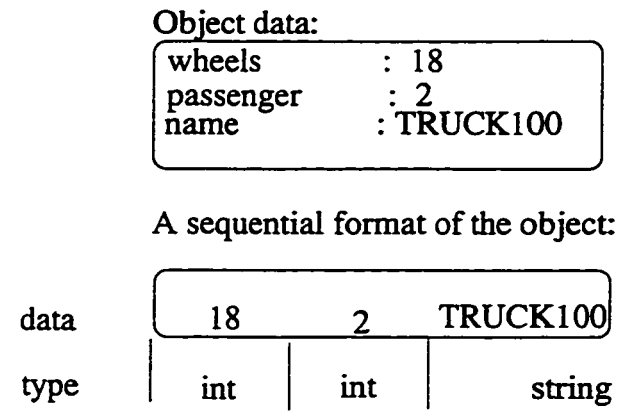


Figure 4.13: A sequential format for an object of the simple class

```
    void decode_recv(char *buf);

    void show(void);

};
```

Since an object of this class has no pointer, and there is no parent class, a sequential format chosen for this object shown in 4.13 is sufficient.

4.5.2 Inherited Class Application Program

Program `prog2.cpp` declares an inherited class as follows:

```
class PK_road_vehicle {

    int wheels;

    int passengers;

public:

    void set_wheels(int num) { wheels = num ;}

    int get_wheels(void)     {return wheels; }
```

```
        void set_pass(int num)    {passengers = num;}
        int get_pass(void)        {return passengers;}
        void encode1(char *buf, int *len);
        void decode1(char *buf, int *len);
};

enum type {car, van, bus};
class PK_automobile : public PK_road_vehicle {
        enum type auto_type;
        char *auto_name;
public:
        void set_type(enum type  t) { auto_type = t;}
        void set_name(char *name){auto_name = new char [strlen(name) +1];
                                  strcpy(auto_name, name);}
        char *get_name(void) {return auto_name; }
        enum type get_type(void) {return auto_type ; }
        void show(void);
        void encode(char *buf, int *len);
        void decode(char *buf,int *len);
};
```

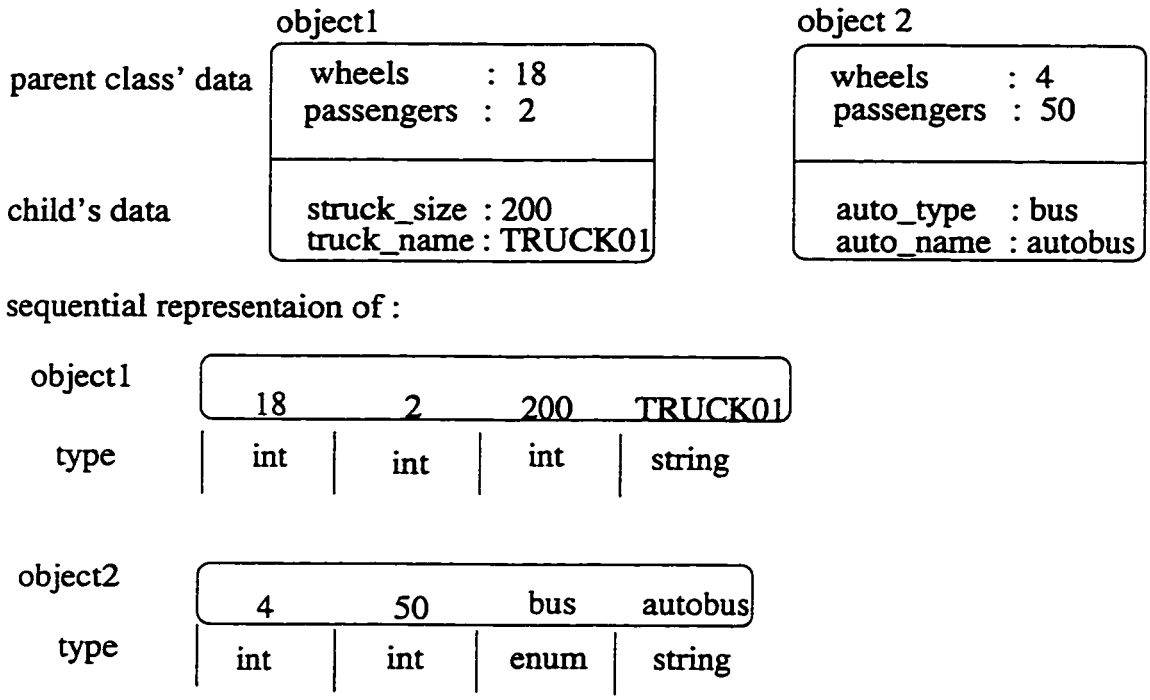


Figure 4.14: A sequential format for an object of the inherited class.

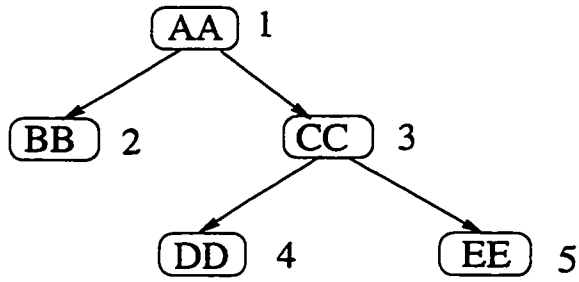
An object belonging to this class will have to consider its parent's data (the data declared in the parent class) and its own data (data declared in the child class) as a whole, *the object's data.* The parent class needs its own `encode` and `decode` methods, and so does the child class. A choice for this specific example in shown in Figure 4.14.

4.5.3 Acyclic Recursive Structure Program

A tree structure is created by program `tree.cpp`. A class is declared as follows:

```
class PK_tree{

      char *name;

      PK_tree   *l;
```

```
        PK_tree   *r;


public:
        void set_name(char *name2) {
                        name = new char [strlen(name2) +1];
                        strcpy(name, name2); }
        char  *get_name(void) { return name; }
        PK_tree(char *s);
        PK_tree *get_left_ptr(void) { return l;}
        PK_tree *get_right_ptr(void) { return r;}
        void  set_left(PK_tree *left) { l = left;}
        void  set_right(PK_tree *right) { r = right;}
        void  print_tree(){
                        cout << name << "\n";
                        if (l != NULL)
                            l->print_tree();
                        if (r != NULL)
                            r->print_tree();


        };
};
```



A sequential representation of the tree shown above is :

AA 1 2 BB 0 0 CC 4 5 DD 0 0 EE 0 0

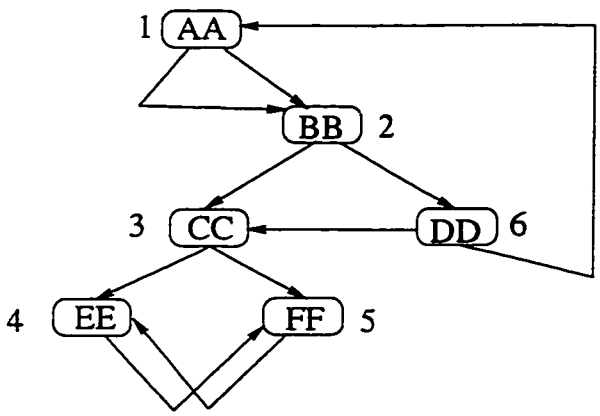
Figure 4.15: A sequential format for a tree structure

This is a kind of recursive structure. A choice of sequential format of a recursive is discussed in Chapter 3. Since a tree does not contain any cycles, a tree traversal algorithm, mentioned in many text-books, is sufficient for coding the `encode` and `decode` methods.

4.5.4 Cyclic-Structured Application Program

Program `cyclic.cpp` reuses the declaration of a tree, but the cyclic structure is formed as shown in Figure 4.16, by allowing pointers of some successors to their ancestors. The `print` method in `PK_tree` class is not suitable to use for cyclic structure, because it will make an infinite-loop at run time. A new `print_cyclic` method will be needed instead.

The cyclic-traversal algorithm discussed in Chapter 3 is used to code the `encode` and `decode` methods, and also for the `print_cyclic` method.



A sequential representation of the cyclic structure shown above is :

AA 2 2 BB 3 6 CC 4 5 EE 0 5 FF 4 0 DD 3 1

Figure 4.16: A sequential format for a cyclic structure.

Chapter 5

Assessment

The PO-system has been implemented and tested. The final demonstration of the project showed the success of the project. However, there are still limitations. In this chapter, I would like to discusss the success and limitations of the PO-system through our test scenarios. The test scenarios include the tests of the sample applications, the security, the application programming interface (API), the time-out clean-up processing, the concurrency, and the consistency of the system.

5.1 Sample Applications

Simple tests were conducted with a single application at a time. The system has been tested successfully with four typical sample application programs. Each of the sample application programs represents one typical application data structure. The data structures are simple, inherited, linked and cyclic structures.

5.2 Security

The security of the system is enforced by a password check for the administrative client, the encryption/decryption of object forwards and backwards between server and the storage media (disk), and object authorization.

The password check and encryption/decryption works well. Object authorization has been tested with several application clients to ensure that a client can access objects of its own and is restricted to read-only for objects of the other clients. Testing results of object authorization are also sufficiently good with one exception. The exception will be discussed below in Section 5.6.

5.3 API

A simple API written in C programming language is provided. Only application programs written in a programming language that can bind with C modules, such as C++, can bind to these functions. For application programs written in other programming language that can not bind with cross languages option (i.e., not able to bind with C modules), then it will be necessary to build similar API function sets in their own programming languages.

5.4 Time-out Clean-up

A scenario where starvation may occur is when application clients do client-registrations to the PO-server and never start running (inactive). Because after a successful registrations, the server will reserve space (object slots in object-table and class slots in class-table) for the registered application client to start at any time, if the client does not run then the reserved slots will never be used by others, and eventually, the server will be short of space for other client registration. So *time-out clean-up* procedure is a procedure used to get rid of inactive client registration and use the reserved space for a new client. This scenario has been simulated and tested successfully.

5.5 Concurrency

The PO-system is designed for concurrent applications. The system has been tested with several application clients running concurrently. The results of the tests from the traces provided by all application clients were the same as each individual running in series. However, the performance was not quite as good as expected.

In general, the more clients running concurrently, the slower the PO-system will be. This is known as a bottleneck effect of the design, and it occurs when one server process serves many client processes.

5.6 Consistency

The consistency of concurrent accesses of a persistent object is ensured by serialization of requests and the PO-system's user restrictions as set out in Chapter 3. The consistency of persistent objects in the PO-system has been verified through a set of the programs.

The outcomes of the tests conformed with the consistency of the PO-system as discussed in Chapter 3. However, the first PO-system restriction, *one process is allowed to run a program at a time*, is not enforced at run-time. The violation of this by mistake will not be detected but may cause inconsistent scenarios.

Chapter 6

Conclusion And Further Work

The project gave me valuable, practical and educational experience. Nevertheless, the final product is not perfect, and needs to be improved.

6.1 Experience

Persistence is an interesting research topic. The implementation of the project was a valuable experience of research and practice for me.

I started the project by asking myself the following questions:

1. What is persistent data?
2. What are the existing solutions for it?
3. What can be a new solution?
4. What would be the advantages of this new approach?

5. What could be done to make this new solution more attractive?

It was the answers to these questions that made up the primary goals for the project: *simple, user-friendly, secure* and *inexpensive* solutions for particular applications.

The specific design of our PO-system relates to other issues, including concurrency, consistency, efficiency, object-oriented programming, data management, data structures, distributed domain application, security. So the project also gave me practical experience. The implementation of the project is actually programming practice. The software development process starting from requirement analysis to final testing has been practised. Feedback from testing was used to alter the design, the modification of the design required the modification of the code and then further testing, and so on until we obtained satisfactory results from the tests.

6.2 Advantages and Disadvantages

The PO-system shows great advantages for the enhancement of existing object-oriented applications.

We chose synchronous communication to implement the PO-system, because of its simplicity. A request sent from a client will always be responded by the server. This synchronization scheme causes some overhead, especilly when data encryption or decryption is required. The time taken for a request-response pair is proportional to the request-processing time on the server site. However, the system still has many

advantages.

In general, the usage of the PO-system has the following advantages and disadvatages for its end-users:

Advantages:

- Programming language independence: the application clients can be written in any programming language.
- User-friendliness: usage of the PO-system is quite simple compared with other existing systems.
- Security: the system was implemented with object-authorization, user-password check, encryption/decryption mechanisms.
- Inexpensive system: the construct of the system is quite simple.

Disadvantages:

- Encryption and decryption overhead: the PO-server has to process the encryption and decryption when loading and unloading objects.
- Concurrency restriction: only one process is allowed to run an application program at a time.
- Additional task for application users: two additional (`encode/decode`) methods are required for each persistent class, and the coding of theses methods manually is error prone.

6.3 Further Work

The project should be considered as a first version of a new solution toward the persistent object issue, but not a commercial product. Improvement of the product is needed and I would suggest the followings:

- Performance: This is the principal area in which improvement is required. Since the server is the sole process that serves all clients, the server itself is the *bottle-neck* of the system. This bottle-neck can be eliminated by creating one server process to serve one client process. This modification will surely improve the performance of the PO-system. The server program may not get bigger, but the scenarios are more complex, because the serialization of requests on the server site is not true anymore. This change is a major change, and may require the revision of the entire server design to make sure that the new server can work as properly as the previous version does.

- Client termination detection: It is necessary for the server to know when its client terminates, for some reason, without sending a process-terminate request to the server. One way for the server to detect this is to check periodically the communication with its clients. Within a predefined period of time, if a client does not send a request to the server, then it is considered to be terminated on the server site.

- Security: Security applcation in the PO-system is too simple (*user-password* pair verification for *adminClient* and *encryption* for *storing data*). Further-

more, the name of the AdminClient and its password were hard-coded, and the encryption algorithm used only a simple formula ($255 - data$). A future version can use more sophisticated mechanisms to do the job.

- Application Programming Interface: although we keep saying that the API is not part of the PO-system, we provide an API written in C as an example for the users. C++ users can make use of this API without difficulty. For programming languages that are compatible with C, users must build their own API. It would be nice to build similar API for other languages.

- Aids for *encode* and *decode* methods: Fully automatic *encode/decode* methods provision for the PO-system users is not our intention, because it is too complicated. It is a solution that requires the combination with a compiler like *A Pickling-system or E language for C++*, and hence *language dependent.* However, in order to ease the task of programmers in coding these methods for persistent classes, we could provide some generic functions in a library that a user can use in their encode and decode methods.

Bibliography

[Alm94] Almasi/Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company Inc., RedWood, CA., 1994.

[Cra93] Daniel H. Craft. A study of pickling. *JOOP*, 5(8):54–66, January 1993.

[DD95] Danilo Dabbene and Silverrio Damiani. Adding persistence to objects using smart pointers. *JOOP*, 8(3):33–39, June 1995.

[GC94] Peter Grogono and Patrice Chalin. *Copying, Sharing, And Alliasing.* Department of Computer Science, Concordia University, 1994.

[Gro91] Peter Grogono. *Issues in Design of an Object-Oriented Programming Language.* Department of Computer Science, Concordia University, 1991.

[GS93] Peter Grogono and Philip Santas. *Equality in Object-Oriented Languages.* Department of Computer Science, Concordia University, 1993.

[Hug91] John G. Hughes. *Object-Oriented Databases.* Prentice Hall International (UK) Ltd, 1991.

[MAM83] Paul Cockshott Malcolm Atkinson, Ken Chisholm and Richard Marchall. Algorithms for a persistent heap. In *Software-Practice and Experience*, volume 13, pages 259–271. John Wiley & Sons Ltd., 1983.

[Mil91] Mark A. Miller. *Internetworking.* MT Publishing Inc., 1991.

[Pit97] Robert I. Pitts. Learning to program the object-oriented way. *SunExpert*, 8(1):56–71, January 1997.

[Sch90] Herbert Schildt. *USING TURBO C++.* Osborne McGraw-Hill, Berkeley, California, 1990.

[Ses96] Roger Sessions. *Object Persistence Beyond Object-Oriented DataBase.* Prentice Hall, New Jersey, 1996.

[Ste90] W. Richard Stevens. *Unix Network Programming.* Prentice Hall, New Jersey, 1990.

[VC90] Greg Voss and Paul Chui. *Tutor C++, Disk-Tutor.* Osbone McGraw-Hill, Berkeley, 1990.

Appendix A

Quick Start and Test Scenarios

A.1 Terminologies

- PO-system: (Persistent Object system) is a persistent data management system. It is implemented as a set of intercommunicating processes (programs), in which one is the PO-server, one is the administrative client (AdminClient), and the others are application (APPC) clients.

- PO-server :(PO-system server) is the server that provides services to its APPC clients and its AdminClient. The services for APPC clients include registration, retrieving and storing persistent objects. The services for AdminClient include APPC client-registration, and PO-system "shutdown" request.

- AdminClient: (Administrative client) is the client that communicates with the PO-server to register (client-registration) for an APPC client, or to request

a "shutdown"of the PO-system. AdminClient also provides means for APPC clients to input client-registration information.

- APPC client: (application client) is an object-oriented application (program) client, running independently to PO-server, that requires persistent object services from the PO-server at run time.

- client name: is a string of characters used by a client to identify itself to the PO-server. The name must be unique with respect to other clients.

- PO-class: is a class in an object-oriented program, whose object(s) will have to be used to register, retrieve or/and store to the PO-server during the execution of the program.

- PO-object: is an object of a PO-class.

- Client-registration: is a request sent from the AdminClient to PO-server to register (reserve space) for an APPC client.

- Object-registration: a request for storing a PO-object sent from an APPC client to PO-server with the object's initialized value. This has to be done at the beginning of the program right after the object has been initialized.

- Object-storing: a request for storing a PO-object sent from an APPC client to PO-server.

- Object-retrieving: a request for retrieving a PO-object sent from an APPC client to PO-server.

A.2 QuickStart

This section shows the usage and testing of PO-system step by step through sample application programs.

1. Step 1: open a window to run pkserver:

   cd /mnt/logic1/grad/banguyen/proj/

   `prompt> pkser`

2. Step 2: open a window to run AdminClient and then register for some application clients

   cd /mnt/logic1/grad/banguyen/proj/

   (a) `prompt> admin1t`

   you will be led to a menu with a list of choices.

   (b) select option: enter "1" in the menu shown in Figure A.1, because this is the first time the client communicates to the PO-server.

   (c) enter clientname

   client name :prog1

   (d) enter the maximum number of PO-objects for the client (in cpp1, there is only 1 PO-object)

   max PO-objects :1

```
-----------------------------------------------------------------

  select a running option for the APPC client or shutdown server request

  0: unpickling objects

  1: initialization (first run of the APPC client)

  2: ignoring pickling objects of the APPC client's last run

  3: shutdown PO-server

  option : (0/1/2/3) :_

-----------------------------------------------------------------
```

Figure A.1: AdminClient Menu.

(e) enter the maximum number of PO-classes for the client (in cpp1, there is only 1 PO-class)

maximum PO-classes :1

(f) enter AdminClient password ("pk" is hardcoded as the password of the AdminClient)

Admin client Password :pk

3. Step 3:

open a new window to run the application client.

cd /mnt/logic1/grad/banguyen/proj/

```
prompt> prog1
```

After the execution of prog1. we can do the following:

1. verify results through traces appearing in windows:

   window 1 (trace for PO-server), window 2 (Admin Client), and

   window 3 (application client prog1).

2. repeat step 2.1 to 3 again with different options (except option shutdown) to see what will be in the traces.

3. repeat step 2.1 with option shutdown Pkserver through step 2.6 (step 2.2, 2.3, 2.4 will be skipped.) to see how PO-server reacts.

4. repeat step 1 through 3 with other application clients:

   - prog2:

     client name : prog2

     max PO-objects:2

     max PO-classes:2

   - tree:

     client name : tree

     max PO-objects: 1

     max PO-classes: 1

   - cyclic:

     client name : cyclic

     max PO-objects: 1

     max PO-classes: 1

A.3 Test Scenarios:

A.3.1 Simple Tests

Simple tests are done by running application client programs that do not access objects of other clients, one at a time. If we go through the guidelines in the previous section, we have tested the following:

1. simple class object.
2. inherited class object.
3. linked structure object.
4. cyclic structure object.

A.3.2 Concurrency

Simple concurrency test is done by running two (or more) independent applications clients simultaneously, providing that none of these clients require the access of object(s) of others.

That is to start the PO-system, and register for pro1, prog2, then run prog1, prog2 concurrently in two windows and verify expected results with the traces.

A.3.3 Security

Object Authorization

Object authorization can be tested by running an application client that requires the access of object(s) of other client(s) that are not currently running.

That is to register prog11 and run prog11, then look at the trace to verify the results (prog11 is the same program as prog1, except that it accesses an object of prog1. Similarly, prog21 is the application program that accesses 2 objects of prog2.)

Object Authorization and Concurrency

The combination of object authorization and concurrency test scenario can be created by running two application clients concurrently, one requiring objects of the other.

That is to register prog1 and prog11, then run prog1 and prog11 concurrently, and verify the results from the traces. Then test the same for prog2 and prog21.

A.3.4 Shutdown-Request

Select "shutdown" request from AdminClient's menu to see if PO-system can shutdown smoothly.

A.3.5 Time-Out Clean-Up

Time-out clean-up function will be called only when the server is short of resource. We can create this condition as follows:

- Register for the first application client with high max PO-objects or PO-classes (to simulate the out of resource condition for the next register of another client). Wait for one minute for the time-out condition becoming true (MAX_WAITING_TIME is set to one minute).

- Register for a second client with also high max PO-objects or PO-classes to simulate a condition of out of resource.

```
ei. 1st max PO-objects + 2nd max PO-objects > MAX_OBJECTS ( = 50)
or
ei. 1st max PO-classes + 2nd max PO-classes > MAX_CLASSES (= 20)
```

- Run the first client and observe the results. The client should get a return code as it has not been registered, because its registration has been cancelled by "time-out" condition.

A.4 Application Client

In this demonstration, all application programs (clients) are provided (prog1 prog12, prog2, prog21, tree, cyclic).

If a user wishes to build up his/her application or modify an existing OO application program to make use of this PO-System, then read this section and verify with the sample programs.

A typical Pk client program should look like this :

```
// class declaration
PO_class1 ...
main(){
PO_class1 Obj1, obj2;
int ret_code = OK;
...
// register the obj1
obj1.encode(buffer_ptr );
ret_code =  PK_register_obj(...);
// do first retrieve obj
ret_code = PK_retrieve_obj(...);
obj1.decode(buffer_ptr);

//...

// do last store obj
  ret_code = PK_store_obj(...);
```

```
    // inform PO-server that the execution of the program
    // terminates here.
      ret_code = PK_terminate(...);
    }
```

Your program should include `cpplib.h` and bind with `cpplib.o` in order to call the following functions:

```
int PK_register_obj (char *client_name,
                     char *owner,
                     char *class_name,
                     char *obj_name,
                     char *buffer_ptr,
                     int  *buf_len);
int PK_retrieve_obj (char *client_name,
                     char *owner,
                     char *class_name,
                     char *obj_name,
                     char *buffer_ptr,
                     int  *buf_len);
int PK_store_obj    (char *client_name,
                     char *owner,
                     char *class_name,
```

```
                        char *obj_name,
                        char *buffer_ptr,
                        int  *buf_len);
int PK_terminate (char *client_name);
```

- client_name is the name that you use for AdminClient to register your program to the PO-server. client_name can be any string but it should be different from other clients.
- owner is the name of the owner of the object named obj_name of the class named class_name. If the client is also the owner of the object, then the client_name and owner in the parameters should be the same.
- class_name (string) is the class name as it appears in the program.
- obj_name (string) is the object's name as it appears in the program
- buffer_ptr is a pointer pointing to the buffer that contains the encoded data of an object and buf_len is the length of the data in the buffer.

It is the application programmer's task to encode an object into a stream of data in a buffer before doing an object-registration or object-storing; and to decode the received stream of data into an object after doing an object retrieved. There are many ways to implement the encode/decode methods. The ways chosen to implement the encode/decode methods in the sample application programs are not unique. I do

not attempt to prove they are the best way for implementing encode and decode methods. An application programmer should keep in mind that **encode** and **decode** must be one-to-one functions.

```
Object A <==decode/encode==> stream of data of object A
```

Appendix B

Common Header File

```
/*=============================================================*/
/*       Project      :   Pickling System                      */
/*       Filename     :   comuitls.h                           */
/*       Discription  :   Common header file                   */
/*                                                             */
/*                        Ba-Nguyen, Tran                      */
/*=============================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>

#include <sys/file.h>

#include <unistd.h>

#include <time.h>

#include <errno.h>


#define  OK 0

#define  FALSE 0

#define  TRUE 1

#define  TMPDIR                "/tmp/"

#define  SERVER_LISTEN_ADDR    "/tmp/PKList"


#define  oops(msg)       { fprintf(PK_output, msg); exit(-1); }



#define MAX_CLIENTNAME_LEN      50

#define MAX_CLASSNAME_LEN       50

#define MAX_OBJNAME_LEN         50

#define MAX_SOCK_ADDR_LEN       50

#define MAX_FILENAME_LEN        50

#define MAX_PASSWORD_LEN        50
```

```
#define MAX_BUF_LEN             2000

/*buffer length + msgid + client+class+obj names lenghts*/

#define MAX_DATA_BUF            2000 + (4+50+50+50)


#define FIRST_MSG_ID           -1

#define HANDSHAKE_MSG_ID        0

#define REGISTER_CMD_ID         1

#define STORE_CMD_ID            2

#define RETREIVE_CMD_ID         3

#define REPLY_MSG_ID            4

#define LAST_MSG_ID             5

#define SHUTDOWN_CMD_ID         6

#define PROCESS_TERMINATE_ID    7



#define MSG_ERR                   100

#define MSG_ID_OUT_OF_RANGE_ERR   MSG_ERR +1

#define MSG_RECV_ID_ERR           MSG_ERR + 2

#define MSG_RECV_LEN_ERR          MSG_ERR + 3

#define MSG_CLIENTS_FULL          MSG_ERR + 4
```

```
#define MSG_CLIENT_EXIST          MSG_ERR + 5

#define MSG_OBJS_FULL             MSG_ERR + 6

#define MSG_CLASSES_FULL          MSG_ERR + 7

#define MSG_CLT_NOT_YET_REG       MSG_ERR + 8

#define MSG_ADMIN_PASSWORD_ERR    MSG_ERR + 9


#define SOCK_OPENED               1

#define SOCK_CLOSED               0


#define SOCK_ERR                  200

#define SOCK_OPEN_ERR             SOCK_ERR + 1

#define SOCK_NOT_OPEN_ERR         SOCK_ERR + 2

#define SOCK_BIND_ERR             SOCK_ERR + 3

#define SOCK_LISTEN_ERR           SOCK_ERR + 4

#define SOCK_CONNECT_ERR          SOCK_ERR + 5

#define PROTOCOL_FORMAT_ERR       SOCK_ERR + 6

#define SOCK_ACCEPT_ERR           SOCK_ERR + 7

#define SOCK_WRITE_ERR            SOCK_ERR + 8


#define GENERAL_ERR               300

#define UNKNOWN_TYPE_ERR          GENERAL_ERR + 1

#define NEG_LENGTH_ERR            GENERAL_ERR + 2
```

```
#define CLT_NOT_REG_ERR          GENERAL_ERR + 3

#define CLS_NOT_REG_ERR          GENERAL_ERR + 4

#define OBJ_NOT_REG_ERR          GENERAL_ERR + 5

#define OBJ_CLS_OWNER_ERR        GENERAL_ERR + 6

#define OBJ_CLT_OWNER_ERR        GENERAL_ERR + 7

#define PICKLING_OBJ_NOT_EXIST   GENERAL_ERR + 8


#define STRING_SEPARATOR                '#'


typedef  struct {
  unsigned        init_cls:  1;          /* 1 OR 3 => INIT CLS*/
  unsigned        init_obj:  1;          /* 2 =>      INIT OBJ */
  unsigned        unuse:     30;
} RunningFlag;


typedef union {
  RunningFlag   rflag;
  int           iflag;
}uFlag;



typedef  struct {
```

```
  FILE    *sock_id;
  int     flag;
} SocketStruct;


typedef struct {
  int             msgid;
  char            cltname[MAX_CLIENTNAME_LEN];
  char            clsname[MAX_CLASSNAME_LEN];
  char            objname[MAX_OBJNAME_LEN];
  char            owner[MAX_OBJNAME_LEN];
}ComHeaderStruct ;


typedef struct {
  int             msgid;
  char            cltname[MAX_CLIENTNAME_LEN];
  char            xprogname[MAX_FILENAME_LEN];
  char            adminPassWord[MAX_PASSWORD_LEN];
  uFlag           u;
  int             num_objs;
  int             num_cls;
}UiComStruct;
```

```
typedef union {
  struct {
    ComHeaderStruct     hd;
    int     dlen;
    char              data[MAX_DATA_BUF];
  } com;
  char    recvbuf[MAX_BUF_LEN];
} cppDataComStruct ;



typedef struct {
  char *clientname; /* progname */
  char            *sock_addr; /* /tmp/progname */
  SocketStruct    send_sock;
  int             send_msgid;
  SocketStruct    recv_sock;
  int             recv_msgid;
  cppDataComStruct          u;
  char *format;
} Session;
```

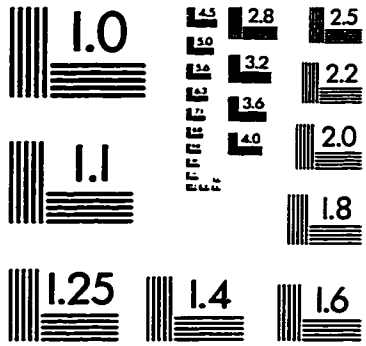
Appendix C

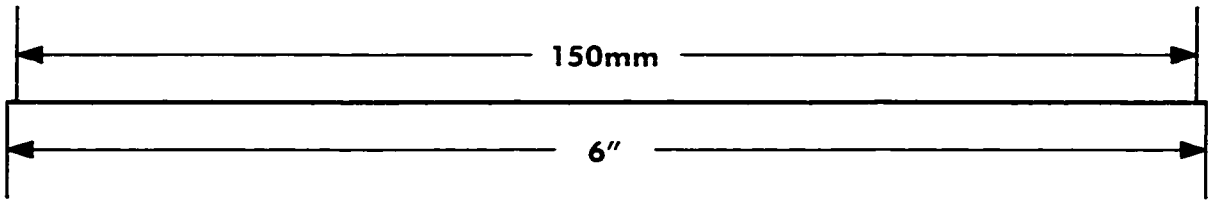
API Header File

```
/*============================================================*/
/*       Project     :   Pickling System                      */
/*       Filename    :   cpplib.h                             */
/*       Discription :   header file needed for a C++ client*/
/*                       program                              */
/*                                                            */
/*                       Ba-Nguyen, Tran                      */
/*============================================================*/

#include <string.h>
#include "comutils.h"
#define PK_output stdout
```

```
int PK_register_obj(char *cltname, char *owner, char *clsname,
                    char *objname, char *buf, int *sendlen);
int PK_retrieve_obj(char *cltname, char *owner, char *clsname,
                    char *objname, char *buf, int *sendlen);
int PK_store_obj(char *cltname, char *owner, char *clsname,
                    char *objname, char *buf, int *sendlen);
int PK_terminate(char *cltname);
```

TEST TARGET (QA-3)





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989