University of Durham

School of Engineering and Applied Science
(Computing Science)

A comparative study of structured and un-structured remote data access in distributed computing systems

WAI CHUNG TANG

A thesis submitted for the Degree of Master of Science

Volume 1

Abstract

Recently, the use of distributed computing systems has been growing rapidly due to the result of cheap and advanced microelectronic technology. In addition to the decrease in hardware costs, the tremendous development in machine to machine communication interfaces, especially in local area networking, also favours the use of distributed systems. Distributed systems often require remote access to data stored at different sites. Generally, two models of access to remote data storage exist: the un-structured and structured models. In the former, data is simply stored as row of bytes, whereas in the latter, data is stored along with the associated access codes. The objective of this thesis is to compare these two models and hence determines the tradeoffs of each model. First of all, an extended review of the field of distributed data access is provided which addressing key issues such as the basic design principles of distributed computing systems, the notions of abstract data types, data inheritance, data type system and data persistence. Secondly, a distributed system is implemented using the persistent programming language PS-algol and the high level language C in conjunction with the remote procedure call facilities available in Unix[1] 4.2 BSD operating system. This distributed system makes extensive use of Unix's software tools and hence it is called DCSUNIX for Distributed Computing System on UNIX. Thirdly, two specific applications which employ the implemented system will be given so that a comparison can be made between the two remote data access models mentioned above. Finally, the implemented system is compared with the criteria established earlier in the thesis.

keywords: abstract data types, class, database management, data persistence, information hiding, inheritance, object oriented programming, programming languages, remote procedure calls, transparency, and type checking.

---

[1] Unix is a trademark of Bell Laboratories.

Preface

This thesis comprises eight chapters and is in two volumes. This volume contains five chapters aiming to present all the background knowledge of the thesis. They are briefly summarized as follows:

Chapter one puts forwards the main motivations of this thesis. Chapter two introduces the basic design issues of distributed computing systems. However, since the issue of remote file server is central to this thesis, the discussion of this topic is developed further in chapter three. Chapter four discusses the notion of abstract data types, followed by the concepts of data type systems and data persistence in chapter five.

Syntax Notation

The following syntax notations are used throughout the thesis:

(a) All the important phrases, terms and keywords of a programming language are in **bold** letters unless stated otherwise.

(b) All the phrases or terms in *italic* text are entries of a glossary section (in volume 2) where a brief definition can be found.

Acknowledgement

I would like to express my gratitude to my supervisor Professor K.H. Bennett for his invaluable advice and continous encouragement.

Also, I wish to thank all the staff in the Computer Science Department for providing so many excellent computing facilities.

Finally, I am deeply indebted to my parents for giving me all their love and support.

Contents

Chapter One -- Motivations

Conceptually, a computing system can be regarded as having several layers of abstraction as shown in Figure 1.1:

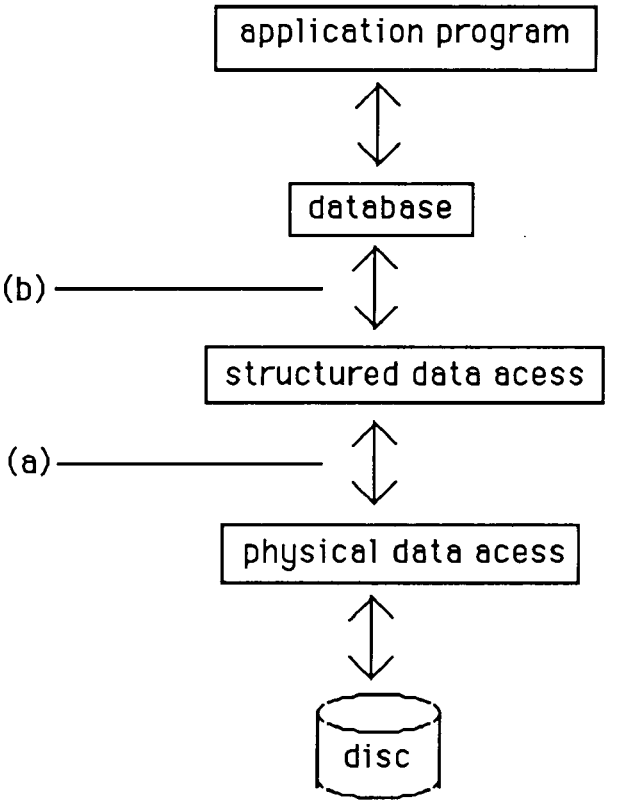
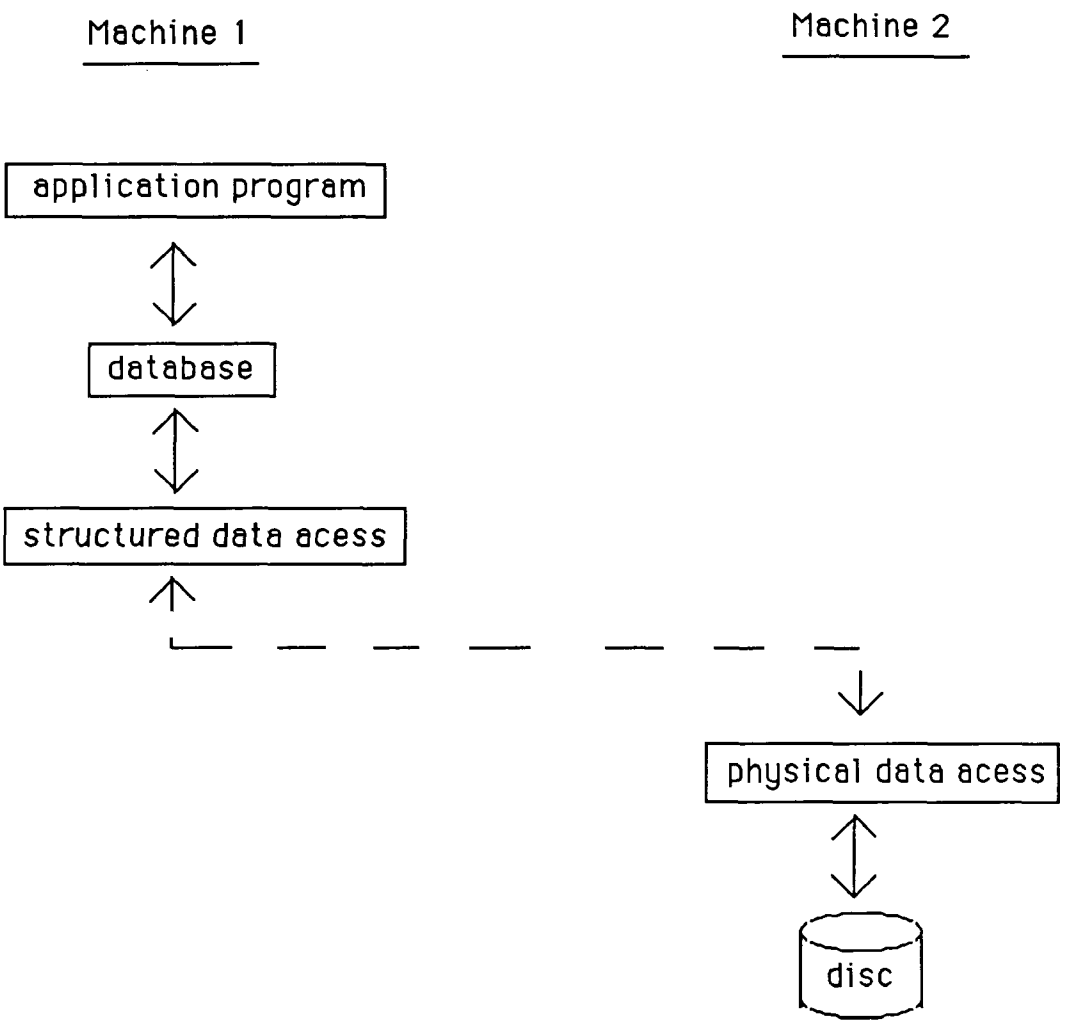


Figure 1.1 Conceptual view of a computing system

This diagram also depicts the route by which data travel between a user program and the physical storage device. However, this route is **bi-directional**, as indicated by the double-headed arrows, because the same underlying data access model will be used for all kind of data transactions, eg. data retrieval, data insertion and so on. Also, in the diagram, the **database** layer refers to a repository where a program and its associated variables are stored. This layer of abstraction serves a different role from the lower two layers which are concerning with the way data is accessed. The first layer, **structured data access** layer, is often implemented by a modern programming technique called **abstract data type** (ADT),

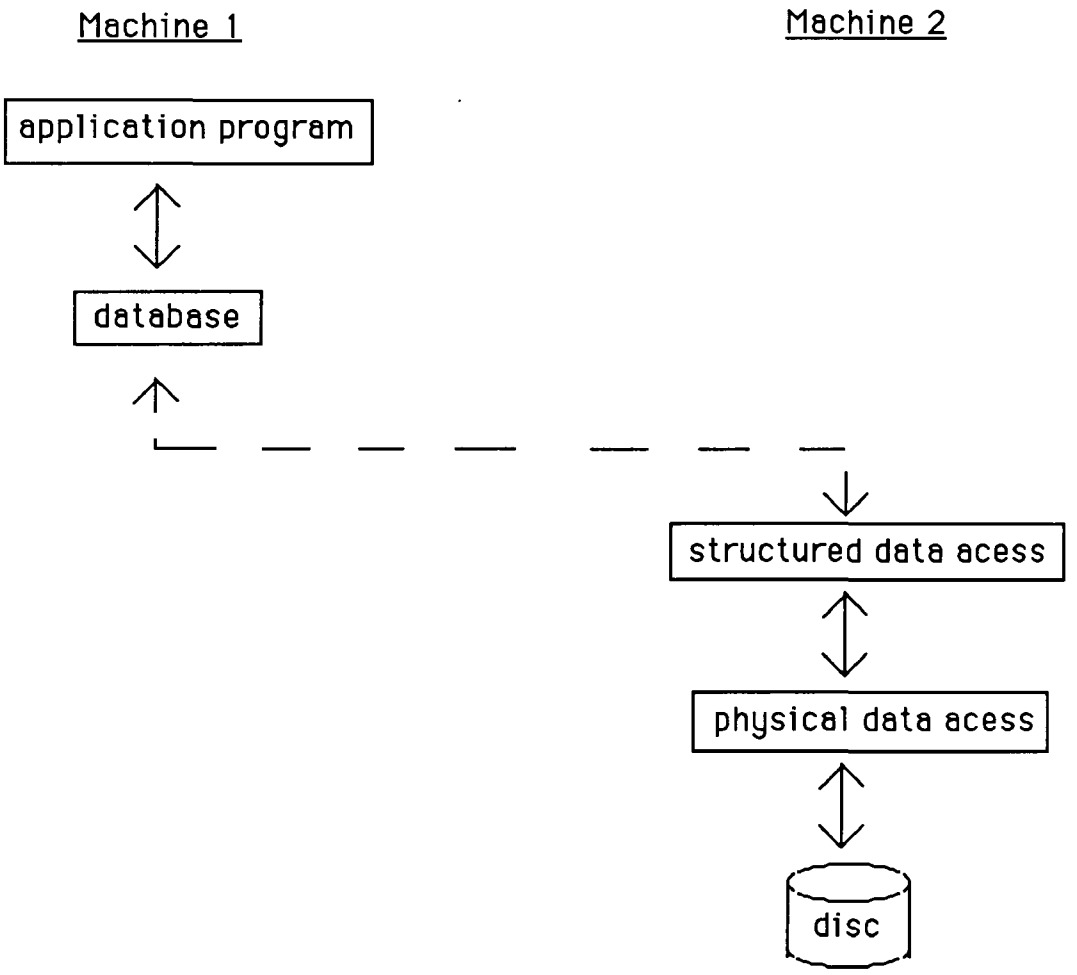
which will be described in chapter four; whereas in the **physical data access** layer, data is simply treated as rows of bytes.

When the above abstraction idea is to be established on a **distributed computing system** (DCS), Figure 1.1 is usually broken in two places. If it is broken in position (a), the following situation will be incurred:



This gives the row of bytes unstructured remote access, or referred as the **unstructured data access model** in this thesis. In this model, data is accessed directly between two (or possibly more) remote machines. But if Figure 1.1 is broken at position (b) as shown below, this gives remote access to structured *data* and will be referred as the **structured data**

access model subsequently.



This model is quite different from the previous one because both the data and its associated access codes are stored only in one machine this time. So, to access the data, the appropriate access procedures have to be called from another machine.

The discussion of the above two access models has pointed out one major design issue of DCSs, that is, how to access remotely stored data. Obviously, this involves transmission of data between different *nodes* via a communication medium, eg. the Ethernet. Unfortunately, data can be very simple or complicated depending on the application and therefore sometimes the unstructured model may be in favour to the DCS designers due to its simplicity. But this does not mean that the structured model is useless especially in terms of **security**. So each of

these models has its own tradeoffs. Among them, the key factors are concerned with data type checking and data transfer rate. These factors will affect the flexibility, reliability, security and performance of a distributed system as illustrated by the airline reservation example in Figure 1.2.

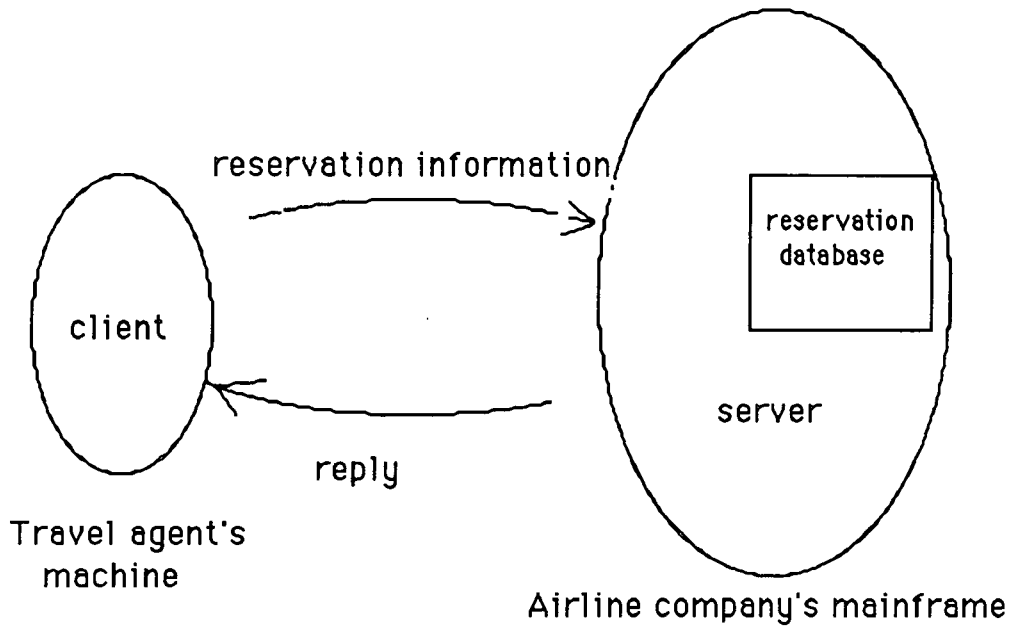


Figure 1.2 An airline reservation system

Before any further discussion, the two terms **client** and **server** in the above diagram, which will be used frequently throughout this thesis to describe remote data access mechanisms in DCSs, are defined as follow:

server: A *node* in which the representation for a given object type and the operations on this representation are implemented.

client : A process which accesses a resource or an object (information) in a server provided that the server and the client do not reside in the same machine.

Referring back to Figure 1.2, if a ticket for a particular flight is being requested by a customer, the following sequence of activities will be typically encountered:

client side

server side

(1) Construct the details for the booking.

---------->

(2) Analyse the received information. Two alternatives may exist:

(a) The information is treated as rows of characters (eg. Unix file structure).

(b) The information is treated as a unit with a well-defined structure (similar to the Pascal-like records). For instance, the first field of the unit must be a string of 30 characters, the second field is an integer etc.

(3) If no mistake is found after the above analysis, the system database will be updated and an acknowledgement is replied, else only an error message is replied.

<----------

(4) If an error message is received, go back to step (1), re-construct the booking information and then repeat the whole process.

In the above schema, the decision in choosing which of the two methods to do the analysis is totally related to the internal specification imposed by the database organization. Here, the term database means simply a repository for data. If the organization is an unstructured one, the rows of characters option should be used, otherwise the alternative must be used in oder to achieve consistency. Consistency is particularly important for the un-structured pair due to its lack of protection mechanism, i.e. clients can interpret the information at their own will. Apart from the way information is analysed, another crucial factor for the airline system is the time taken for each transaction (update). For example, if the flight is a very popular one, it is likely that several customers attempt to make a reservation at the same time while the system database is being updated. This may lead to a situation where two customers are dealing with two different versions of the database at that instant. One obvious solution is to introduce *locking* on the database to ensure no access can be made when it is busy. Finally, the application program in the client's side should also be able to aware of system crash, especially during a transaction, for instance, using a suitable time-out interval.

From the aboved airline example, the choice between the structured and un-structured data access approaches is vital to DCS designers. The main objective of this project is to use a carefully implemented distributed system to compare the structured and un-structured strategies so that DCS designers can have a better overall picture in their mind. However, problem of system faults such as node failure, loss of data, data duplication are outside the scope of this thesis.

The rest of this thesis is arranged as follows: Chapter two introduces the basic design issues of DCSs. However, since the issue of remote *file server* design is central to this thesis, the discussion of this topic is developed further in chapter three. Chapter four discusses the notion of abstract data types, followed by the concepts of data type systems and data persistence in chapter five.

Having provided the required background knowledge for the project in the first five chapters, a distributed system based on the programming languages PS-algol (introduced at the end of chapter five) and C is described in chapter six. Chapter seven examines the tradeoffs of the two data access models by presenting two detailed applications on the implemented system. Finally, chapter eight concludes the work of this thesis.

Chapter Two -- Distributed Computing Systems

Nowadays, distributed computing systems (DCSs) have become very popular both in industry and in many University research projects. The key advantage of using such systems is to make use of multiple processors without user notice. In other words, the users view the system as a **virtual uniprocessor**, not a collection of distinct machines. As a rule of thumb, if a user can realize which computer is being used, it is not a distributed system. The user of a true distributed computing system should not know (or care) on which machine (or machines) programs are running, where files are stored and so on. This is known as **location transparency**. Other important attributes of a DCS includes: replication transparency, performance transparency, fault transparency, processing transparency etc. A full discussion on **transparency** can be found in [1]. So, it is the software, not the hardware which determines whether a system is distributed or not.

In recent years, there are two common types of DCSs that are of interest: **tightly-coupled** and **loosely-coupled** distributed systems. Both of them use multiple processors. In the former, processors share a single memory and such a system is said to be tightly-coupled because each processor has direct access to the same physical resources (shared memory) and logical resources (shared information). Moreover, in this type of systems, two processors could inhibit interrupts in their respective processors and still not be mutually excluded from accessing shared variables at the same time. The problem of race conditions between the occurrences of events and the checking for the occurrences in different processors also exists. Process synchronization is based on *semaphore* techniques. Other features in a tightly-coupled system include the existence of multiple clocks, system initialization of multiple processors, inter-processor interrupt facilities, scheduling for dynamic load balancing and the existence of both shared and local memories. Examples of tightly-coupled systems are CM* [2] and CYBA-M [3].

On the other hand, in a loosely-coupled system, processors do not have shared memories. They communicate by sending and receiving messages. This type of systems also allow inter-process communications to use the same mechanisms for processes communicating over a network as for processes communicating within the same processor. Other features include the static and dynamic allocation of processes to processors, flow control and message

routing over alternate paths. This kind of distributed systems can be dispered over a wide areas such as Arpanet [4].

Due to the tremendous development in hardware components, one can use distributed systems to design large projects that consist of many small, cheap and powerful microprocessors. Some of the well-developed DCSs are the Cambridge Distributed System [5], Locus [6], V [7] and Eden [8]. In order to highlight the power of distributed systems, the next section gives a comparison between DCSs and the well-known network computing systems of the early 1970s [9].

2.1. Difference between distributed computing systems and network computing systems

A typical example of a network computing system is the UK Joint Academic NETwork (JANET). This is a non-commercial wide area network composed of heterogenous machines. JANET can be used by members of the UK academic community for terminal access and transfer of mails, files and jobs, between academic institutions. Also, access may be made from JANET to other academic networks such as the European Academic Research Network (EARN).

Network systems have many aspects in common with the distributed ones, but they also differ in the following ways:

(a) Each computer has its own private operating system, instead of running part of a global systemwide operating system.

(b) Each user normally works on his or her own machine; using a different machine invariably requires some kind of **remote login**, instead of having the operating system dynamically allocate processes to CPUs.

(c) Users are typically aware of where each of their files are kept and must move files between machines with explicit file transfer commands, instead of having file placement managed by the operating system.

(d) The system has little or no *fault tolerance* (this issue will be discussed further later); if one percent of the personal computers crashes, then one percent of the users is out of business, therefore one percent worse performance.

2.2. System models for constructing distributed computing systems

Three models are normally used for the establishment of DCSs [10]. They are **minicomputer**, **workstation** and **processor pool** models.

In the minicomputer model, the system consists of a few minicomputers (eg. VAX[1]s), each with multiple users and each user is logged onto a unique machine, with remote access to the other machines such as the JANET system described eariler.

In the workstation model, each user has a personal workstation. Nearly all the work is done on these workstations. The SUN Microsystems' Network File System (NFS) was built on this model which supports a single and global file system, so that data can be accessed without regard to their locations.

Finally, in the processor pool model, one or more CPUs are temporary allocated to a user who needs computing power; when the job is finished, the CPUs go back into the pool to wait for the next request. The Cambridge Distributed System [5] is a typical example of this model.

---

[1] VAX is a trademark of Digitial Equipment Corporation

2.3. Design issues of DCSs

Although the notion of DCSs is so important, it covers a lot of grounds and therefore it is hard to define it just in a few sentences. But a definition is sometimes useful to help one to infer principles and guidelines for the design of DCSs, so a simplified definition is given in the glossary section. However, the best way to understand DCSs is to identify all its distinctive characteristics.

This section will be divided into six sub-sections. Since a DCS is often constructed from a number of inter-connected computers, in order to exchange information between computers, each of them must provide an **interface** through which they communicate with each other. As a result, a communication system is required. Unfortunately, these computers usually do not have shared memory (loosely-coupled), so communication is performed via the technique of **message passing**. The objective of the first sub-section is to discuss the communication primitives of the various message passing algorithms.

The second sub-section addresses the naming issue of DCSs. Naming system (sometimes referred as the identification system) is an area at the heart of all computing system designs because it is used for a wide variety of purposes such as protection, error control, resource management, locating and sharing of resources.

Following the discussion of naming in DCSs, the third sub-section presents another distinctive feature of DCSs which is concerned with **resource management**. There is a wide range of resources that require management such as processors, processes, I/O devices, communication channels and so forth. Allocating and scheduling of resources are the major tasks of a resource manager. Besides, it is also the responsibility of the resource manager to ensure that no deadlock occurs. All these issues will be discussed in details.

In the fourth sub-section, the notion of **fault-tolerance** will be described. Baasically, a fault-tolerant distributed system is the one which can tolerate certain kind of **expected** errors, typically node crashes and communication failures such as

lost messages. Also, in this sub-section, two widely used fault-tolerance techniques will be discussed.

The fifth sub-section concentrates on the issue of **file system** which is a vital component in any distributed systems. Apart from providing a storage place for data, it also acts as an extension of main memory and as a means for inter-user communication.

To conclude this section, a list of user-level services are given in the last sub-section.

2.3.1. Communication primitives

2.3.1.1. Communication models

2.3.1.1.1. The client-server model

Underlying communication between different machines in a distributed computing system is via the technique of **message passing**. The most common model is the **client-server** model in which a client process sends a request to the server and then waits for a reply message as shown in Figure 2.1 below:

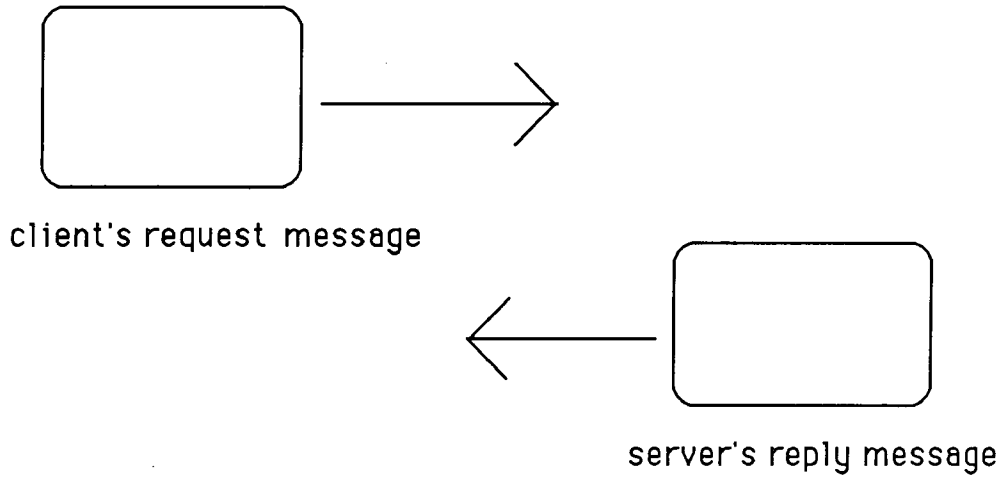


Figure 2.1 Client-Server communication model

The system should provide at least two primitives, SEND and RECEIVE, to accomplish this model. The SEND primitive specifies the destination and provides the message; the RECEIVE primitive specifies (or listen) from whom a message is desired and provides a buffer where the incoming message to be stored. So, it can be realized that this model does not need any initial setup.

According to Liskov [11], three important decisions have to be made. The first decision is about **reliable** and **un-reliable** transmissions. In the former, the SEND primitive will handle lost messages, re-transmissions and acknowledgements internally so that when SEND terminates, the program can make sure that the message has been received and acknowledged. However, for un-reliable transmission, the SEND primitive can only put a message out into the network without guarantee of delivery and no automatic re-transmission. Thus, in terms of processing power, the reliable approach is often more expensive than the un-reliable one. But on the other hand, the un-reliable strategy is more difficult to implement due to its complexity.

The next concern is about **blocking** and **non-blocking**. For non-blocking SEND, it returns control to the user program as soon as the message has been queued for subsequent transmission (or a copy is made), the corresponding RECEIVE will signal a willingness to receive a message and provides a buffer for it to be placed. The only advantage of these non-blocking primitives is that they provide the maximum flexibility: message I/O can be computed and performed in parallel. However, a big drawback in non-blocking primitives is that they make programming difficult because timing dependent programs are hard to write and debug. Therefore, blocking primitives are preferred at the expense of less flexibility. A blocking SEND does not return control to the user program until the message has been sent (un-reliable blocking primitive) or until the message has been sent and an acknowledgement is received (reliable blocking primitive). Either way, the program can immediately modify the buffer without danger. A blocking RECEIVE does not return control until a message has been placed in the buffer. Reliable and un-reliable RECEIVEs differ in that the former automatically acknowledges the receipt of a message while the latter does not.

The last decision is the choice of **buffered** or **un-buffered** messages. In the latter, the sender is blocked until a RECEIVE has been done, at which time the message is copied from the sender to the receiver. This is sometimes called **rendezvous**. In the former, senders can have multiple SENDs outstanding, even without any interest on the part of the receiver. The *kernel* of the system

provides a kernel buffer, sometimes referred as **mailbox**. To communicate, the sender sends messages to the receiver's mailbox, where they will be buffered until requested by the receiver. Buffering is more complex such as creating, destroying and managing mailboxes, the need for special high-priority interrupt messages protection against the mailbox, what to do with a dead mailbox, etc.

2.3.1.1.2. The remote Procedure Call technique

Another form of message passing mechanism which has been used widely recently is known as **remote procedure call** or RPC for short. Essentially, RPC is a distributed programming facility which enables a *client* program to call procedures whose codes and parameters are residents on a remote machine. After the invokation, the client blocks while an underlying mechanism, the RPC protocol which atcs as an interface between the client and the remote machine, will take care of the following activitives:

(a) sending a call message to the remote machine,

(b) initializing a computation on the remote machine which will execute the called procedure,

(c) and finally, transmitting the computation result back to the calling program in the client's machine.

Ideally, it would be desirable to treat RPC as a local procedure call so as to make the distribution transparant to the user. For instance, Figure 2.2 shows a typical RPC design for a DCS.

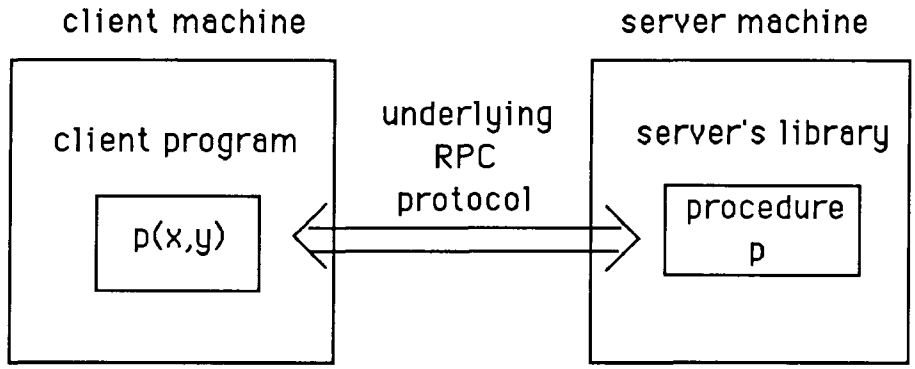


Figure 2.2 RPC communication model.

In this model, the client (calling program) first makes a normal procedure call, say p(x,y) on its machine, with the intention of invoking the remote procedure p on some other machine. Secondly, a dummy or stub procedure is included in the caller's address space. This procedure, which may be automatically generated by the compiler, collects the parameters and packs them into a message. Thirdly, the message is sent to the remote machine via an output handling procedure and then the calling program is suspensed by the kernel to wait for an answer. At the remote machine, another stub procedure, which was created after procedure p had been registered on the server machine, should be waiting for a message invoking p. Once this is detected, the parameters of the message will be un-packed using an input handling procedure and then makes the **local** call p(x,y). So, the only procedures that know the call is remote, are the stubs. After the result of the call of p has been obtained, it will be returned to the client with an analogous path in the reverse direction. All the stub procedures and I/O handling routines will be parts of the underlying RPC protocol as indicated in the diagram.

Although the RPC scheme seems to work fine, some problems still exist. The first problem is how parameters and results are passed over the network. Usually they can be passed either **by value** or **by reference** (pointers). Passing by value is easy because the stub procedure just copies the data into the message. However, passing by reference needs a unique, systemwide pointer for

each object so that it can be accessed remotely. For large objects, such as a payroll file, some kind of *capability* mechanisms [12] could be set up and then using these capabilities as the pointers. Unfortunately, the amount of overhead and the algorithm required to create a small object, such as an integer or a boolean, are so great that it is un-satisfactory in practice.

Another problem is how parameters and results are represented. This can be quite complicated particularly if several different types of machines are involved in the communication as each of them may use different internal representation for the same set of objects. For example, a floating point number produced on one machine may not have the same value on a different machine or even a negative decimal number will create problems between the 1's complement and 2's complement machines. One obvious solution is to convert every incoming and outcoming messages into a standard format. But it is quite expensive and time-consuming especially if the sender and receiver machines are actually using the same internal format. An alternative solution is that the sender uses its internal format to represent parameters and results, and then an identification code is included in every messages to indicate which format it is. When the receiver receives the message, a conversion will be taken place according to the identification code. Thus, every machine must be prepared to convert from every other formats. However, when a new machine type is introduced, much of the existing software must be up-graded. So, whatever the solution is, with RPC, certain difficulties exist.

The issue of how clients bind with servers is also important. Consider a distributed system with multiple servers. If a client creates a file on one of the file servers, it is usually desirable that subsequent writes to that file go to this specific file server directly. But using RPC, the client will only make a call such as

write(Filedescriptor,BufferAddress,NumberofBytes)

in his or her program with no information about where the file is. Nevertheless, this will work if a table is maintained by the system which contains the location details of each file-descriptor.

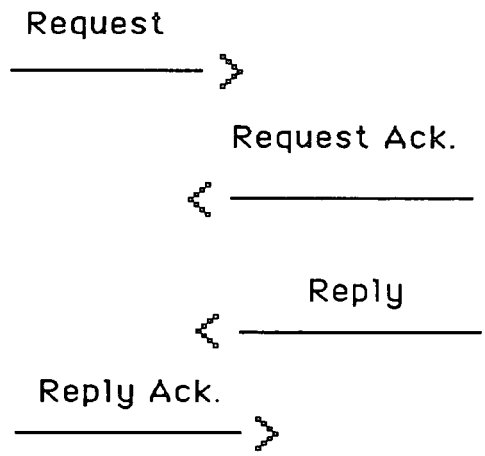
2.3.1.2. Implementation issues about communication primitives

In the last sub-section, two most commonly used message passing techniques have been described. Another widely discussed strategy for message passing is the ISO-OSI reference model [13] which has seven layers, each performing a well-defined function. By using this model, it is possible to connect computers with a wide range of operating systems and hardware architectures. Unfortunately, the overhead caused by all the seven layers is substantial. This overhead could still be tolerable for distributed systems consisting of huge mainframes, connected by low *bandwidth* communication lines. But for a system that is composed of only small computers, such as microcomputers, connected by fast local network, the price to pay for the ISO model is too high. This observation leads to a great deal of research work in making message passing as efficient as possible.

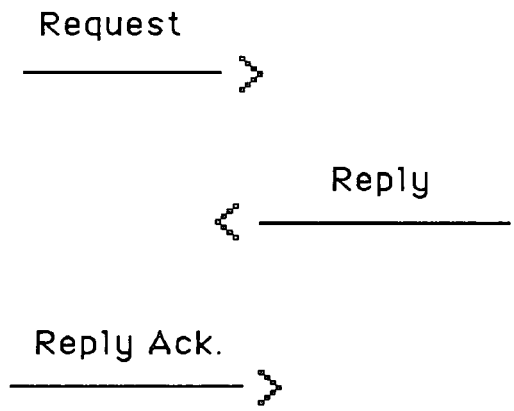
One major implementation issue of communication primitives is to minimize the copying of data because the time for copying excessive data and the other sources of overhead eg. the reply message, the time waiting for access to the network, transmission time and so on, will slow down the *throughput* of the system. Also, another important implementation issue to be considered is the substantial fixed overhead with preparing, sending and receiving a message (even for a short one), eg. a request to read from a remote file server. In these circumstances, the kernel must be invoked, the state of the current process must be saved, the destination must be located, the various tables must be updated, permission to access the network must be obtained (eg. wait for the network to become free or wait for a token), and also some book-keeping must be done. This fixed overhead argues for making the message as long as possible to reduce the number of messages. However, if the message are too long, a highly interactive user may occasionally be delayed and hence degrading the *response time*. So, the optimum size should depend on the work load.

Besides the above issues, there is also much controversy over the past years about whether RPC should be built on top of a *virtual circuit* or a *datagram*. Saltzer et. al. [14] have pointed out that since high reliability can only be achieved by end-to-end acknowledgements at the highest level of a communication protocol, the lower levels need not be 100 percent reliable. The overload incurred in providing a clean virtual circuit upon which to build RPC is therefore wasted. This suggested for building RPC on a raw datagram interface. But on the other side of the coin is that it would be nice if a distributed system can encompass heterogeneous computers in different countries with different post, telephone, telegraph networks and possibly different national alphabets, although this environment requires complex multi-layered protocol structures. So both arguments seem to be valid depending on whether one is trying to forge a collection of small computers into a virtual uniprocessor or merely access remote data transparently.

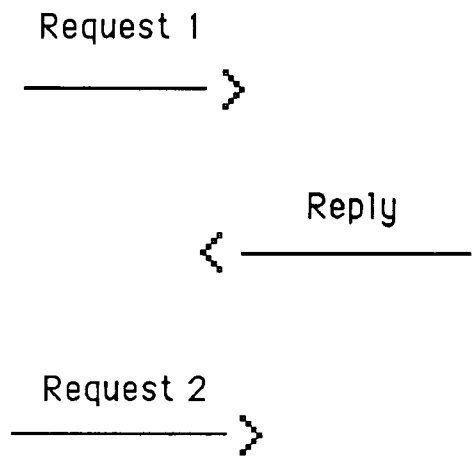
Unfortunately, further consideration is still required even though datagram is chosen to implement RPC. There are currently several *protocols* where datagram services are implemented. The simplest one is to have every request and reply separately acknowledged as shown in Figure 2.3(a). The ACKs are managed by the system kernel without user knowledge. However, the number of message can be reduced from four to three by allowing the REPLY to serve as the ACK for the request as depicted in Figure 2.3(b). But a problem arises when REPLY is delayed for a long time, eg. if a login process sends a RPC to a terminal server asking for characters, it may take a long time before someone types something on it. In this case, an additional message has to be included to allow the sending kernel to inquire whether the message has arrived or not. Finally, it is also possible to eliminate the other ACK, as shown in Figure 2.3(c). This time lets the arrival of the next REQUEST imply an acknowledgement of the previous REPLY.


Request
Request Ack.
Reply
Reply Ack.

(a)


Request
Reply
Reply Ack.

(b)



(c)

Figure 2.3 Remote Procedure Call (a) with individual acknowledgements per message, (b) with the reply as the request acknowledgement, (c) with no explicit acknowledgements.

Nevertheless, even a straightforward implementation of an efficient communication scheme can have unexpected consequences as suggested in [15]. Consider a ring containing a circulating token. To transmit a message, a machine captures and removes the token from the network, then puts the message on the network before it replaces the token to allow other users to transmit their messages. This scheme looks fair but suppose two users each wants to read a long file from a file server. User A sends a request to the file server and then replaces the token on the network for B to acquire. After A's message arrives at the server, it takes a short time for the server to handle the incoming message interrupt and re-enable the receiving hardware. So the server is deaf until the receiver is re-enabled. Within this short period of time, A may replace the token and let B grab it and sends its request to the temporary deafed file server. Even if the server has re-enabled halfway through B's message, the message will still be

rejected due to missing header, bad format and so forth. Furthermore, according to the ring protocol mentioned earlier, after sending a message, B has to replace the token and therefore A can grab it again to make another transmission. Once again B transmits during the server's deaf period, the same result happens. This would go on and on until A has read the whole file. So B will get no service at all until A has finished. Sventek [15] claimed that the problem can be solved either by inserting random delays in places to break the synchrony or using a buffer to store the messages. From the result of this example, it is necessary to build and observe real systems to gain insight of the problems rather than totally rely on abstract formulations and simulations.

2.3.1.3. Error handling

Error handling in distributed systems is quite complicated partly due to the lack of centralized information about running processes and partly due to the massive number of processors involved. It is often quite difficult to find out what has gone wrong with the system. For instance, if a client process has initiated a RPC with a remote server which crashes after the call is accepted, the client may just be left hanging forever for a reply and become an **orphan** process unless a timeout is set on the call or an **exception** is signalled. However, the former may introduce race conditions as the client may timeout too quickly due to the slow servers whereas in the latter, the RPC is said to be returned **abnormally**. Conversely, if a client crashes after it makes a RPC, an orphan process will be left in the server machine because the result of the RPC cannot be returned. Even worse if the server does not know when it is safe to discard the result of the RPC which may not be reproducible later.

Another problem is what happens if a client cannot be sured whether or not a server has crashed One solution is to wait until the server has rebooted and then try again. This solution sometimes works and sometimes does not. This will work if and only if the client's request does not cause any serious consequence such as reading ten bytes of a file but not withdrawing one million pounds from a bank.

Going back to the orphan problem, apart from using up some of the valuable system resources, another danger consequence caused by orphans is concerned with their **interference** with other legitimate computations. As an example, suppose a client obtains a lock on a remote terminal and has issued two RPCs, say DISPLAY(file1) and DISPLAY(file2). If DISPLAY(file1) returns abnormally and leaves an orphan which is allowed to interfere with the computation for DISPLAY(file2), then the lines of the two files could freely mix on the terminal screen. In order to solve these problems, the following notion of RPC **call semantics** are generally adopted:

Exactly-once semantics : When a RPC returns, the called procedure is known to have been executed exactly once if the return from it is successfully, or partially up to the point when an exception was raised if the return is abnormal. This semantic can be used to detect orphans.

At-least-once semantics : For some particular applications, one might settle for allowing more than one computation to take place on the behalf of a single RPC. This semantic is used to detect interference.

At-most-once semantics : Under this semantic, a RPC returns successfully iff (if and only if) it has given rise to one execution; abnormally iff it has had no effect. Sometimes this semantic is referred as **atomicity** and it is mainly used to deal with node crash recovery problems.

Conclusively, orphans are un-wanted executions caused by communication or node failures. Mancini [16] has mentioned several methods to treat orphans. Recently, the issues of **orphan-detection** and **orphan-killing** have received a great deal of attention owing to the fact that orphans are the main source for interference to occur in DCSs. **Crashcounts, extermination** and **expiration** are three of the best known strategies for interference prevention through orphan-killing. In the paper given by Nelson [1], the relative merits of these three methods have been compared and he has also pointed out that a reliable DCS should be able to control interference transparently to the users.

2.3.2. Naming systems

Naming system (or identification system) is of fundamental importance to the construction of a DCS. A name is actually an identifier, typically a character string or an integer, used to identify an object such as a file, a directory, a *node*, a service and so forth. To provide an un-ambiguous way of identifying objects, the idea of **unique identifier** has become very popular. A unique identifier is a name in a global context that names all the objects in the system. An example of a unique identifier is the CPU serial number which is a fixed length integer carefully chosen to be large enough to exceed the number of objects even likely to be created by the system. Shoch [17] has further categorized identifiers into three classes: **names**, **addresses** and **routes**. A name refers to an object; an address refers to an object's location in memory; and a route refers to the means of finding the address of an object. All these three classes of identifier can occur at all levels of a system architecture. Throughout the rest of this sub-section, Shoch's definition for name will be adopted wherever the term name is used.

In the following three sub-divisions, the basic principles of naming will be discussed, followed by the design issues of name servers and their goals in DCSs.

2.3.2.1. Objectives of naming

Generally, all naming systems are based on the following principles :

(a) **Identification** : Objects are identified by names without reference to their properties. Naming provides a unique method for mapping between an object and its **identity**. Thus, naming can be viewed as a way of abstracting over identities. The process of replacing a name with the identity of its corresponding object is called **name resolution**. However, in some cases, a given object may have more than one name or two distinct objects may have the same name and therefore the meaning of a name will depend on the *context* in which it is resolved.

(b) **Data protection** : Before any object is used, it must be identified. If an identified object is to be manipulated or accessed, the identifier must be mapped, using the appropriate mapping function, into the corresponding address in memory. The mapping function may contain information concerning the access rights of the object. This strategy has ensured that objects are managed in a controlled way and hence data is protected from misuse.

(c) **Resource sharing** : Communication between two processes can be taken place by passing around the name of a common object. For example, assuming that there are two processes: one calculates the average of the current DOW-JONES stock prices in New York and the other makes changes to a database (an object) where the current DOW-JONES stock prices are kept. In order to fulfill the tasks of both processes, the stock prices database must be shared by passing its name as a parameter. Although names permit sharing, it is not always in the most desirable way especially if the use of a shared object requires the user to have knowledge about the names of the other objects associated with the shared object.

(d) **Transparency** : Since one can use names to access objects, the users are shielded from the implementation details of the objects. This provides the basis for building abstractions over certain system's identities such as the locations of the objects. This kind of location transparency is of particular interest to distributed systems.

(e) **Structure description** : Names can (if desired) serve to describe objects to which they refer. In other words, names reflect the similarity in their referents; similar names refer to similar objects such as the hierarchical file structure of Unix. Moreover, this descriptive property of naming can be used to draw the distinction between **human-oriented** names and **machine-oriented** names. In the latter, names are implemented in terms of bit pattern which can be easily manipulated and stored by machines and be directly useful with protection such as Unix's

inode number. On the other hand, human-oriented names are usually human readable character strings with mnemonic value (eg. the name of a machine) and at the some stage they must be mapped into the machine-oriented ones. This shows the fact that human-oriented names are essential for the higher levels of the system whereas machine-oriented names are useful in the lower levels and always invisible to users.

Unfortunately, naming imposes additional difficulties in distributed systems because in some cases, mapping only implies a single level of mapping but in other cases, it can imply multiple levels. Considering a process wants to use a printing service, it may first have to map the service name into the name of a server process that is prepared to offer the service. Secondly, the server process may then be mapped onto the number of the CPU on which it is running. However, the mapping may not be unique if there are more than one process prepared to offer the same service. These problems are the major objectives of a distributed name server as described shortly.

2.3.2.2. Name servers

Recently, several mechanisms have been attempted to implement distributed naming system. One strategy is to employ a central authority called the **name server** which is often one of the processors within the distributed system. The name server accepts names in one domain and maps them onto names in another domain. For instance, to locate a service, one sends the service name (in ASCII string) to the *node* number where that service can be found. To facilitate the mapping, the name server needs to build a database by registering services, processes, etc, that want to be publicly known with the exception that file directories are regarded as a special case of name service because it is supported by the operating system. This approach is only acceptable by small computing systems such as Smalltalk-80 [18]. For large distributed systems, it is undesirable to have a single centralized component (the name server) which will bring the efficiency of the whole system down when it is overloaded.

Another approach is to partition the system into domains, each with its own name server. One way to organize the system is to have a global naming hierarchy tree structure with files and other objects having names of the form /a/b/c (a Unix-like system). When such a name is presented to a name server, it can immediately route the request to some name server which will send it to another name server and so on until it reaches the name server in the network where the required object is located and the mapping can be done there. Another way to organize the name servers is to have each one effectively maintian a table of name-pointer pairs where the pointer is some kind of *capability* for objects in the system.

So when a name such as a/b/c is looked up by the local name server, it may yield a pointer to another domian (another name server), to which the rest of the name b/c is sent for further processing. This facility can be used to provide links to files or objects whose location is managed by a remote name server. For example, if a file TEST is located in another local network, n, with name server n.s. One can make an entry in the local name server's table for the pair (x,n.s) and then access x/TEST as it were a local object. Any appropriate authorized user or process can also make its synonym, say s, and perform the access using s/x/TEST. Finally, the most extreme way of distributing the name server is to have each machine managed its own names. To look up a name, one broadcasts it on the network. At each machine, the incoming request is passed to the local name server which replies only if it finds a match.

Although name servers frequently maps a string onto a number, which is used internally to the system such as a process identifier or node number, the reverse mapping is sometimes useful in situation like machine crash; upon rebooting, it could present its node number to the local name server to inquire what it was doing before the crash so that the server can take the appropriate remedial actions.

2.3.2.3. Goals of distributed name servers

Distributed and non-distributed name server differ in that the former provides the ability for one machine to identify an object stored on another machine. This is often considered to be a major characteristic of a DCS. Every (local or remote) objects must be identified before they are accessed, and therefore the efficiency of the name server is directly proportional to the overall performance of the system. Although distributed name servers are so important, no standard has yet been imposed on thier specifications. Nevertheless, the following goals are worth aiming at when designing name servers for distributed computing systems.

(a) Support at least two levels of names (identifiers), one convenient for users and one convenient for machines. They are human-oriented names and machine-oriented names respectively. Also, there should be a clean separation of mechanisms for these two levels of identifiers.

(b) Provide a system viewed as a global space of identified host computers containing locally identified objects. Similarly, the naming mechanism should be independent of the physical connectivity or topology of the system. This means that the boundaries between physical components and their connection as a network should be invisible to certain extent and be recognized as logically artifical. If these boundaries are represented in identifiers, it should be within a unified identifier structure such as a hierarchical form.

(c) Support relocation of objects by means of a mechanism which can update the appropriate *context* for the required mapping when an object is moved. There will be at least two levels of identifiers, a name and an address, and the binding between them would be dynamic.

(d) Support multiple copies of the same object such that if the content of the object is only going to be read or interrogated, one set of constraints are

imposed; if the content can be written or modified, another set of (usually more restricted) constraints are used. The key advantages of multiple distributed copies are that it reduces response time in case of read only access requests because objects can be stored in a node where they are used most frequently, and for higher availability/reliability in the presence of host computer crashes.

(e) Allow special purpose objects to share the same identifier. This is extremely useful to support broadcasting or to group identifiers for conferencing or some other special applications.

(f) Finally, the number of independent naming systems across and within the system architectural levels must be minimized.

2.3.3. Resource management

For the reason of reliability, distributed system designers have always rejected the idea of having any central tables, for example, to find out whether a processor is free or not. Furthermore, even if there is a central table, recent events on outlying processors may have made some table entries obsolete without the table manager knowing it. Lack of accurate global state information making the task of resource management very difficult.

2.3.3.1. Processor allocation

One of the key resources to be managed in a distributed system is the set of available processors. A typical strategy is to organize them in a hierarchy structure independent of the physical structure of the network. This approach is used by MICROS [19] which assigns some of the machines as **managers** and the

others as **workers** as illustrated in Figure 2.4. For each group of say k workers, one manager machine (the department head) is assigned the task of keeping track of who is idle. If the system is large, there will be a large number of department heads; some machines will function as **deans** managing the k department heads. This hierarchy can be extended infinitely with the number of leaves growing algorithmically with the number of workers. Since each processor only maintains communication with one superior and k sub-ordinates, the information track is manageable.

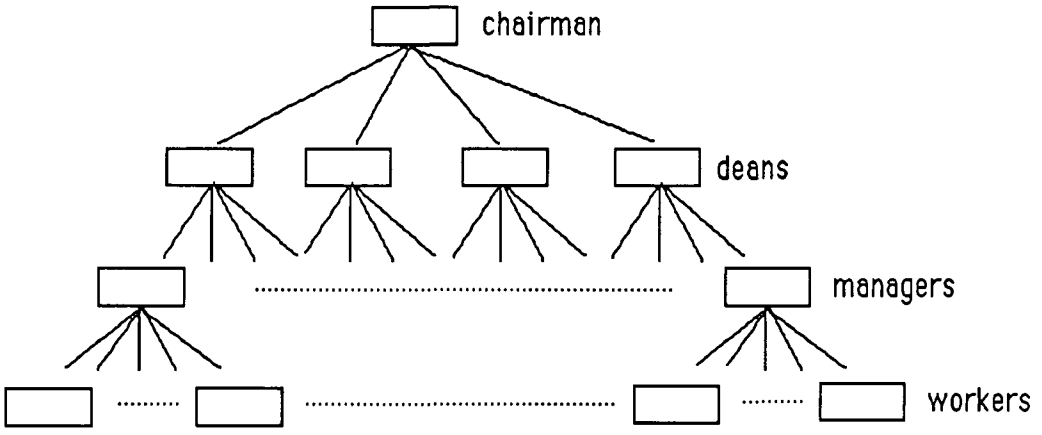


Figure 2.4 MICROS processor hierarchy structure

However, there is a problem when one of the department heads or deans has crashed. One solution is to promote one of the direct sub-ordinates of the faulted manager to fill the place. This choice may vary from case to case. Sometimes it even better not to have a single manager at the top of the tree, therefore the tree is truncated at the top, and having a committee as the ultimate authority. When a member of the committee malfunctions, the remaining members promote someone from one level down as the replacement. The system is self-repairing under this scheme and can survive occasional crashes of both workers and managers without any long term effects.

Also in MICROS, all processors in this system are mono-programmed that is if a job requires S processors suddenly appears, the system must allocate S processors for it. Since jobs can be created at any level of the hierarchy, each manager will keep track of approximately how many workers below it are available (or possibly several levels below it). If it thinks there are enough workers, it reserves some of them, say R, where R>=S in case some of the machines may be down later. However, the manager receiving the request thinks that it has not got sufficient workers for the job, it passes the request upwards in the tree to its boss. If the boss cannot handle it either, the request continues propagating upwards until it reaches a level that has enough workers or the top of the tree is reached. In the former, the manager at that level will split the request into parts and delivers them to the managers below it which then do the same process until the scheduling of the request reaches the bottom level. At this point, the processors are marked as **busy** and the actual number of processors allocated is reported back up the tree. In the latter, the request has to be suspended until there are sufficient processors available.

One of the critical factor in the above strategy is the choice for R. If R is not large enough, there will not be enough workers to finish the job and the whole process has to restart at one level upwards in the hierarchy, wasting considerable time and computing power. On the other hand, if R is too large, too many processors will be allocated, wasting computing capacity. Besides the choice of R, this method can be made complicated due to the fact that requests for processors can be generated randomly anywhere in the system, so at any instant, multiple requests are likely to be in various stages of the allocation algorithm, potentially giving rise to out-of-date estimates of available workers, race conditions and deadlocks.

2.3.3.2. Scheduling

The hierarchical model described in the previous sub-section did not specify how scheduling is done. No problem will arise if each process uses an entire processor and each process is independent of others. However, if several processes are working together and must communicate frequently with each other such as nested RPCs, it is important to make sure that the whole group runs at once.

As an example to illustrate the difficulty in scheduling, assume each processor can handle up to N processes and there are plenty of machines and N is sufficiently large. Suppose processes A and B run on one machine and processes C and D run on another. Each machine is time-sliced in say 100 millisecond, with A and C running in the even slices and B and D running in odd ones. If A sends many messages or makes many RPCs to D. During time slice 0, A starts up and calls D immediately which is not running because it is now C's turn. After 100 milliseconds, process switching takes places, then D gets A's message and carries out the task and replies quickly. However, since B is running at this time, it will take another 100 milliseconds before A gets the reply and proceed. The net result is that one message exchange every 200 milliseconds. If there are a thousand of such processes, the waiting time will grow considerably large.

Scheduling strategies:

Although it is difficult to determine dynamically the interprocess communication patterns, in many cases a group of related processes are started off simultaneously as filters in a Unix pipeline. Stone [20] has suggested a method to deal with scheduling. That is to find out all the processes in the system that are working together, so that closely related groups of processes can be placed on the same machine to reduce interprocess communication costs. On the contrary, Ousterhout [21] has proposed several algorithms, based on the concept of **co-scheduling**, which take interprocess communication patterns into

accounts while scheduling, to ensure that all members of a group running at the same time. In general, the objective of Ousterhout's work is to place processes that work together on different processors, so that they can all be run in parallel. Since Stone and Ousterhoust' proposals on scheduling, much attention has been received on the subject. As a result, another approach called **load-balancing** [22, 23] was developed which aimed to keep load uniform: to prevent a situation in which some processors are overloaded while others are relatively free or even empty.

Each of the approaches to scheduling mentioned above, have made certain assumptions about what is known and what is important. People using co-scheduling to maximize *throughput* and people trying to group processes to minimize communication costs often assume that: any processes can run on any different machines, the computation needs of each process are known in advance and the interprocess communication traffic between each pair of processes are also known in advance. However, these assumptions may be hardly met in real systems due to the dynamic changes in computing environment. Conversely, people using load-balancing to do scheduling do not make any assumption about the future behaviour of any process. So, loading-balancing provides a more flexible and adapative way for scheduling. Several load-balancing algorithms can be found in [10, pp. 437-438].

2.3.3.3. Distributed deadlocks

There are two kinds of deadlocks in DCSs known as **resource deadlocks** and **communication deadlocks**. Resource deadlocks arise when all or some set of processes are blocked waiting for resources held by other blocked processes. This kind of deadlocks is hard to be detected in distributed systems because there are no centralized tables giving the status of all resources. On the other hand, communication deadlocks occur when several processes waiting for a message from each other before they can proceed like a circular loop. For instance, process A waiting for a message from B and B is waiting for a message from C and

C is waiting for a message from A. Chandy [24] has presented an algorithm to detect such deadlocks. He has assumed that each process that is blocked waiting for a message knows which process or processes might send the message. When a process blocks, he assumes that it does not really block but instead it sends a query message to each of the processes that might send it the real message. If one of these processes is blocked, it sends query messages to the processes it is waiting for. If certain messages eventually return to the original process, it can can concluded that a deadlock exists. Unfortunately, there is no easy way to tackle distributed deadlocks, but one can always try to re-start those processes involved in a deadlock at the expense of some extra CPU time.

2.3.4. Fault tolerance

A distributed system is potentially more reliable than a centralized or a network system because the system can still work even if one instance of some critical components such as a CPU, a disc or a network interface is down. In addition to hardware failures, one can also consider software failures. Basically, there are two common types of software errors: *implementation error* and *specification error.* Distributed systems can allow both hardware and software errors to be dealt with in different ways. In order to understand these methods, the difference between fault tolerant and fault intolerant systems must be distinguished. A fault tolerant system is one that can continue functioning even if something goes wrong; a fault intolerant system collapses as soon as any error occurs.

Since distributed systems have enough resources to achieve fault tolerance, at least with respect to **expected** errors, eg. loss of packets during a transmission due to power cut, which is unavoidable. These systems can be made to tolerate both hardware and software faults by using the software to clean up the mess. Recently, two kinds of techniques are adopted to make distributed systems fault tolerance. They are the **redundancy** and **atomic transaction** techniques. Each of them deals with different types of fault tolerance.

2.3.4.1. Redundancy techniques

All these techniques aim to make a DCS fault-tolerance in the presence of a process crash. They take advantage of multiple processors by duplicating every process with a backup process on different processors. All processes communicate by means of message passing mechanism such as RPC. Whenever anyone sends a message to a process, it also sends the same message to the backup process. The system then ensures that the primary and the backup processes can continue running until it has been verified that both have received the message correctly. If one process crashed, the other can still continue running. But at this point, the remaining process must clone itself and make a new backup process to maintain fault tolerance in the future. Two big disadvantages exist in this scheme: extra processors are needed to duplicate every processes; and if processes exchange messages at a high rate, a considerable amount of CPU time is required to keep processes synchronize at each message exchange.

To solve the above problems, Powell and Presotto [25] have described a redundancy system that puts almost no additional load on the processes being backed up. In their system, all the messages sent on the network are recorded by a special **recorder** process. From time to time, each process checkpoints itself onto a remote disc. If a process crashes, recovery is done by sending the most recent checkpoint to an idle processor and informs it to start running. The recorder process then sends all the messages that the original process received between the checkpoint and the crash. Messages sent by the newly re-started process are discarded until the new process has worked its way up to point of crash, then it begins sending and receiving messages normally again without further help from the recording process.

The benefit of such scheme is that the only additional work that a process must do to become immortal is to checkpoint itself from time to time. If the recorder process has enough disc spaces to store all the messages that are sent by all the currently running processes, checkpoints can also be disposed. If no checkpoints are made, the recorder will have to replay the process's whole

history when a process crashes. When a process has terminated successfully, the recorder process can discard all the messages it received. Unfortunately, no scheme is perfect, one shortcoming of this scheme is that it requires reliable reception of messages all the time. Generally, local area network are very reliable, but occasionally messages can be lost, thus the whole scheme becomes less effective.

All the redundancy techniques discussed so far are concerned with crashes caused by hardware failure. The way they achieve fault tolerance is to allocate a spare processor and to re-start the crashed program. However, the new process may crash too, leading to the allocation of another processor and another possible crash. Eventually some kind of manual intervention will be required to figure out what is going on.

For software fault tolerance, a mechanism suggested by Avizienis and Chen [26] is to structure each program as a collection of modules, each one with a well-defined function and a precisely specified interface to the other modules. Instead of writing each module only once, N programmers are asked to program it. During execution, the program runs on N machines in parallel. After each module finishes, the machines compare their results and vote on an answer. The answer agreed by most of them is taken as the final answer of the module and then they can continue running in parallel with the next module. This method have a good chance to eliminate the occasional software fault bug. It is even better if there are some formal specifications for each modules because the answers can be checked against these specifications to prevent any non-sense answers. A variation of this idea to improve system performance is that all the processes can continue with the next module as soon as k suitable number of the machines have agreed on an answer, those that having not yet finished are ordered to drop what they are doing and just taken the value found by the k processes as the answer.

Obviously this scheme needs a lot of processors and labours, therefore it is only worth considering for large projects. Also, the whole scheme depends on the definition of the vote mechanism.

2.3.4.2. Atomic transactions

These techniques play a different role from the redundancy techniques described before. Consider the problem of crash recovery in a data storage system which is constructed from a number of independent computers, The portion of the system which is running on some individual computers may crash and then be re-started by some recovery mechanism. This may result in the loss of some information whch was present just before the crash. The loss of this information may, in turn, lead to an inconsistent state for the information permanently stored in the system. The task of **atomic transaction** is to maintain the consistency of a file system in the presence of these possible errors.

What is atomic transaction? A **transaction** is composed of a sequence of read and write commands sent by a client process to the file system. The write commands may depend on the results of previous read commands in the same transaction and vice versa. A transaction is said to be **atomic** if it possesses the following properties:

(a) the transaction is either run to completion or have no effect at all. (**failure atomicity**)

(b) if two or more concurrently executing transactions access shared data, then the effect is as if the transactions had been obeyed one after the other. (**serialization atomicity**)

(c) once a transaction terminates, the results produced are not destroyed by subsequent node crashes. (**permanence of effect**)

The idea of atomic transaction resembles the old magnetic tape system by having a master file and an update file as shown in Figure 2.5.

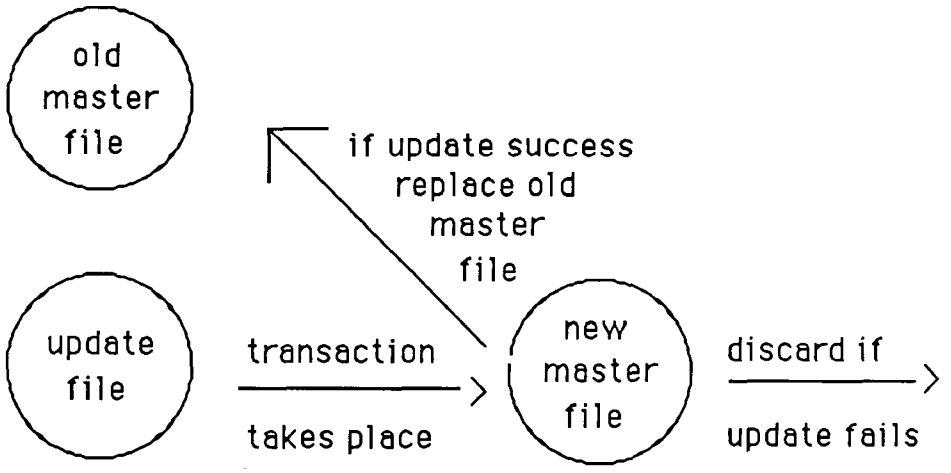


Figure 2.5 Atomic transactions

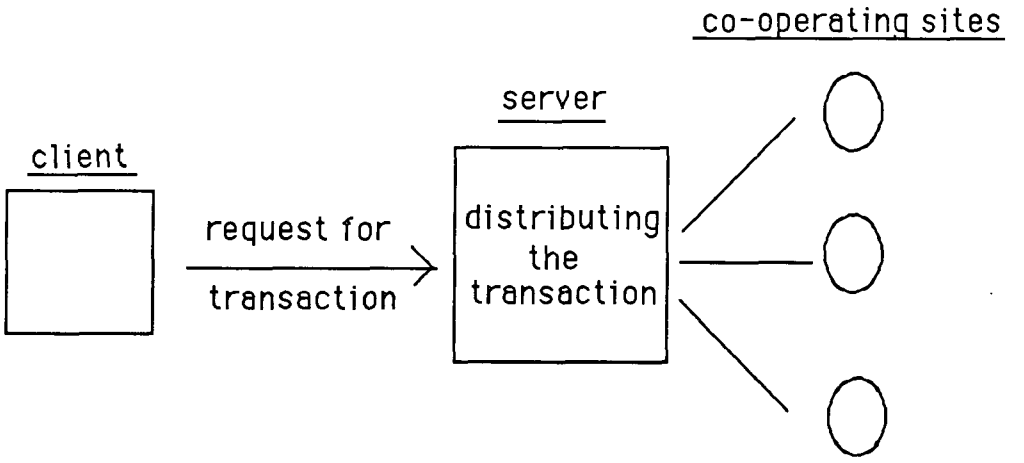
In the diagram, the master file is the old version while the update file contains all the new information. The update file is run to produce a new master file. So if the update program crashes, one could always go back to the previous master and update file. In other words, an update run can be viewed as either running correctly to completion (and producing a new master file) or have no effect at all (crash part way through, new master file discarded). Moreover, update jobs from different sources always run in some (undefined) sequential order, therefore no two users will be run concurrently.

From the three properties of atomic transaction, it can be realized that they are concerned with **recoverability, error control** and **synchronization**. By having the ability to support atomic transactions, a system can guarantee that no information will be lost during a system crash. However, the use of atomic transaction is of particular interest when those read/write commands are executed in several different nodes of a DCS. If any of these commands do not complete for some reasons, the whole transaction will be aborted. Hence, consistency is maintained.

2.3.4.2.1. Two-phase commit

In order to acheive atomic transactions, the **two-phase commit** protocol was emerged. This protocol transforms a system from one consistent state to another and it works as follows:

A client process makes a transaction request to a server which is accompanied by certain hosts within a distributed system as depicted in the following diagram:



As soon as the server has received the request, the transaction will proceed in two stages:

(a) **Preparation phase**:

This phase involves both the server and its co-operating sites. On the server side, there is a process called the **commit co-ordinator** [27] which sends **request commit** messages to all participant sites. If any site indicates NO (or cannot be contacted), the co-ordinator aborts the whole transaction. If all vote YES, the co-ordinator records the information necessary to perform the transaction (usually on stable storage which will

be discussed in the next sub-section). Then it sets a commit bit, broadcasts a **commit** message to each participants and waits for their acknowledgements. Finally, it re-sets the commit bit.

On each participant sides, after the request commit message has arrived, the participant is asked to go into a state in which it can either redo or undo the actions allocated to it. If this fails, the participant will reply NO as the answer, otherwise it votes for YES.

(b) **Commit phase:**

At this phase, the transaction will run to completion regardless of crashes of the co-ordinator or any participants.

Several points are worth noting about this protocol:

(1) The decision to commit is centralized and is stored in one place.

(2) It is assumed that none of the participants will lie about their willingness to commit.

(3) This protocol cannot survive system crash without the aid of a recovery process. This process will be situated at the server site and is started to run after a crash but before any user activities. At this point, if the previous transaction has not yet committed as indicated by the information stored on the stable storage, the recovery process rolls it back (by broadcasting abort), re-broadcasts the commit message to the associated participants and re-starts the whole transaction. This recovery mechanism also applies to the situation where the commit co-ordinator crashes after setting the commit bit but before the end of the transaction. However, the two-phase commit protocol may still not work in the absence of crashes, due to problems such as communication links failures.

2.3.4.2.2. An atomic transaction example

As an example, Lampson [28] has described a way of achieving atomic transactions by building up a hierarchy of abstractions. In his model, certain assumptions were made. Real disks can crash during READ and WRITE operations in an unpredictable way. Even if a disc block is correctly written, there is still a possibility that it will be corrupted by a newly developed bad spot on the disc surface. His model also assumed that spontaneous block corruptions are sufficiently infrequent and the probability of two such events happening within some pre-determined time T is negligible. To deal with real disks, the system software must be able to find out whether or not a block is valid, eg. using a checksum.

Three layers of abstractions existed in Lampson's model. The first layer of abstraction, on top of the real disc, is the **careful disc** in which every CAREFUL_WRITE is read back immediately to verify that it is correct (perhaps using Cyclic Redundancy Code checks). If the CAREFUL_WRITE persistently fails, the system marks the block as **bad** and then intentionally crashes. Since CAREFUL_WRITEs are verified, CAREFUL_READ will always be **good**, unless a block has gone bad after being written and verified. The next layer of abstraction is **stable storage**. A stable storage block consists of an ordered pair of careful blocks, which are typically corresponding careful blocks on different drives, to minimize the chance of both being demaged by a hardware failure. The stable storage algorithm guarantees that at least one of the blocks is always valid. The STABLE_WRITE primitive first performs a CAREFUL_WRITE on one block of the pair, and then the other. If the first one fails, a crash is forced as described above and the second is left un-touched. After every crash and at least once every time period T, a special cleanup process is run to examine each stable block. If both blocks are **good** and **identical**, nothing needs to be done.

However, if one is **good** and one is **bad** (failure during a CAREFUL_WRITE), the bad one is replaced by the good one. If both are **good** but

different (crash between two CAREFUL_WRITEs), the second one is replaced by a copy of the first one. This algorithm allows individual disc blocks to be updated atomically and survives infrequent crashes.

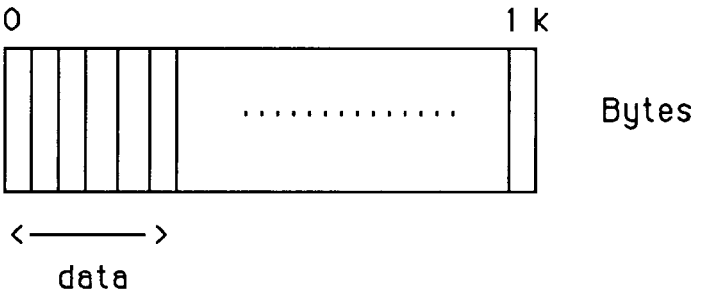
In Lampson's paper, he also pointed out that stable storage can be used to create **stable processors**. To make itself crashproof, a CPU must checkpoint itself on stable storage periodically. If it subsequently crashes, it can always restart itself from the last checkpoiont. Stable storage can also be used to create stable monitors in order to ensure that two concurrent processes never enter the same critical region at the same time, even if they are running on different machines. Given a way to implement crashproof processors (stable processors) and crashproof disks (stable storage), it is possible to implement multicomputer atomic transactions. Before updating any part of the data in place, a stable processor first writes an intentions list to stable storage, providing the new value for each datum to be changed. Then it sets a **commit flag** to indicate that the intentions list is complete. The commit flag is set by atomically updating a special block on stable storage. Finally, it begins to make all the changes called for in the intentions list. Crashes during this phase have no serious consequence because the intentions list is stored in stable storage. Furthermore, the actual making of the changes is idempotent, so repeated crashes and restarts during this phase are not harmful.

2.3.5. File system

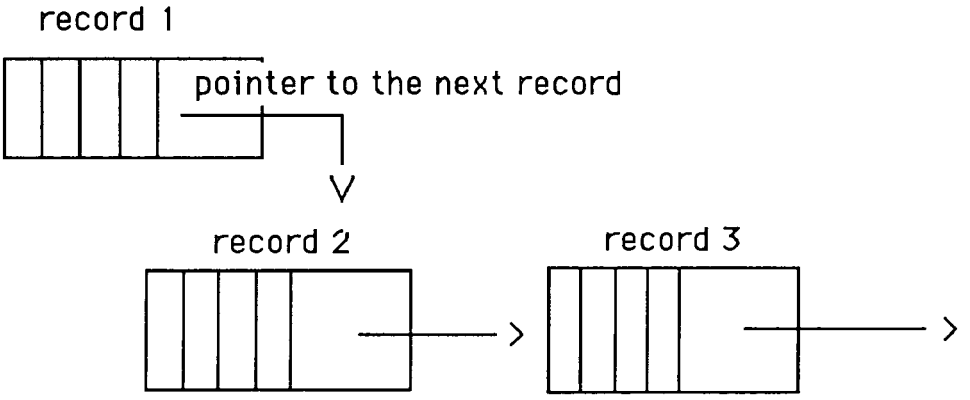
This is the most important service in any distributed systems. Typically, a distributed file system has three components: *filestores*, *file servers* and *files*. A filestore is a repository for data, providing a mnemonic naming scheme for files. A file is considered to be a sequence of user-defined arbitrary bytes which is stored by the filestore under a given name (or names), but not interpreted by the filestore. A file server is taken to mean a repository, where files can be stored (usually temporary) and which provides an index address for files contained in it. This address may contain authorization information. Associated with the file server

is another server called the **directory server** which will provide a mapping from user-arbitary mnemonics to the file server index. In addition, the directory server will also provide some kind of data protection mechanism, perhaps using the authorization facilities possessed by the file server. This protection issue will be discussed further in chapter three.

File services can be loosely classified into two types: **traditional** and **robust**. Traditional file service is offered by nearly all centralized systems (eg. the Unix file system). Files can be opened, read and rewritten. In particular, a program can open a file, seek to a particular position of the file or update blocks of data in the file. Thus, there is no structure imposed on files which are simply treated as row of bytes in the computer memory as shown in Figure 2.6(a). All the updates of the data are undertaken by a process, the file server, which will overwrite the relevant disc blocks if necessary. However, for concurrency control, it usually involves the *locking* of these files before updating them. On the other hand, robust file service is aimed at those applications (eg, databases) that require extremely high reliability by introducing a well-defined structure on files as in Figure 2.6(b). Therefore, users are prepared to pay a significant penalty in performance. Finally, robust file services offer atomic transactions (as described earlier) and similar features lacking in the traditional file services. The next sub-section will describe the design issues of these two kinds of file services.



(a).



(b).

Figure 2.6 (a) unstructured file construct in a traditional file service,

(b) structured file construct in a robust file service.

2.3.5.1. Traditional file services

Before going into the design details of a traditional file service, its components are discussed first which are the **disc service, flat file service** and **directory service.**

The disc service is concerned with reading and writing raw disc blocks without regard to how they are organized. A typical command to the disc service is to allocate and write a disc, and then return a *capability* or an address so that the block can be read later. The flat file service is concerned with providing its clients with an abstraction consists of files and each of them is a linear sequence of records, possibly one-byte records (as in Unix) or client defined records. The operations involved are reading and writing records, starting at some particular places of the file. The clients need not be concerned with how or where the data in the file is stored. The directory service provides a mechanism for naming and files protection, so that files can be accessed conveniently and safely. The directory service normally provides objects called **directories** which map ASCII names onto the internal identification system used by the file service.

One important design issue of traditional file service are how closely the above three components are integrated. At one extreme, one can have distinct disc, flat file and directory services running on different machines and only interacts via the official interprocess communication mechanism. This approach is most flexible because anyone needs a different kind of file service can use the standard disc server. It is also potentially the least efficient since it generates considerable inter-server traffic. At the other extreme, the three components are handled by a single program, typically running on a machine to which a disc is attached. With this model, any applications that need a slightly different file naming scheme is forced to start from scratch, making its own private disc file server. In this approach, there is an increase in run-time efficiency because the disc, file and directory services do not have to communicate over the network.

Another design issue is about **garbage collection.** If the directory and file services are integrated, it is a matter to ensure that whenever a file is created, it is entered into a directory. If the directory system forms a rooted tree, it is always possible to reach every file from the root directory. However, if the directory service and the flat file service are distinct, it may be possible to create files and directories that are not reachable from the root directory. This may be acceptable in some distributed systems but not in those where unconnected files may be regarded as garbage to be collected by the system. Conversely, another approach for garbage collection is to forget about rooted trees and permits the system to remove any file that has not been accessed for a pre-defined period of time. This approach is intended to deal with the situation where a client created a temporary file and then crashed before recording its existence anywhere. When the client is rebooted, it creates a new temporary file and the old one is lost forever unless a time-out mechanism is used.

2.3.5.2. Robust file services

Since robust file services normally include traditional file services as a subset, so all the issues discussed in the previous sub-section also apply to them. However, reliability is the additional key design issue in robust file services because the main task of these file services is to serve applications that demand a higher degree of reliability. One of the simplest method to achieve this extra degree of reliability is called **mirrored disks**: Every WRITE request is carried out in parallel on two disc drives. At every instant, the two disc drives are identical and either one can take over instantly for the other in the event of failure. A refinement of this approach is to have the file server offer **stable storage** and **atomic transactions**.

Moreover, all file services have to decide whether they are **virtual-circuit oriented** or **stateless**. In the former, the client must perform an OPEN operation on a file before reading it, at which time the file server fetches some information about the file (as i-node in Unix) into the

memory, and the client is given some kind of a connection identifier. This identifier is used in subsequent READ and WRITE commands. In the stateless approach, each READ request identifies the file and its position in full, so the server needs not keep the i-node in memory.

Both virtual circuit and stateless file servers can be used with the ISO OSI and remote procedure call models. When *virtual circuits* are used for communication, having the file server maintain open files is natural. However, each request message can also be self-contained so that the file server need not keep the file open throughout the communication session. Similarly, RPC fits well with a stateless file server, but it can also be used with a file server that maintains open files. In the latter, the client performs a RPC to the file server to OPEN the file and then gets back a file identifier of some kind. Subsequent RPCs can carry out the READ and WRITE operations using this file identifier.

Nevertheless, the difference between virtual circuit and stateless approaches becomes clear when one considers the effects of a server crash on active clients. If a virtual circuit server crashes and is then quickly rebooted, it will always lose its internal tables. When the next request comes in to read the current block from the file identifier, say 28, it will have no way of knowing what to do. The client will then receive an error message which will lead to client's process aborting. In the stateless model, each request is completely self-contained (file name, file position etc), so a newly rebooted server will have no trouble carrying it out. Obviously, the only price to pay for this robustness is a slightly longer messages.

It can be realized that the file server has been involved in every stages of a file transaction (update) which emphasizes the point that the performance of a DCS is strongly related to the type of file server used. Since file servers are so important to DCSs, the next entire chapter will be devoted to this issue.

2.3.6. Other services

Other services provided by DCSs concern heavily with hardware implementation issues, so they will not be discussed in details here. These services include:

(a) Print service,
(b) Process creation service,
(c) Terminal service,
(d) Time service,
(e) Boot service,
(f) Gateway service.

2.4. Conclusions

In this chapter, a literature survey of distributed computing systems (DCSs) has been presented. The discussed topics were: the motivations and characteristics of DCSs, the distinction between DCSs and network computing systems, examples of some existing DCS models and the major implementation issues of DCSs are also given. Finally, this section summarizes this chapter by pointing out the tradeoffs of DCSs.

Advantages :

(a) Distributed systems have both the price and performance advantages over traditional systems such as centralized and network systems.

(b) The relative simplicity of the software which dedicated a function to each processor.

(c) Easy to add computing power . If users need ten percent more computing power, ten percent more processors are added.

(d) Reliability and availability. A few parts of the system can be down without disturbing users using the other parts.

Disadvantages :

(a) If a user wants to load a file (which may be in the user's machine), a complicated communication protocol may be needed. The request may have to be passed through several machines before the file is located. So, it is easy for communication protocol overhead to reduce the efficiency of the whole system. For instance, one of the machines may require the full computing power of some processors just to run the protocols, leaving nothing to do the work. Therefore, a high degree of fault tolerance is often required. The only sensible solution to this problem is to construct an efficient communication protocol.

(b) Lack of global state and up-to-date information due to the frequent access of files over machines. This may result in some inconvenient situations such as it is hard to schedule the processors optimally because one cannot be sured about how many processors are available at any moment.

Chapter Three -- Distributed file servers

A file server provides remote storage of data to single-user machines, typically *workstations*, connected to it via a communication network; it facilities data sharing among workstations and supports inexpensive workstations that have limited, expensive or no secondary storage. There are processes called *clients* which can invoke or control the remote file *services*using a set of operations that form the client *interface*.

For brevity, hereafter, the term **file server** will be referred as a distributed file server unless stated otherwise. Also, for simplicity, systems with multiple file servers, such as the LOCUS distributed system [6], will not be considered in details in this thesis because they were intended to support replicated copies of data.

In the previous chapter, two types of file services have been described: the **traditional** and **robust** file services. The former differs from the latter in that it does not impose any specifications on the nature of files, i.e. no structure. The main theme of this chapter is to describe the functions of the corresponding file servers of these services in distributed computing systems. As a preparatory stage, the basic design issues of a distributed file server are discussed. Then examples of un-structured and structured file servers will be presented to illustrate how they fit into the general structure of distributed systems.

3.1. Requirements of distributed file servers

For any file servers, four kinds of files may be supported. **Ordinary files** consist of a sequence of disc blocks that may be updated in place and that may be destroyed by disc or server crashes. **Recoverable files** have the property that groups of WRITE command(s) can be bracketed by a construct such as BEGIN TRANSACTION and END TRANSACTION, and that a crash or abort midway leaves the file in its original state. **Robust files** are files written on stable storage and contain sufficient redundancy to survive disc crashes. Finally, **the multi-version files** consist of a sequence of versions, each of them is being immutable. Changes are made to a file by creating a new version. Different file servers support various

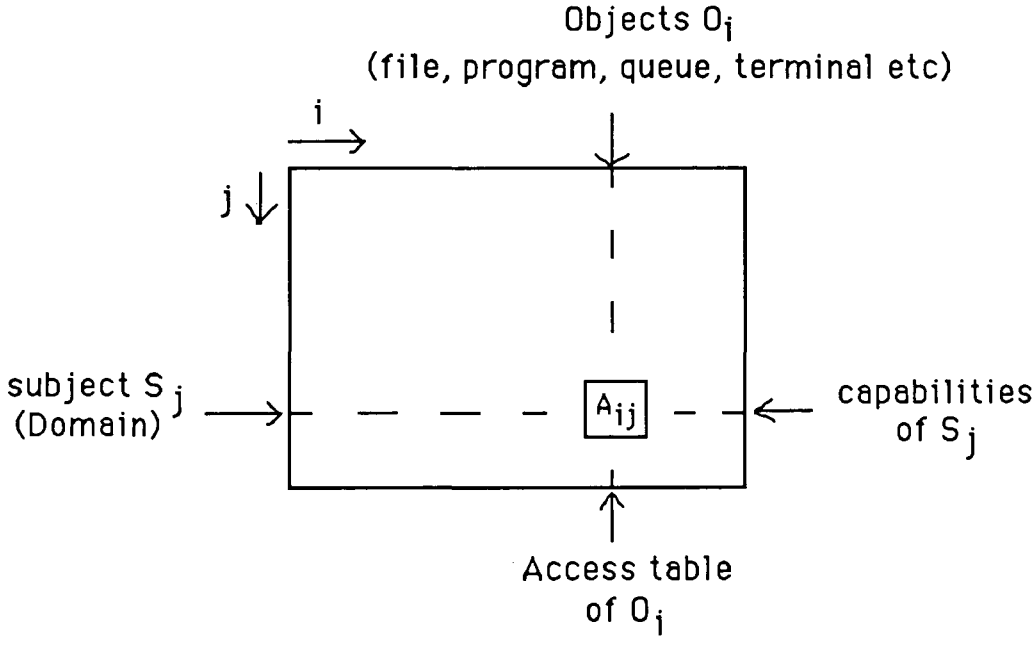
combinations of these four types of files. In addition to these files, robust file servers also need some mechanisms to handle concurrent updates to a file or group of files. Many of these servers allow users to lock a file to prevent conflicting WRITEs. Although *locking* introduces the problem of deadlocks, it can be dealt with by using techniques like two-phase locking in a DCS [29].

3.1.1. Design issues

There are several important issues concerned with the design of a file server.

Access control:

The first issue faced by all file servers is who is allowed to use which resource. In centralized systems, this problem is solved based on the concept of **access matrix.** By postulating a set of objects O_i to be protected and a set of subjects S_j wish to access those objects, a matrix A is set up such that A_{ij} contains the access rights of subject j to object i as shown in the following diagram:



An entry A_{ij} may contain rights such as read, write, execute, change rights etc. However, it is impractical to store the matrix in a computer because of its sparseness, so two techniques are adopted to represent it compactly yet allowing fast lookup.

In the first technique, the column of the matrix under O_i, the so-called **access control list** of that object, is stored in association with the object and checked when access is requested. Thus, each subject S_j of the list must have a unique identifier. For instance, the Unix RWX bits are a simple form of access control list which divides all users into three categories: owner, group and others.

An alternative way to store the access matrix is to take a row correponding to one subject and store these data with the subject. Each element of the matrix is then called a **capability**. When a subject requires access to an object, it presents a capability, like a ticket, for that object which is then checked for validity. The checking will be simple because no extra data is needed.

Unfortunately, with a distributed system using remote file servers, both of these approaches have problems. With access control lists, the file server has to verify that the user is in fact who he or she claims to be. With capabilities, one has to prevent the users from making them up. One way to solve the access control list problem is to insist the client first sets up an authenticated *virtual circuit* with the file server [29]. The authentication may involve a trusted third party. With the capability problem, one possible solution is to encrypt all the capabilities [30].

Performance:

This is another important issue in using a remote file server (particularly for diskless workstations). Reading a block from a local disc requires a disc access and a small amount of CPU processing. Reading from a remote server has the additional overhead of accessing across the network. This overhead has two components: the actual time to move the bits over the wire and the CPU time that the file server must spend running the protocol software. Cheriton and Zwaenepoel

[31] have done some experiments to measure the network overhead in the V system which has a very fast file server. With an 8-megahertz 68000 processor and a 10-megabytes per second Ethernet, they observe that reading a 512 byte block from the local machine takes 1.3 milliseconds and from a remote machine takes 5.7 milliseconds, assuming that the block is in the memory and no disc access is needed. They have also observed that loading a 64K program from a remote file server takes 255 milliseconds versus 60 milliseconds locally. From the above results, one can roughly conclude that access to a remote file server is four times as expensive as to a local one. One way to improve the performance of a distributed system is to reduce the network overhead by having both the clients and servers maintain **caches** of disc blocks or possibly the whole files.

Reliability:

For a distributed system which has one file server on a single machine, reliability is only a matter of disc management and the simplest strategy is to use good quality disks and make periodic tape back-ups. However, for a distributed system in which multiple servers are working together, it becomes possible to enhance reliability by replicating some or all files over these multiple servers. Reading will be easier because the work load is split over several servers but writing is much harder since the multiple copies must be updated simultaneously. One solution to this problem is to distribute the data or files but keeping some of the control information centralized so that one site can be chosen to proceed with the update first and leaving the other sites to be brought up-to-date later. Nevertheless, this thesis will only focus on the single file server situation.

3.2. Un-structured file servers

The characteristic of these servers is that they do not impose any kind of formats on the files content other than row-of-byte. Two un-structured file servers are considered representative. Each of them used a different approach to provide the traditional file services. The selection criteria were as follows. First,

they have been actually implemented and available. Second, many research studies have been carried out with them.

3.2.1. Cambridge File Server

The Cambridge File Server (CFS) was written using the BCPL language and runs on a Computer Automation LSI4/30 minicomputer with 64K, 16-bit words of memory and an execution rate of approximately 1Mips (millions of instructions per second). The CFS program occupies about 50K words, leaving 14K for data and disc buffering. The Cambridge Ring [33] serves as the communication medium between the CFS and its clients. It has a raw *bandwidth* of 10M bits/second and a maximum point-to-point data bandwidth of about 1M bits/second. The CFS uses disc units with average *access time* of 35 msecs and *transfer rate* of about 1M bytes/second and have an 80M bytes disc capacity.

3.2.1.1. Design issues of CFS

CFS was intended to replace the backing store management of the CAP computer [34]. This file server provides a service to a range of *client* machines which do not necessarily wish to operate identical *filestores*, and may not have their own secondary store [35]. The design objectives of the *file server* include high speed transfer to random access word-addressed files and a high degree of crash resistance. For the purpose of simplicity, the CFS concentrates its full bandwidth on one particular client for the duration of a file read/write before serving another client. During the period of a single transaction, the CFS uses the multiple-readers/single-writer interlocks mechanism to lock the entire file. CFS also allows only one file to be updated atomically in a single transaction. However, CFS supports concurrent random access to files.

In CFS, objects stored are either files or indices, each identified by a unique identifier (UID) which is 64 bits long including 32 **random** bits. A file is a sequence of 16-bit words on which random access read and write operations are available to clients. An index is a list of UIDs, on which PRESERVE, RETRIEVE and DELETE operations are available. Finally, files and indices may be created at will and interconnected in an arbitrarily complex graph. Note also that this file server does not keep information between operations apart from the files and indices.

Naming in CFS is in terms of UIDs; a client can set up a general directed storage graph where nodes are files and indices. Thus, there is a single global context in which file UIDs are unique, but it is implemented by stepping through indices whose UIDs are known. The UID is also used as a capability for protecting access to files and indices. A client must remember the UID of the node given as its root; subsequently the client may construct directed graphs [36] and exchange UIDs with others knowing only its root UID. No delete operations are provided in CFS. A special system **root** index is designated and a periodic asynchronous garbage collector is invoked [37] to remove all objects not accessible from the root. If a client wants to use its own naming scheme (eg. a Unix-like directory scheme) it may do so in terms of the file server interface. The file server itself has no knowledge of this higher level structure. Thus, Unix directories would be implemented using files.

N.B. A Unix-directory scheme uses the technique of **direct access** which is to use the name of a file to obtain an address and that address called an **inode** is used for accessing the contents of the file.

CFS using the following mechanism for communication: Cambridge Ring is the communication medium between the CFS and its clients. It uses a **mini-packet** containing local source and destination addresses and two data bytes as the basic transmission unit. Because of the small data capacity of mini-packet, CFS converses with its clients using the **basic-block protocol** (BBP) which forms messages from sequences of mini-packets. A port number is included in the destination machine to identify a process. The two well-known

problems in designing protocols are **error control** and **flow control**. That is, the detection of errors in transmitted data and means for matching the speed of transmission to the speed of reception respectively.

In order to tackle the above two problems, CFS exploits the hardware features of Cambridge Ring. For error control, the CFS only reports reception errors to a transmitting client after a complete write operation, which is implemented as a sequence of basic blocks flowing from the client to the server, because of the reliability of the ring (one bit error in $5*10^{11}$) [33]. For flow control, the CFS uses the hardware response bits of a mini-packet that are set when it returns to its source. If there is not enough buffer space, the CFS rejects one of its early mini-packets to discard the incoming basic block. As the transmitter detects this rejected response, it will restart the basic block transmission after a suitable interval [33]. This method of flow control is not supposed to be used heavily. It is a policy that a file transfer occurs at the ring speed, a client only reads as much data as it can be stored immediately in memory and the server can accept indefinite amount of data at full ring speed.

Operations provided by the CFS are designed to be repeatable. File reads and writes specify the starting offsets in the file absolutely. Thus, a client can retry a command without a reply from the server and the sever can discard its state as soon as it has sent a reply and need not wait for a client's acknowledgement of the reply. Therefore, almost all CFS operations consist of a single request followed by a single reply; reading or writing a file additionally involves an exchange of basic blocks containing pure data. However, the CFS only assumes purely local area network.

3.2.1.2. Crash recovery

Failure part way through a transaction can leave the storage in an inconsistent state. In order to prevent the inconsistency, the client must define a new file to be **normal** or **special**. In the former case, no provision is made for

recovery; update may be in place. Special files in CFS are updated using an **intentions list** mechanism [38] which permits a single operation comprised of several block operations to be carried out atomically. The block allocation tables for special files contain the following state information:

(i) **allocated/de-allocated**
(ii) **intending to allocate/de-allocate**

The intentions list algorithm is:

```
FOR all blocks to be updated DO
  BEGIN
    choose a de-allocated block and mark it intending to allocate;
    change old block to intending to de-allocate;
    write to the new block
  END;

set the commit bit; {on stable storage}

FOR all relevant blocks DO
  BEGIN
    change all intending to allocate blocks to allocated;
    change all intending to de-allocate blocks to de-allocated
  END;

reset commit bit;
```

Up to the setting of the commit bit, all changes can be done. After setting the commit bit, the changes will be done eventually. On a crash, the commit bit indicates whether to go forward or backward. In the forward case, all intending blocks must be definitely allocated or de-allocated. Clearly, a crash part way through setting the commit bit or the intention bits will invalidate the above algorithm.

3.2.1.3. Summary of the CFS

Success of CFS :

In general, CFS is a very good un-structured file server. CFS provides only a slow backing store service due to its basic principle of keeping the design simple such as the simple file transfer protocol described earlier. As a result, CFS is used as a file server interface for simple microprocessors. Also, the distinction between normal and special files have prevented clients from un-necessary atomic transactions. This leads to the intention that the client operating system uses special files to contain file directory information and normal files for most other purposes.

Another success of CFS is the interface presented by the server to the clients in form of a directed graph with capability access to files and implicit storage reclamation. Capability access control is simple to implement and avoids the question of client identity which may be difficult to determine reliability in local networks. The embedding of disc addresses directly in these capabilities is to be recommended as a means of speeding up the access to files. Also, the graph structure of indices maintained by the CFS provides a good abstraction as a building block for operating systems: the operating systems of CFS's clients can construct their file directories from an index and an associated file which contains string names and access rights.

The automatic garbage collection of CFS is another success [34]. A **reference count** mechanism deals with most deletions; an object will be deleted whenever the count references to it falls to zero. However, CFS does not ensure the accuracy of these counts and certain types of server crashes may leave these counts high.

Disadvantages of CFS :

One of the most obvious shortcomings of CFS is that atomic transactions are restricted to updating a single file or index only. For instance, since file directories are implemented as an index with an associated file, it is impossible to update both structures in a single transaction. Besides, due to the assumptions made on the file transfer protocol, it is difficult to extend CFS to wide area networks, so CFS is only a local area network file server.

3.2.2. Newcastle Connection

3.2.2.1. Unix United system

Before going into details of Newcastle Connection (NC), the principles behind Unix United is described. Unix United [39] which is composed of several Unix systems aims to be functionally equivalent to a single conventional Unix system. All the common Unix features including naming, protecting and accessing devices, files and directories are provided. Typical examples of a single Unix and a Unix United *namespaces* are shown in Figure 3.1(a) and Figure 3.1(b) respectively.

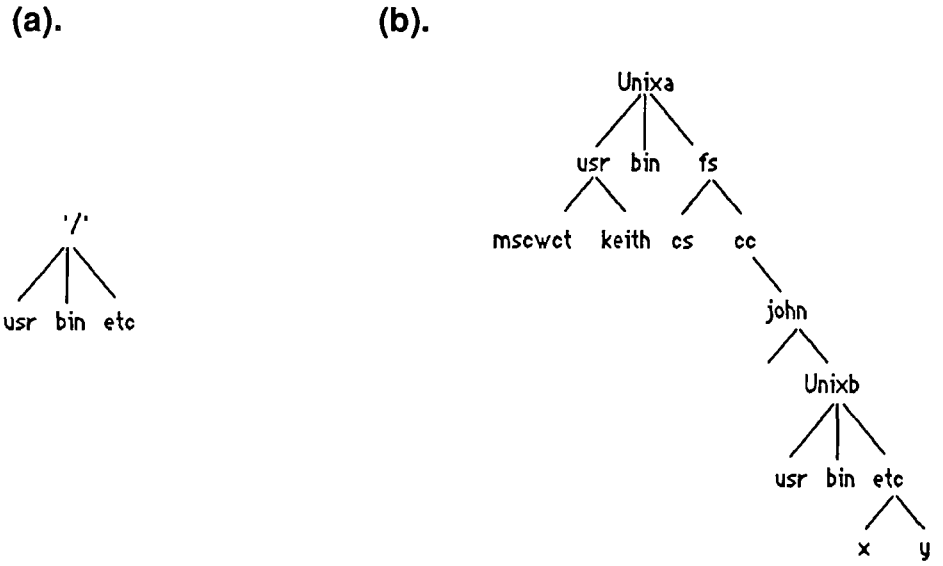
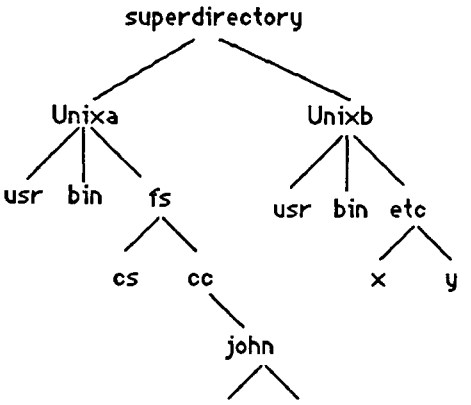


Figure 3.1 (a) a single Unix system , (b) a Unix-United system.

In the single Unix namespace, files are named relative to either the root directory "/" or the current directory. For Unix United, two Unix systems can be simply joined together by attaching one of them as a leaf, where appropriate, in the hierarchy as shown in Figure 3.1(b). More Unix systems can be joined in a similar fashion to form even larger systems. One major advantage of this scheme is to eliminate the problem of file names clashes, provided that the files are in different sub-systems. Subsequently, files can be accessed across the united system using the appropriate references. Consider the Unix United system in Figure 3.1(b), if the current directory is "mscwct" of the sub-system Unix a, the file y of sub-system Unix b can be accessed using the pathname /fs/cc/john/Unixb/etc/y. However, there is a more efficient way of joining the various Unix systems with the creation of a **virtual superdirectory** above the roots of all connected systems as depicted in the following diagram:



In this case, the same file y will be referenced by a shorter pathname /../Unixb/etc/y where "/.." referring to the superdirectory and the rest is the same as the usual Unix naming scheme. Under this scheme, all the files in the united system are named relative to the superdirectory only. This strategy possesses two interesting features. First, each sub-systems can manage its own hierarchy structure independent of the others. Second, each file of the united system can be identified by a unique pathname, provided that the hierarchy of its resident system has not been changed.

In order to implement a Unix United system, an additional layer of software-the Newcastle Connection [39] in each of the component Unix systems is required. It is situated between the Unix *kernel* and the rest of the operating system as shown in Figure 3.2. NC is implemented as C library routines and its functions will be discussed in the next sub-section.

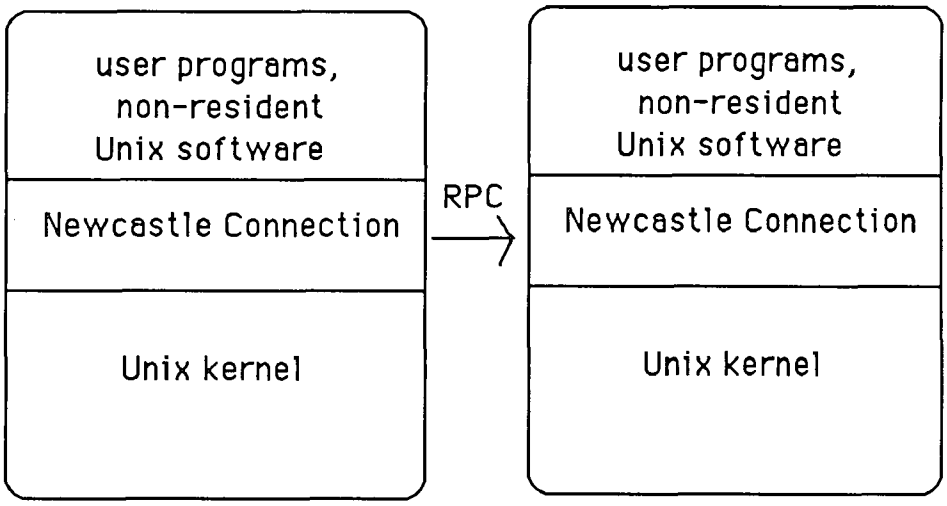


Figure 3.2 Newcastle Connection software layer

3.2.2.2. Goals of Newcastle Connection

The function of the Newcastle Connection layer is to intercept system calls. To perform the interception, NC classified system calls into three categories: those which take pathname arguments, those which take arguments such as file descriptors which have been derived indirectly from a pathname by a previous system call and finally those which take arguments such as user IDs or process IDs.

The first two categories mentioned above can be handled quite comfortably because pathnames may be examined to see whether they contain a reference to a remote system or start from a remote context whilst file descriptors, which have been created by a previous system call, may be looked up in a table maintained by the NC. However, the third category is problematical because there is no obvious structure in a system identifier that can be extended to a distributed system. The best solution may be to assume that such identifiers refer to the system where the root directory "/" is located. Once the system call has been analysed, local

calls and remote calls will be passed to the appropriate machine. This process is invisible to users and the operating system programs. Inter-machine communication is performed via the technique of Remote Procedure Call (RPC), as described in chapter two. On the remote machine, a spawner process, which runs continuously, initially receives an open file request and spawns a file server process as a result. Subsequently, the originator communicates directly with the file server using the name returned by the spawner. When the file is opened, NC makes an entry in a per-process table indicating whether the file descriptor (which is often an integer used to refer to the file between open and close calls) refers to a local or remote file. This table also holds the corresponding remote node addresses so that remote access can be routed immediately.

3.2.2.3. Remarks on Newcastle Connection

The major advantage of the Newcastle Connection (NC) design is that processes can access remote files transparently via the appropriate pathnames; no changes are required for the Unix kernel because file access is managed by the NC software layer (which is implemented in terms of C routines).

However, there are still several disadvantages with this design. System performance may be degraded. Because of the larger C library, each process takes up more memory even though it makes no remote references; the library duplicates some of the kernel functions and takes up more spaces. Also, larger processes take longer to start up and may cause greater contention for memory, inducing a higher degree of page swapping on the system. Local requests may execute more slowly as they take longer to get into the kernel, and remote requests may also be slow since they have to do more processing at the user level to send requests across the network. The extra user level processing provides more opportunities for *context swapping* and *paging*. Also, programs must be recompiled with new libraries to access remote files as old programs and vendor-supplied modules will not work any more.

Finally, there is apparently no published document on the implementation of the NC *protocol* which means no standard has ever been imposed on the message exchange format. Therefore, a 16-bit machine cannot communicate with a 32-bit machine. However, this is a political issue rather than a technical one because the task of confining messages onto the correct formats for exchange is totally optional; it can be done either on the sending site or on the receiving end. So, it should be realized that NC is just used as a **naming system** which does not care about the hardware architecture of the individual sub-systems.

3.3. Structured file servers

For applications that demanding a higher level of reliability and consistency, robust file services are required which, in turns, leads to the evolution of a modern data storage technique namely, the **database approach.** In this section, the following issues of database will be addressed: the evolution of databases, the fundamental concept of a database, the advantages of the database approach over the conventional data file approach and finally a fast-growing database system will be given as an example.

3.3.1. History of databases

A **database** may be regarded as the most modern technique of data storage, which started with the invention of punched cards by Dr. Herman Hollerith of the U.S. Bureau of Census in 1880s [40]. This introduced the era of mechanized card files which had been used as an information storage medium for the next 60 years.

In 1946, the first electronic computer, ENIAC, was in operation. It was designed by Professor Eckert and Mauchley of the University of Pennsylvania for the U.S. Defence Department mainly to calculate trajectories and firing tables. In

these early days, computers were mostly used for scientific computations where the facilities for the storage of data did not feature as an important requirement - the speed of arithmetic calculation was all that was interested. But when the use of computers was gradually extended to *data processing*, the limitations of card files began to be felt. One of the organization that became seriously concerned by such limitations was the U.S. Bureau of Census. Faced with the approaching 1950 Census, the Bureau became particularly anxious to have a faster storage device. In 1951, a new computer, later to be called Univac-1, designed again by Eckert and Mauchley, was delivered to the Bureau to cope with the 1950 Census data. It had a unique device called the **magnetic tape system**, which could read a magnetic tape both forwards and backwards at high speed. Thus, the problem of Census had lead to two major inventions in 70 years - the punched card and magnetic tape files.

The impact of magnetic tape on data storage was overwhelming because in those days, card files processing often make people end-up with the frustration of finding mispunched, slightly off-punched and screwed up cards scattered in large card files produced by a computer. The fear of accidentally dropping a box of cards and getting them out of sequence was also considerable. With the advent of the magnetic tape system, all these nightmares were over because it was a medium that was light, reliable and neat. The storage capacity and speed of magnetic tape was phenomenal compared with a card file. However, the use of magnetic tape did not alter the processing mode substantially as the file organization was still sequential although it allowed the representation of variable length records in a more convenient form. All the terminologies of card file systems such as files, records and fields were carried over to the magnetic tape system.

The data processing systems used in 1950s were mostly simple payroll sub-systems designed in isolation, that is, independent of other related sub-systems. A typical sub-system would consist of a large number of small programs with many files each containing fragmented information. The situation changed when subsequent attempt to computerize more complex systems brought in a new breed of experts - the **system analysts**. They took a global view and introduced the concepts of integrated files to be shared by a number of programs in more than one sub-system. These shared files were relatively large and the

problem of writing long data descriptions in each program for each file was resolved by COBOL's COPY verb, which allowed a program to copy a pre-written general data description of a file. From then on, the evolution of databases became the history of the progress from less to more integrated storage of data.

The introduction of magnetic disc in the mid 1960s gave a further boost to the integration. To access a record on a magnetic tape, it is necessary to scan all the intervening records sequentially, but on a disc it is possible to access a record directly by passing the other records. This would gain an overall retrieval speed of about 2 to 4 orders of magnitude over magnetic tapes. Disc storage thus provided the much needed hardware support for the large integrated files.

Also, by 1960s, the concept of **Management Information System** (MIS) gained attention. The basic approach was to run the programs of the MIS package on the output files of all the relevant sub-systems. However, it was soon found that for a large organization the number of input files to the MIS package was excessively high, with the attendant problems of extensive sorting and collating. Moreover, the failure of one sub-system could easily wreck the whole operation. Data duplication in the files leading to update inconsistencies posed yet another problem. Thus, these MIS packages turned out to be unreliable, cumbersome and generally unsatisfactory. This highlighted the need for still greater integration and led many organizations to opt for development in this direction.

One of the outstanding products in that time was the **Integrated Data Store** (IDS) introduced by General Electric (now owned by Honeywell) in 1965. As the name suggested, IDS was used to create large integrated files that can be shared by a number of applications. It was a forerunner of the modern **database system** and can support a number of data structures. Its pioneer Charles W. Bachman, subsequently played a very active role in the development of *Codasyl* database proposal, which incorporated many of the features of IDS. IDS was soon followed by other MIS packages based on integrated files for major systems. Many organizations invested large amount of money in them only to discover that their MIS packages were not as effective as they would like. The problem was the lack of co-ordination between the files of the major systems. It was realized later that what was needed

was a **database**, containing a generalized integrated collection of data for all the systems of the organization, serving all the application programs. It was also recognized that such a database should be both program-independent and language-independent if it was to serve all applications; in particular a change in the data should not require a change in the application program. If a database is to respond efficiently to the conflicting needs of all the application programs, then it must support a variety of data structures from simple to complex, providing multiple access paths. This concept of a database crystallized only in the early 1970s. although the term **database** or **data bank** had been used loosely since the mid 1960s to refer to almost any integrated files.

A number of database systems have appeared on the market in the past few years which gave variable performance and hardly any compatibility. Codasyl became interested in databases in the late 1960s and set up a task group to provide a common framework for all database designs. Since the publication of its draft specifications, there has been a noticeable movement away from diversity, converging on the Codasyl model - the **network data model**. Nowadays, all major computer manufacturers except IBM are committed to implement the Codasyl proposal. In 1972, the ANSI (American National Standards Institutes) has set up a working group to standardize databases; its proposal were likely to lead to a standard model based on the Codasyl specifications.

Parallel to these developments, the IBM research laboratories produced two new models for future database designs, in addition to the **hierarchical data model** on which their present database system, the Information Management System (IMS), is based. One of these is the **relational database model** and the other is the **data independence accessing model** (DIAM). The former is the most popular one.

Conclusively, database systems are designed to manage large bodies of information. The management of data involves both the definition of structures for the storage of information and the provision of mechanisms for the manipulation of information. In addition, the database system must provide for the safety of information stored in the database, despite system crashes or attempts at

unauthorized access. If data is to be shared among several users, a system must avoid the possibility of anomalous results. Finally, the design of a database system must include consideration of the interface between the database system and the operating system because in most cases, the operating system can only provide the most basic services and then the database system has to build on that base.

3.3.2. The database approach

From the history of databases, it can be realized that the database concept was rooted in an attitude of sharing common data resources, releasing control of these data resources to a common responsible authority and co-operating in the maintenance of those shared data resources.

Recently, several definitions have been proposed for the term database as in [40] and [41]. However, they all generally agree on the fact that a database has two properties: it is integrated and it is shared. By integrated, they mean that previously distinct data files have been logically organized to eliminate (or reduce) redundancy and to facilitate data access. By shared, they mean that all authorized users in an organization have access to the same data for use in a variety of activities.

The major components of a database environment have been identified by Clark [42] as shown in Figure 3.3:

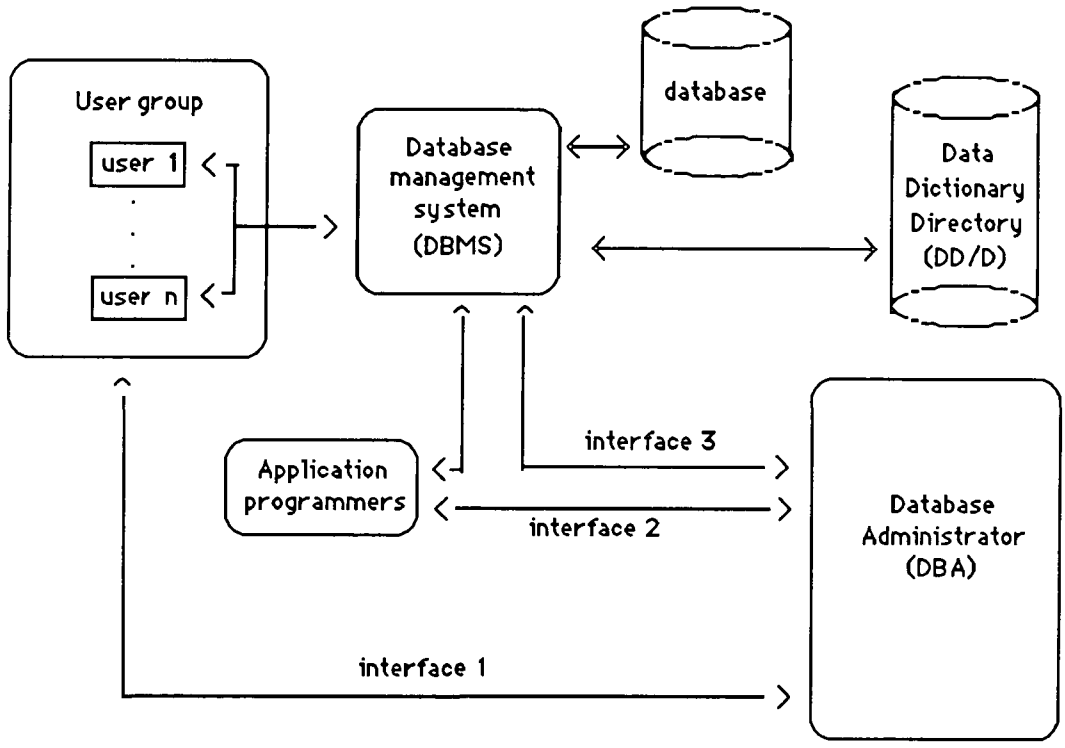


Figure 3.3 A database environment

With the help of the above diagram, it is easier to understand the database approach. The functions of each of the six components are summarized briefly as follows:

(a) **User group** :

It consists of all requesters of data. There are three basic categories of user requests: **read only**, **add/delete** and **modify**. All the user requests for data are made through the database management system (DBMS).

(b) **DBMS** :

It is a software system that receives and satisfies all requests for data. Normally, the DBMS provides concurrent access to multiple database users. Also, the DBMS must be able to recover or restore a damaged database from

backup copies.

(c) **Database :**

It is the physical repository (eg. magnetic tape) where all user data will be kept.

(d) **Data dictionary/directory (DD/D) :**

It is a place where all the definitions of the data used by the organization are stored. Therefore, it is the key tool in managing data resource, for instance, data item names, lengths etc.

(e) **Application programmers :**

People who write code to process user requests for data.

(f) **Database Administrator (DBA) :**

A manager who is responsible for the cost-effective definition of an organization's data via the three interfaces as indicated in Figure 3.3. In interface 1, the DBA defines user rights and responsibilities in retrieving data and mediates any conflicts that may arise. In interface 2, the DBA controls all data definitions and establishes standards for all application programs that access the database. The DBA also trains users and programmers to use the DBMS. Finally, in interface 3, DBA monitors operational performance of the DBMS and initiates changes that may be necessary to improve *response time* or other operational characteristics.

The following sub-sections will describe the various data models for database designs, followed by two most commonly used database access methods.

3.3.2.1. Data models

A **model** is a representation of real world objects, events and their *associations*. It is an abstraction from reality and often is simplified for ease of understanding and manipulation.

A **data model** is a collection of conceptual tools for describing data, data relationships, data semantics and data constraints within an organization. The main task of a data model is to represent data and to be understandable. If a data model accurately and completely represents the required data and it is understandable (and perhaps easy to use), it can be used for special purpose applications such as database design.s There are three most widely accepted data models for database systems.

3.3.2.1.1. Hierarchical model

A hierarchical database consists of a collection of **records** of the same type and they are connected with each other through **links**. A link is an association between exactly two records. Each record of the database is a collection of fields and each field contains only one data value. Furthermore, records are organized as collections of **trees**. Therefore, a **tree-structure** diagram is the best way to describe a hierarchical database as shown in Figure 3.4.

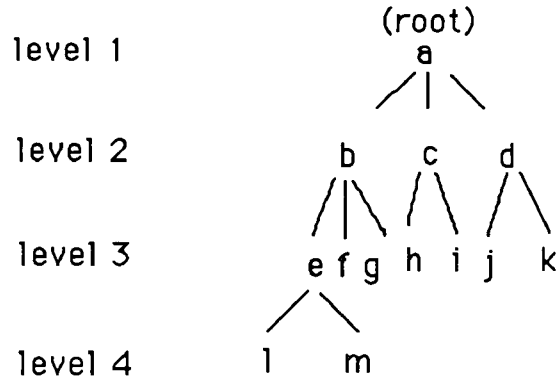


Figure 3.4 A hierarchy structure

This hierarchy is made up of elements, called **nodes**, which corresponds to record types. The arrows in the diagram correspond to links. At the uppermost level, there is only one node called the **root**. Except for the root, every other nodes has one node related to it at a higher level which is terned as its **parent**. Each element can have only one parent but many lower level elements known as the **children**. So, the tree-structure diagram specifies the overall logical structure of the database. However, when a given record is associated with several other records that belong to its parent's level, the single-parent rule of the hierarchical data model will force redundant and excessive data and structure.

Despite of redundancy, the hierarchical model is still well-established in practice. For example, IMS [43] is a DBMS running on the IBM mainframes that supports this hierarchical view of data. IMS uses the retrieval sub-language called DL/1 as the navigational method of extracting data in an IMS database.

3.3.2.1.2. Network model

To eliminate the redundancy problem in the hierarchical model, the **network data model** was emerged. This model originated from the proposal of the Database Task Group (DBTG) of the Conference on Database Languages (Codasyl). Codasyl is the same group that responsible for the standardization of the language COBOL.

The basic building block in a network database is the structure referred as a **set**, which consists of a collection of information called the **members** of the set. These member collections are attached to the owner via a linkage system similar to that of a linked list, with the exception that the last pointer of a set structure always points back to the owner of the set where the **head pointer** is stored. Thus, beginning at a set's owner, one can easily find out all the members of the set.

The result of the above set approach to data storage is a database which contains a network of paths that can be followed by the user to find the required information at any instant. Such a model closely reflects the way data is stored in bulk storage. Figure 3.5 below gives an example of such model.

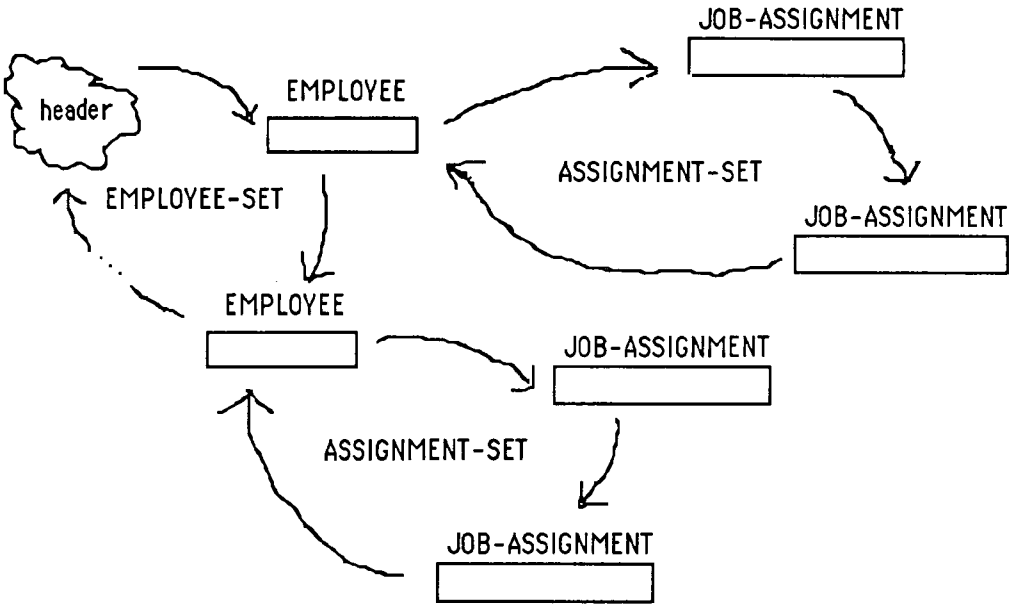


Figure 3.5 A network database example

In Figure 3.5, the head pointer (header) serves as an entry point to the database. The rectangular boxes represent the group occurrences of a set, with the name of the set nearby such as the set EMPLOYEE-SET. By interrogating the database, the job history of each employee can be examined.

IDMS [44] was a network database system developed by Cullinane Database System Incorporation. IDMS adhered closely the DBTG model. It includes a detailed data manipulation language allowing the database designer has a higher degree of control over the physical organization of the database. Tha data manipulation language included features identical (or very nearly) to those of the DBTG model [45]. However, additional features are included which can enable programmers to write more efficient queries. For example, the **obtain** command, which combines the **find** and **get** commands of the DBTG model into one request. An optional **where** clause may be attached to an **obtain** command to find and get the next record satisfying the **where** clause. This feature relieves the programmer for writing an explicit test of a record located via the **find** command.

Unfortunately, the network model has been criticized of being complex and difficult to use. Network diagrams, unless thoughtfully arranged, frequently look very messy. Moreover, this model has been typically implemented in ways to be consistent with the so-called **record-one-at-a-time** processing languages such as COBOL, which make database processing become harder. As a result, another model, the relational data model , was developed.

3.3.2.1.3. The relational model

It is the most popular model owing to the simplicity of its structure. This relational approach to databases is based on the mathematical theory of **relations**. The approach thus uses **relational algebra** and **relational calculus** terms to describe the database and operations on the data. In the former, one or two relations are manipulated as operands to produce a new relation as the result whilst the latter manipulates relations implicitly by specifying conditions that can involve data items from several relations. Typical relational algebra operators are SELECT, PROJECT and JOIN. Relational calculus usually combines the three algebra operators into one single operator called RETRIEVE, and a WHERE clause to specify a condition. Although a relational calculus statement can become very complicated, it can be stated only in one language command, the equivalent of many relational algebra commands.

Data in a relational database is stored in a tabular form known as a **relation**. Each distinct **data item** is called an **attribute** value. A **tuple** is a collection of values that composes of one row of a relation whereas a **domain** is a set of possible values for an attribute. Figure 3.6 illustrates a relation.

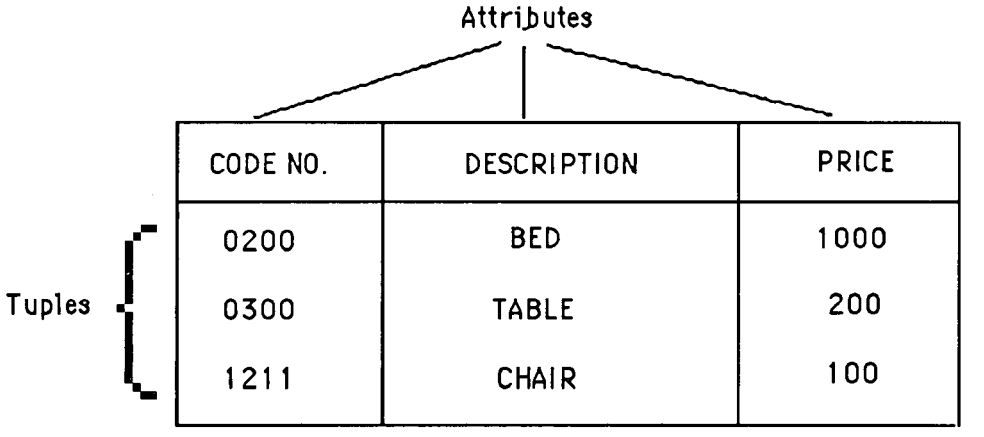


Figure 3.6 A product relation

Codd [46] has contributed significant papers on the relational model and he has also proposed the concept of **normalization**. Normalization was defined by Codd as a step by step reversible process of replacing a given collection of relations by successive collections in which the relations have a progressively simpler and more regular structure. The aim is to find relationships between data that are free from undesirable interactions, and then simplifying the process of maintenance and data retrieval.

Nevertheless, the prime objective of normalization is the production of an ideal data format known as the **third normal form** or 3NF for short. Figure 3.7 shows the stages to 3NF:

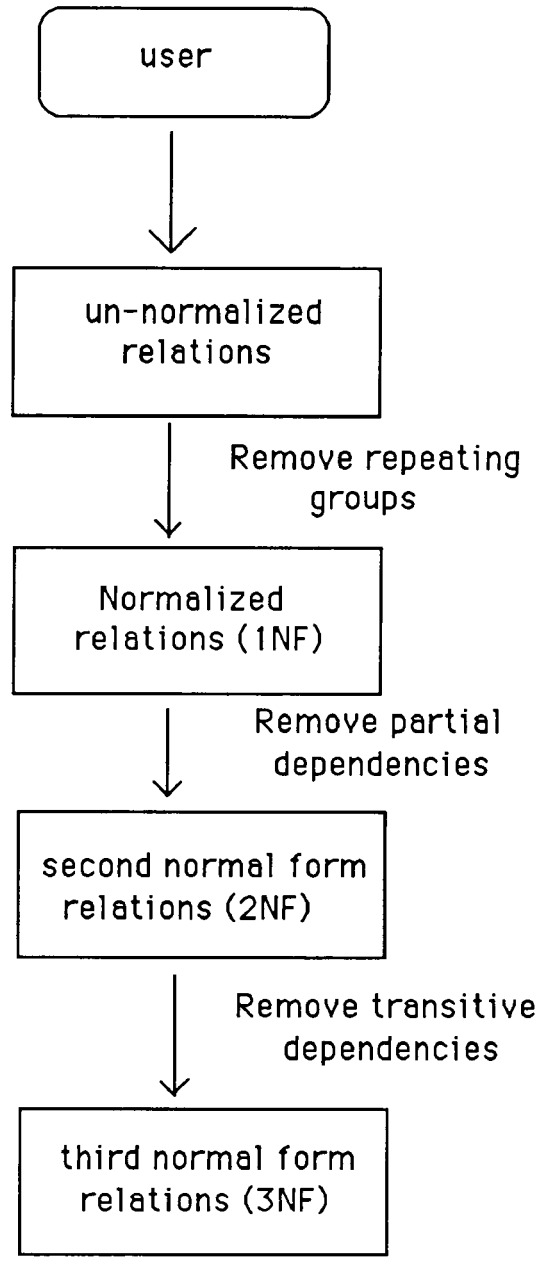


Figure 3.7 Steps in normalization

In figure 3.7, the **first normal form** (1NF) is a relation which contains no repeated groups. **Second normal form** (2NF) is a relation which is already in 1NF and any partial functional dependencies (attributes depend only on one primary key which is a data item that uniquely identifies a record) have been removed. Finally, the **third normal form** is a relation which is

already in 2NF and contains no transitive (hidden) dependencies. **Boyce-Codd normal forms** and **the fourth normal forms** are later additions to the third normal forms but under normal circumstances, nothing beyond the third normal forms should be required.

INGRES [47] was a relational DBMS developed at the University of California at Berkeley originally running on the Unix operating system. Since the first implementation of INGRES, it can now be run on a wide variety of operating systems such as VMS[1], PC-DOS[2] and MS-DOS[3]. INGRES has a relational query language modelled after the ALPHA data sub-language called QUEL. Another tool for information retrieval in INGRES is EQUEL which allows QUEL statements to be embedded in a program. Additionally, another form of query language that accesses an INGRES relational database is called QUERY BY EXAMPLE. This allows queries by entering an example of a possible answer in the appropriate place in an empty table. Each operation is specified by using one or more tables built up on the screen, with field names being supplied by the system and other parts by the user.

3.3.2.2. Database access methods

An **access method** is a file management subprogram provided by the operating system. It is responsible for delivering a single stored record (whether it is structured or un-structured) of a file to and from an application program. Access methods normally support the following services (which are transparent to the application programmer) : blocking and de-blocking of records, locating and accessing required data blocks and transmitting them between main memory and secondary memory, and handling exceptions. When a program is executed, a copy of the access method, which is usually maintained in the system library, will be

---

1 VMS is a trademark of Digital Equipment Corporation.

2 PC-DOS is a trademark of IBM.

3 MS-DOC is a trademark of Microsoft Incorporation.

linked to it. Thereafter, whenever a READ or WRITE instruction is reached, the CPU switches execution to the access method. The concept of access methods is essential as they insulate user from many hardware details. They also allow a file to be visualized as a series of storage locations. Each storage location can hold one or more records and may be addressed by specifying its location relative to the beginning of a file. The access method translates this relative address to the appropriate hardware address and manages other hardware-dependent details as discussed shortly.

In order to describe the access methods used in the database approach, the following terms must be defined first:

A **record** is a namely collection of data items and/or data aggregates. A **data aggregate** is a collection of data items that is named and referred as a whole, eg. a data aggregate called Name might be consisted of data items Last-name, First-name and Middle-name.

A **key** is a data item used to identify a record. Essentially, there are two types of keys: **primary key** and **secondary key**. A primary key is a data item that uniquely identifies a record. The primary key of a record corresponds to the identifier of a real world entity. For example, the data item student number would be probably the primary key for each student's record, although there may be several possible primary keys for the same record. On the other hand, a secondary key is a data item that normally does not uniquely identify a record but instead it identifies a number of records in a set that share the same property. For example, the course taken by a student might be used as a secondary key for the student's record.

A **file** is a named collection of all occurrences of a given record type. A **file organization** is a technique for physically arranging the records of file on a secondary storage device (usually magnetic disc). The most commonly used file organization is the **indexed sequential organization**. The reason is that this kind of organization allows access to records efficiently in both the **sequential** and **random** modes as described shortly. Currently, there are two

implementations of the index-sequential organization: hardware-dependent and hardware-independent, usually referred as the **indexed sequential access method** (ISAM) and the **virtual sequential access method** (VSAM) respectively. An **access method** is a file management sub-program provided by the operating system. VSAM is newer and more powerful and therefore it has replaced ISAM in many applications. Nevertheless, both of these access methods operate on records that are stored in sequence according to a primary key. In addition, the DBMS using these access methods would build an **index**, separated from the data records, which contains key values together with pointers to the data records themselves. The type of index used is referred to as a **block index** in which each index entry refers to a block of records rather than just a single record. The index in the yellow pages of a telephone directory is a very good example of a block index. By looking at this block index, one can quickly locate the page that contains the particular subject in interest and then scans through that page until the desired company name is found. The basic techniques of ISAM and VSAM will be described separately in the following sub-sections.

3.3.2.2.1. Indexed sequential access method

Architecture :

In this kind of access method, the block indexes are organized by tracks and cylinders and this is the reason that it is called a hardware-oriented implementation. Figure 3.8 shows the architecture of ISAM and this figure will be referenced throughout the discussion of ISAM:

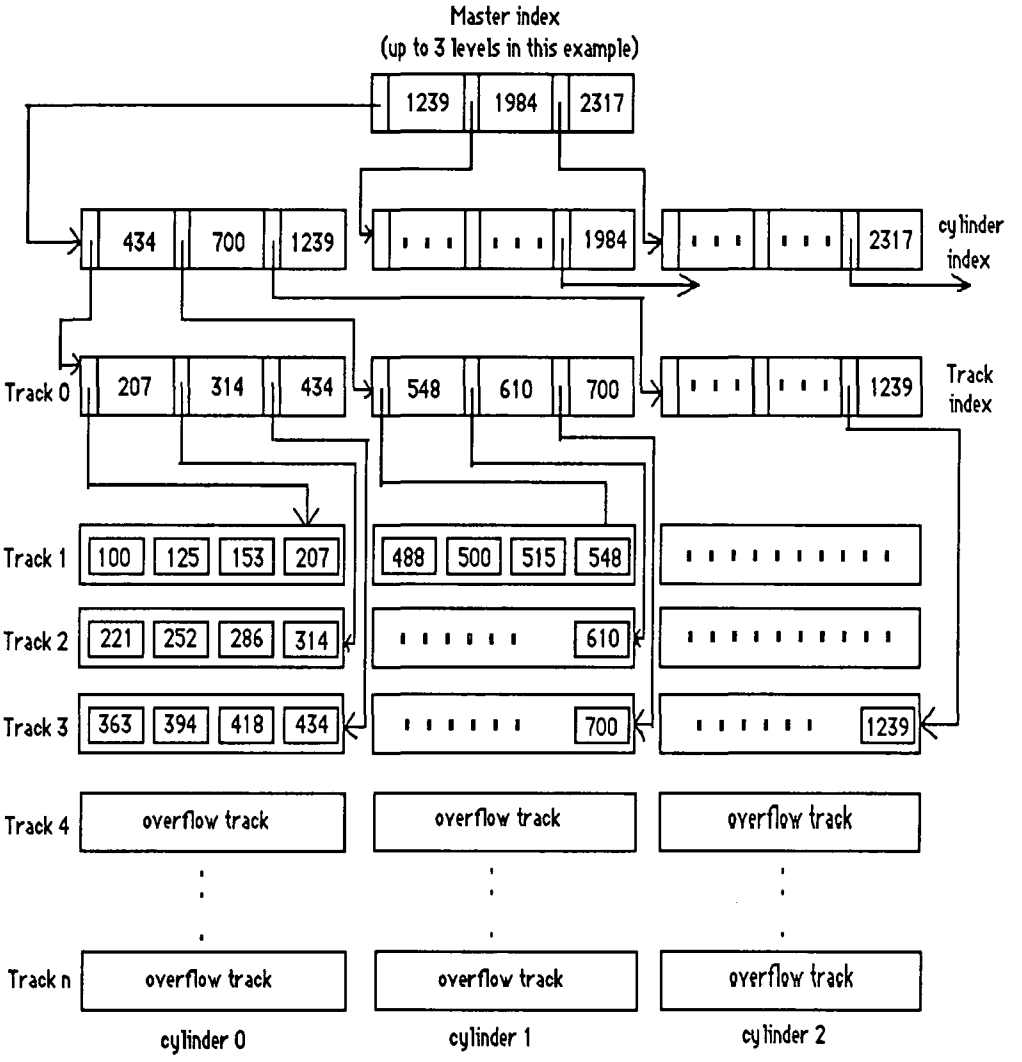


Figure 3.8 ISAM architecture

Three area can be found normally in ISAM :

(a) The **prime** area which contains the data records and the track indexes.

(b) An **overflow** area which contains the overflow tracks and it is used to store records that are added to the file but does not fit in the prime area.

(c) An **index** which contains the master index and the cylinder indexes.

Also, in ISAM, data records of a file are organized according to the following rules :

(a) All data records are stored in ascending order using a primary key (usually the record number) such that the last records of each track always have a greater primary key value than the others in the same track. One exception is that, in the overflow area data records are simply chained together without regard of the primary key values.

(b) No record will be stored in a new cylinder unless all the tracks (except tracks in the overflow area) of the previous track are full.

(c) The notion of **multi-level indexes** is adopted in order to organize the records of a file. Referring to Figure 3.8, each arrow in this figure represents a pointer from one index to a lower-level index. Suppose the record 500 is the target record. First of all, the master index is searched until a value that equals or exceeds the target record number is found, which is 1239 in this case, and this will direct the search to the first cylinder index. Then, searching the cylinder index using the same strategy just described which indicates record 500 is in cylinder 1. At this point, the disc head will be moved to cylinder 1 and the track index for that cylinder is read into memory (the track index is assumed to be on track 0). By searching that track index, it can be found that the desired record is on track 1 of that cylinder. Finally, the target record 500 is read by scanning track 1. This mechanism is termed as the **random access mode**. ISAM also permits rapid **sequential access mode**. Since records are in primary key sequence (except in the overflow area), an entire cylinder can be read without moving the access arm. Therefore, records can be read from the beginning until the desire record is reached.

Processing ISAM files :

When records are first loaded into an ISAM file, the access method will create the indexes for the file as those shown in Figure 3.8. During subsequent processing of that file, searching and maintenance of this index are also carried out by the access method and are completely transparent to the application program. Although application programs are the only means to access an ISAM file., they can perform various operations on the file.

A program can request a particular record by specifying its primary key value. The access method (not the program) will search the indexes using the mechanism described above and then delivers the requested record to the program's data area.

Updating records in an ISAM file is carried out as the following. Firstly, the required data record is read into the main memory using either random or sequential mode. The record is then modified and written on top of the old one.

Records can also be deleted from an ISAM file but not physically removed (or erased) from the file immediately. Instead, a special delete character is placed in the first character position of each deleted record. This character is then used in subsequent accesses to inform the application program that the record has been logically deleted from the file. This deletion method causes a side-effect as explained later.

Inserting a new record into an ISAM file (actually all sequential files) presents special problems. These problems arise from the matter of how the records are maintained in primary key sequence. One obvious solution is to push all the records down beyond the point of insertion. However, it turns out not to be too efficient practically. So, two other techniques are emerged. In the first one, some free spaces are left on the track (not shown in Figure 3.8). Those free spaces will allow occasional insertions but not for a cluster of records. For a large amount of new records, the second technique is used. In this method, an overflow area is required which is usually at the last few

tracks of each cylinder. For instance, if record 176 is needed to be inserted to the file of Figure 3.8, this record will be inserted on track 1 in order to maintain primary key sequence. This means record 207 has to be replaced by record 176 and therefore record 207 must be moved to one of the overflow track. Subsequently, a pointer will be created in the track index giving the address of the first overflow record of track 1 as indicated in Figure 3.9 below:

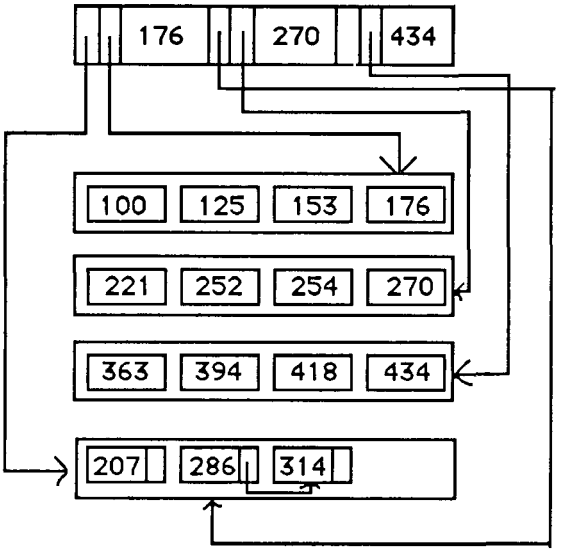


Figure 3.9 Inserting a new record into an ISAM

Similarly, records numbered 254 and 270 will be inserted into the file on track 2. The records they replaced, 286 and 314, will be moved to the overflow track and then they are chained together by an arrow from record 286 to record 314. One important point is that the track index has to be updated after the insertion immediately.

One drawback of the second technique is that over a period of time, the number of records in the overflow areas of an ISAM file will increase whereas the performance of the system will decline since more accesses are needed on the average to retrieve each record. Thus, an ISAM file has to be reorganized periodically. In the reorganization process, the entire file is re-loaded and

records in the overflow areas are moved to their proper location in the prime area. Also, the indexes are updated if necessary. All these operations will be carried out by the access method.

Summary of ISAM :

The major advantages of an ISAM organization are that the file can be processed both in sequential and random fashions, new records can be inserted in the middle of the file and then processed either randomly or sequentially. However, the disadvantages of this file organization are that the file must be reorganized from time to time to clean up overflow records and deleted records, random access to individual records is relatively slow and the indexes are organized by hardware boundaries (tracks and cylinders). Because of this last issue, when a file is transferred to a new disc probably with greater track capacity, the indexes must be completely reorganized again which is time-consuming.

3.3.2.2.2. Virtual sequential access method

Architecture :

This type of file organization also uses multi-level indexes, which is similar in concept to the one used by the ISAM with one major distinction. That is, indexes are free of boundaries, i.e. tracks and cylinders. The basic architecture of VSAM is depicted in Figure 3.10.

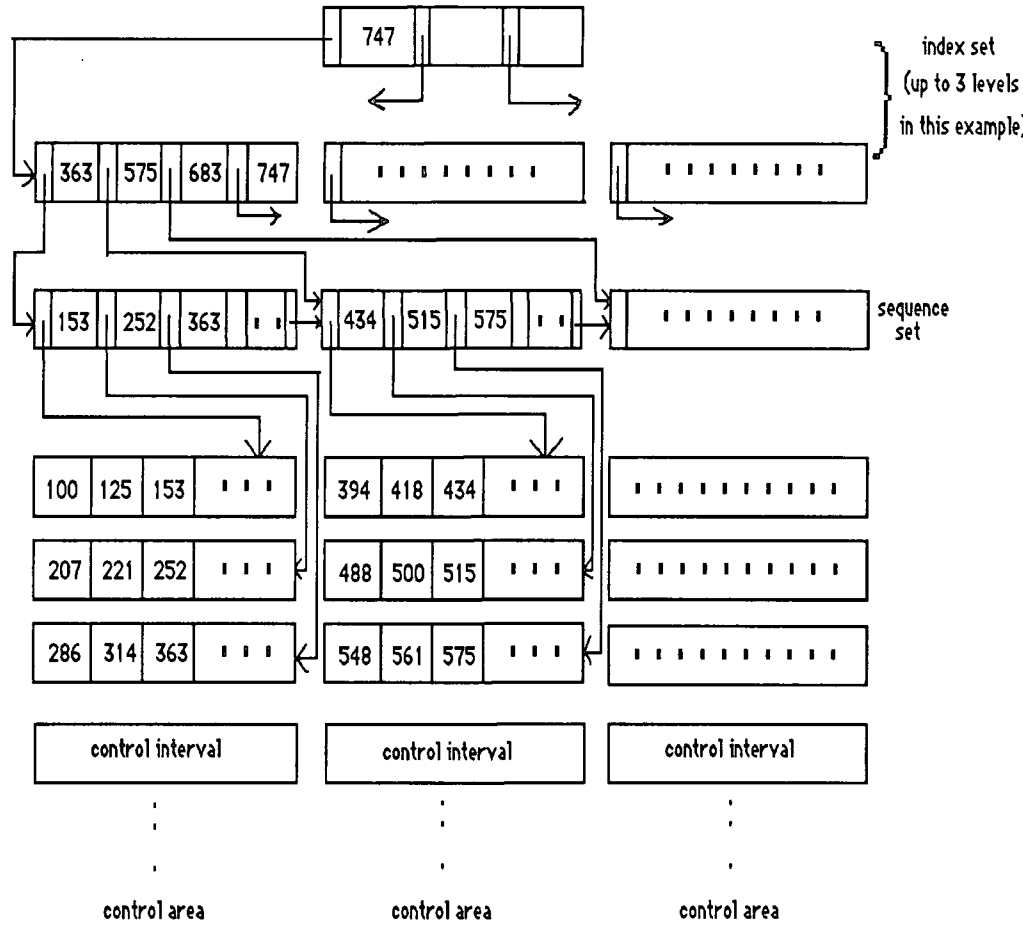


Figure 3.10 VSAM architecture

In a VSAM file, the basic indexed group is called a **control interval** which may be considered as a virtual track. The size of the control intervals are chosen by the file designers which may be less than, equal to or greater than the length of the disc track. Control intervals in VSAM are grouped into **control areas** (or called the virtual cylinders). As shown in Figure 3.10, spaces are reserved automatically at the end of each interval for the insertions of new records. This is so-called **distributed free space**. Also, some control intervals in each control area are left empty. The amount of empty spaces in each control interval and the number of empty control intervals in a control area are specified by the file designer initially before any file is loaded. As with ISAM, data records of a file are organized in primary key sequence in the control intervals.

The index structure in VSAM is very similar to that in ISAM. The index is divided into two parts : the **index set** (up to 3 levels in the following example) and the **sequence set**. One additional feature in the index structure of VSAM is that the components of the sequence set are linked together via pointers (see Figure 3.10). This chain is used for sequential processing as described later. As with ISAM, locating a random record proceeds by starting with the highest level in the index set and progressively searching the index until the aimed control interval is identified. The control interval is then scanned to locate the desired record.

Processing VSAM files :

When a file is first loaded, all its records are arranged in primary key sequence as shown in Figure 3.10. Again, all the processings on VSAM files are performed using the application programs.

In VSAM, the mechanisms used for updating and deleting records are exactly the same as those in ISAM. However, VSAM uses more refined and efficient methods for handling record insertions. Two situations may occur during an insertion:

(a) If the intended control interval for a new record is not full, all the existing records are moved to the right by the access method and the new record is inserted in key sequence as illustrated by the example in Figure 3.11, where record 350 has been inserted into control interval 2. To make room for this new record, record 363 is moved back in that control interval.

(b) However, if the appropriate interval is full, a different strategy is needed as in the following case. Suppose record 240 is inserted into the file in Figure 3.11. This record should be placed in control interval 1 between records 221 and 252 according to the primary sequence key rule. Since control interval 1 is already full, the access method (VSAM) needs to perform an operation namely **control interval split** : half of the records in control interval 1, records 252 and 263, are placed into the

first empty control interval which is control interval 3 in Figure 3.11(b). A new entry is also placed in the appropriate position of the sequence set so that this new control interval can be accessed subsequently.

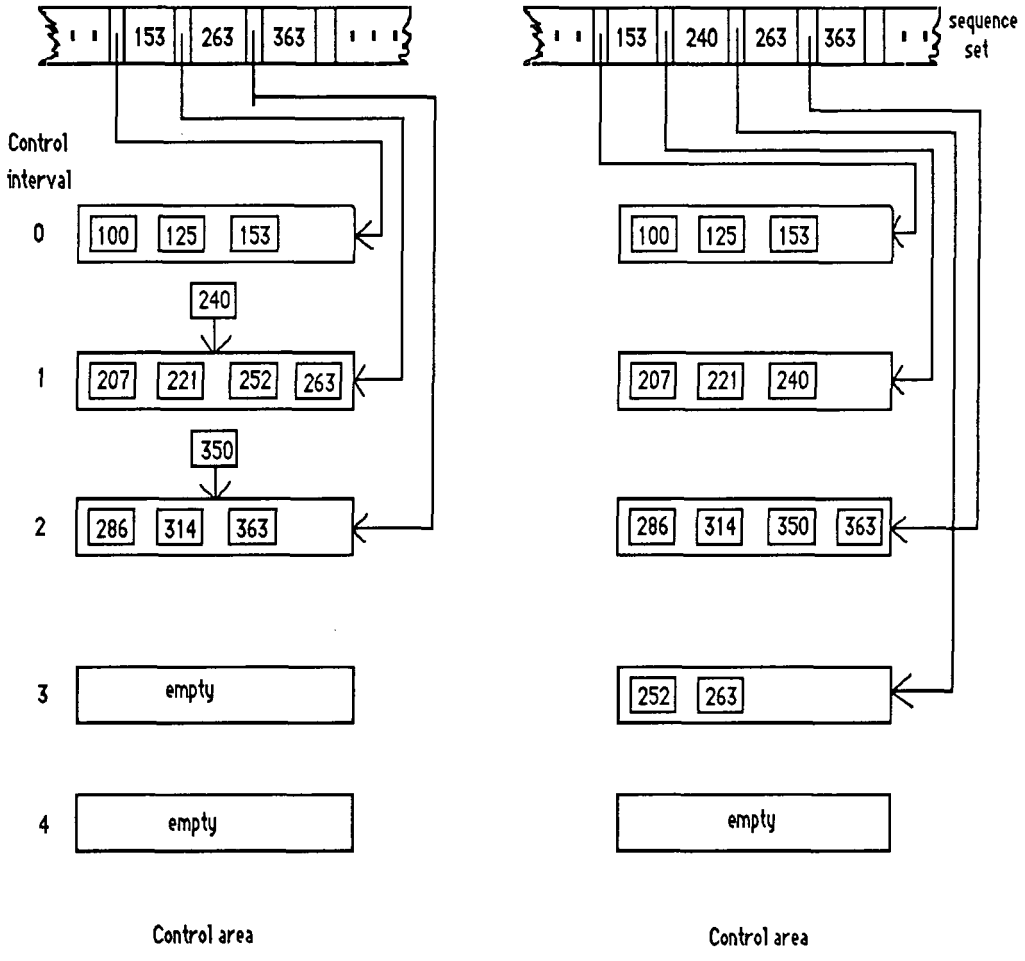


Figure 3.11 Managing record insertions in VSAM
(a) Before the insertions,
(b) After the insertions.

Normally, records are maintained in sequence within each control interval. However, after a control interval split, the records will be no longer

in sequence within the control area as a whole. Fortunately, it can be noticed that the entries in the sequence set are still in primary key sequence. This elegant arrangement and the chain which connected the components of the sequence set allow VSAM to access records of a file in a logical order, i.e. sequentially.

Furthermore, a control area may become full after a large number of insertions. When this happens, it is no longer possible to perform another control interval split for further insertions since all control intervals have been occupied. In this circumstance, VSAM will carry out a **control area split**, that is, allocating a new control area to the file. Approximately, half the records in the control area (that has become full) are moved to a new control area and then the indexes are adjusted to reflect the new file structure.

Summary of VSAM :

VSAM needs a more sophisticated program for file handlings than ISAM. Besides, VSAM still suffers the same problem of cleaning up the deleted records as in ISAM but VSAM offers three major advantages over ISAM. First, periodic file organization is not required since the file can grow indefinitely by means of the two splitting processes as described before. Second, the file organization is independent of hardware characteristics. Hence, a file can be moved to a different volume without restructuring the indexes. Finally, some versions of VSAM can support secondary keys although this issue is outside the scope of this thesis.

3.3.3. Benefits of using the database approach

With the aid of Figure 3.3, this section summarizes all the advantages of the database approach over the conventional data file approach (i.e. the traditional file services mentioned in chapter two).

(a) **Minimal data redundancy** :

As illustrated in Figure 3.3, all redundant data files are integrated into one single and logical structure. In addition, each data item occurrence is ideally recorded in only one place - the **database**. As a result, it will save a lot of storage spaces and perhaps some input time when all the occurrences of a data item are being updated. Despite of the fact that in some circumstances, multiple copies of the same data (eg. for data validation checks) are required, the redundancy is controlled in the database approach because the system is awared of the redundancy.

(b) **Data consistency** :

By controlling (or eliminating) data redundancy, a database approach reduces greatly the chance of data inconsistency because each data item is usually stored only once. Even if the data item appears in more than one place, the database system (DBMS) will enforce consistency by updating each occurrence of the data item when a change occurs.

(c) **Data integration** :

In a database system, data are organized into a global structure, with logical relationships defined between associated data entities. In this way, user can easily relate one item to another related item.

(d) **Data sharing** :

A database is intended to be shared by all authorized users in the user group of Figure 3.3. However, each user is provided with its own view of the database and each of these user views is a subset of the *conceptual database model*. These user views simplify the sharing of data because they provide each user with the precise view of data required to make a decision or perform some functions without making the user aware of the overall complexity of the database.

(e) **Enforcement of standards :**

Establishing data administration functions is a major task of the DBMS. Thus, the DBMS has the authority for defining and enforcing data standards such as the length of data names, data formats and data usages. Moreover, all changes to data standards would have to be approved by the DBMS.

(f) **Ease of application development :**

Another major advantage of the database approach is that the cost and time for developing new business applications are greatly reduced. Referred back to Figure 3.3, application programmers can have direct access rights to the DBMS via interface 2 so they do not have to saddle with the burden of designing, building, maintaining master files etc, as they all have been organized by the DBMS. Hence, the cost of software development is reduced and new applications are available to users in a shorter period of time.

(g) **Uniform security, privacy and integrity controls :**

Since the DBMS of the database system has complete jurisdiction over the database and therefore it takes the responsibility for establishing controls for accessing, updating and protecting data. Having centralized controls and standard procedures can always offer improved levels of database protection. However, if proper controls are not applied, a database system will probably be more vulnerable than a conventional file system since a larger user community is sharing a common resource.

(h) **Data accessibility and responsiveness :**

A database system provides multiple retrieval paths to each item of data. Retrieval of data can sometimes cross several domains. To illustrate this point (see Figure 3.3), suppose a customer call requesting information about several items that have been ordered. While on the phone, the salesperson can

lookup the customer's record to display the particular order in question and then display the product record for each item on that order. Finally, the work order status for each item is displayed to determine their completion date, for instance. From this example, it can be realized that a particular item of the database can be reached using several different paths.

Although the above example is a planned sequence of retrievals usually written in an application program, a database system can also satisfy certain ad-hoc requests for data without the need of application programs by using a **user-oriented query language**, which is compatible with the database system. For example, INGRES's QUEL language. Query languages permit an interactive programmer to construct record retrievals using expressions that specify which records are desired, not the process of record-by-record retrieval. So, the database approach is generally more responsive to changing information requirements.

(i) **Data independency** :

The separation of data descriptions from the application programs that use the data is called **data independency**. The DBA in Figure 3.3 defines the data formats in conjunction with the Data Dictionary, whereas application programmers write programs to process user requests for data. As a result, an organization's data can be changed and evolved (within limits) without necessitating a change in the application programs which process them. Therefore, data independency allows various changes to the database with minimum impact on its users.

(j) **Reduced program maintenance** :

In a fast-growing organization, data may have to change frequently for a number of reasons. For instance, new data item types are added, data formats are changed, new storage devices or access methods are introduced and so on. In database terminology, the term **maintenance** refers to modifying or re-writing old programs to make them conform to the new data formats, access

methods and so forth. Since data independency is achieved in a database system as described previously, either the data or the application programs that use the data can be changed without affecting the others. Thus, program maintenance can be reduced significantly.

So far, ten chief benefits of the database approach have been identified. Many of them can also be translated into business terms because they result in reduced costs of programming new applications, reduced costs of program maintenance, improved quality of managerial decisions and also reduced costs of information retrieval. Thus, database approach is extremely powerful and this is the main motivation for many research projects in this area lately. The next section will describe the basic concepts of one of the modern database technology.

3.3.4. Distributed databases

Although the principles of the database approach should prevail well into the future, the data management technologies used nowadays are inevitably replaced or enhanced by emerging equipment, software and ingenious combinations of DBMS with other technologies. Recently, the most fast-growing data management technology is the **distributed database** (DDB) technology [48].

A distributed database is a database stored on secondary memory devices (usually disks) which are attached to several electronically connected computers and from which a user at one of the computers can access data stored at any other computers. The computers may be relatively far apart geographically, or they may be in adjacent rooms or even in the same room. The point is that there are two or more connected computers involved in managing parts of a single database and subsequently a program can access the data stored at multiple nodes of the computer network.

The notion of DDB is quite different from the notion of de-centralized database. A de-centralized database is a database stored on secondary memory devices which are at separate, independent computers and a user at one computer cannot access or query data stored at other computers in the network. Although one can still consider the stored data conceptually as one database, a de-centralized database is actually implemented as a **set** of databases.

3.3.4.1. Distributed database management systems

In order to have a DDB, a distributed DBMS must be present which co-ordinates access to data at the various *nodes*. Even though each node may have a local DBMS managing local data at that site, a master DBMS is still required to determine from a distributed data dictionary the location from which to retrieve the requested data; to compose response to the data request; to translate a request from one node using a local DBMS to another node using possibly another DBMS and different data model; and also to provide security, concurrency, deadlock control, crash recovery and other data management functions across the various parts of the database. The general architecture of a distributed computing system with a distributed DBMS is shown in the following diagram :

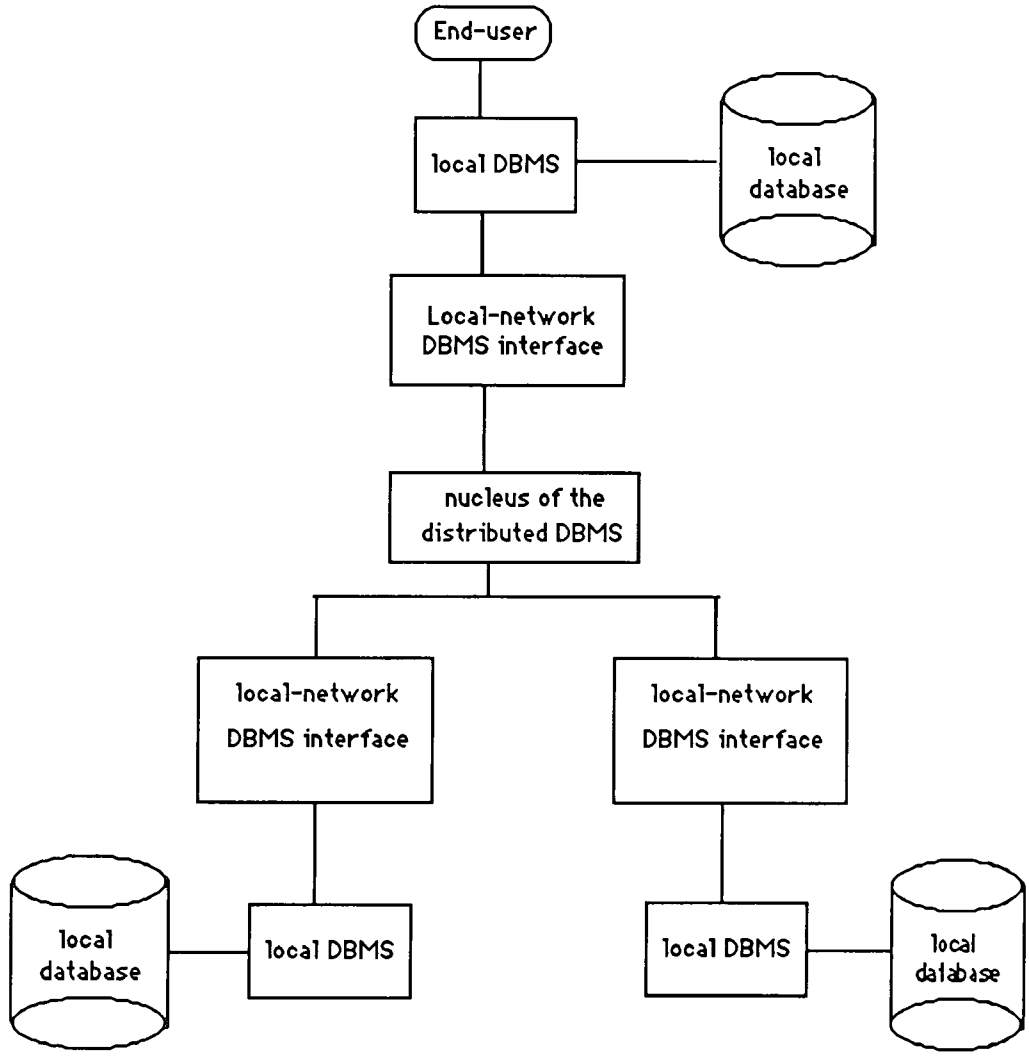


Figure 3.12 A distributed DBMS environment

The distributed environment depicted above consists of local databases with associated DBMS at these sites. Interfaces between the local DBMSs and the distributed DBMS may need to translate data requests from a local syntax and data model to an intermediate level when a mixed DBMS configuration is supported. The nucleus of the distributed DBMS (see Figure 3.12) co-ordinates the internode communication of retrieval requests and data.

3.3.4.2. Objectives and design issues of DDBs

The goal of a distributed database is to provide data management services as transparently as possible so that the user does not realize that data are being accessed from multiple locations. That is, the database should appear to be stored at only one site and it is managed by the local DBMS at the user's site. Furthermore, data maintenance from one site should be performed as if the data to be updated, added or deleted, were stored at only that site despite the possibility that the data may be at one or more sites. To achieve these goals, two issues are considered : the **logical** and **physical** database designs.

All aspects that related to the view of a database that an application programmer is or may be presented, are collectively described under the topic of logical database design. On the other hand, physical database design concerns those aspects that related to the physical placement of data on the storage media which should ideally be transparent to the application programmers. The rest of this sub-section will only focus on the logical design aspects whilst a discussion of those physical design aspects can be found in Appendix A.

3.3.4.2.1. Logical database design

System architecture :

First of all, a data model is required which defines how the database is represented. Secondly, all the main functions of the system, and the most important, frequent or time-critical traversal paths (or relationships) around the database must be identified. The effect of the latter is that :

(a) It validates the data model by showing that all the necessary navigations can be done.

(b) It may simplify the data model by identifying some seldom used paths.

(c) It gives an indication of relationships that are traversed frequently, so that some special functions can be developed for them to enhance the database performance.

Distribution performance :

Two major issues of distribution performances are :

(a) Is the database closely integrated or is it possible to divide it into more loosely related elements?

(b) Even if the database can be divided, is it advantageous to de-centralize it?

The first question can be answered directly from the statistics of activity on records, i.e. the traversal paths as follows:

(1) Identify all the **low-activity** record types. Low-activity means to look for a **break** in the distribution of activity across record types. Is there somewhere that the activity is one or two magnitude lower than the rest? If there is no break, does the *activity distribution curve* slope steeply downwards in the region of the least active 20% ?

(2) Assuming that low-activity record types can be identified, do they appear at random in the database or can they be joined to form boundaries by which the database can be divided? This database division process can either be achieved by drawing a line between record types or across occurrences of the same record type. If the latter is used, one must ensure that all different sets are low-activity.

If both (1) and (2) are successful, then the database can be split into two, three or any number of what effectively are closely but almost self-contained databases. So, if it is possible to do this, distribution is possible; otherwise the centralized alternative is the only feasible solution.

However, a database, that can be distributed, does not yet prove that it should be. Another crucial factor is in relation to the geographical locations where inputs are received to those parts of the database that they require. In other words, one has to assess whether the possible divisions in the database related to the locations from which transactions are received; whether therefore it is possible to break up the total system, database and functions, into a number of sub-systems, each consisting of a subset of the database and a subset of the total system's functions. If all these can be fulfilled, each sub-systems can be geographically distributed. In this situation, the transmission costs are likely to be reduced by de-centralization of the data and functions.

The underlying principle of the above strategy is that: It is not feasible or economic to distribute a database that needs to be closely integrated. Such a database and such a system run best on centralized hardware. The realistic objective of distributed database technology is to permit the implementation of a distributed database, even though sometimes a relationship is used, that cuts across geographical boundaries.

Cost of a DDB :

Cost in implementing a DDB is considered in the following ways:

(a) **Hardware** : The main tradeoff is likely to be between (1) one central processor configuration with a large number of long lines, not all with a large bandwidth and possibly not all very highly utilized. This pattern follows a **star-type** network, (2) one smaller central processor, several distributed processors, a smaller number of wide band lines, several shorter lines from the users to their local distributed processors. This pattern follows from a **partially distributed** star-type network, (3) several distributed processors, all of similar size, wide band lines joining them and relatively short local lines. This pattern follows a **ring-type** network.

In general, one is trying to minimize both the cost of data communication and total expenditure on processors.

(b) **Software** : Tradeoffs on hardware means little unless software is taken into account. Complex DBMSs and operating systems are only available on large mainframes. Minicomputer software is built for simplicity and performance. In order to implement a sophisticated DDB, the only solution is to use large mainframes such as the IBM/360 machines in the expense of high cost and wasting some of the computing power of the mainframes.

Finally, another important consideration is the availability of a required DBMS package on the machine types that are used to build the distributed system.

System development :

Since DDB is still a relatively new invention, much research has to be carried out, therefore a DDB is sold as an experimental venture. So, a more realistic approach is to develop several independent systems with the following foundations in mind:

(a) standardize on data names and field formats,

(b) standardize on file names,

(c) standardize on DBMS software,

(d) standardize on database design,

(e) and leaves **hooks** in the design of the basic software so that communication modules can be added when required.

If the distributed system does not eventually get implemented, there are very little to lose. In the worst case, the user organization will still have a

computer facility (the several independent systems) and some data interchange can always be achieved by transporting magnetic tapes, disks etc.

3.3.4.3. Tradeoffs of DDBs

Advantages :

DDBs permit localized, centralized and de-centralized data managements in an organization by means of the local and distributed DBMSs. DDBs reduce data communications traffic by storing data close to the node which uses the data most frequently as well as supporting convenient consolidation of data across locations. This reduced communication time has a considerable impact on processing throughput (rate) and response time. Furthermore, data is more consistent since there is a single DBMS which co-ordinates synchronized updating of redundant data. Finally, DDB permit specialization of hardware and DBMS software at each node to suit the needs at that site without sacrificing the integration of information resources of the organization. Prudent redundant storage of data can achieve greater reliability and availability because users are not dependent on data from only one site.

Disadvantages :

Current distributed DBMS technologies permit less freedom of choice than may be intended, regarding the mixing of DBMSs at different sites. Another shortcoming of DDBs is that they are expensive and not widely used yet, therefore the user of a distributed DBMS is at the forefront of DBMS technology with all the concomitant risks.

3.4. Conclusions

This chapter has addressed the major characteristics of structured and un-structured distributed file servers. Two examples of un-structured file servers are given: the Cambridge File Server (CFS) and Newcastle Connection (NC). Database systems are typical examples of structured file servers. One of the modern database technology, the distributed database (DDB), has also been discussed.

Conclusively, a distributed file system (DFS), such as CFS and NC, differs greatly from a distributed database in the difference of complexity between their implementations and interfaces.

For the former case, one can consider the problem of **file allocation**. That is, the problem of where to allocate a file and its copies, given a known set of retrievals and updates and their execution frequencies such that a cost function is minimized. The solutions for the file allocation problem in a DFS do not characterize solutions to the same problem in a DDB for the following reasons:

(a) The objects to be allocated are not known prior to the allocation. Relations which describe logical relationships between data are not suited as units of allocation because users at different sites might be interested in different fragments of a relation.

(b) The way of accessing the data is far more complex. In the file allocation problem, the only transmissions required to combine data from different files are transmissions from sites containing files to the result site where the result is computed. In current research in distributed query processings, it can be observed that to process a query, data transmissions between sites where fragments are allocated are also needed. This means that the fragments cannot be allocated independently.

N.B. The difference between a query and an update is that the latter will access all copies of a file whereas the former access only one (usually the one nearest to the user machine).

On the other hand, in order to distinguish the complexity in interface designs between a DFS and a DDB, the problem of **data retrieval** is considered.

To retrieve data in a DDB, a user process generates a request for data from the distributed database **globally** since the distributed details have been hidden. The distributed DBMS (see Figure 3.12) then intercepts the request and determine where to send it for processing, or which nodes must be accessed to satisfy the request. This process can be facilitated by the introduction of **global directory**. A global directory is capable of indicating the storage nodes at which various units of data within the distributed database environment exist. Having accessed this directory, the distributed DBMS must co-ordinate the processing and response to the user request if it spans nodes, i.e. the target data exists at multiple nodes. The local DBMSs will be responsible for retrieving data at the local databases. Before sending back the required data, a local process called the **local network processor**, which resides on each node to provide the interface to the network, will carry out any data and process translations if necessary. Finally, the distributed DBMS synthesizes all the local retrievals and presents the user process with the global response to the request.

So, it can be realized that the interfaces between the distributed DBMS and the local DBMSs are quite complicated in comparison to those in a DFS. The main reason is that DFS does not need to interpret the meaning of the retrieved data which has left to the user process. Nevertheless, the choice of using a DFS or a DDB for a particular application is often affected by the degree of consistency and reliability required.

Chapter Four -- Abstract Data Types

The concept of *abstract data types* (ADTs) have received much attention lately. This chapter will explain the importance of ADTs, how to define an ADT , how to implement an ADT and how ADT supports the new program design technique of **object-oriented programming**. In order to achieve these goals, the following stack example will be used throughout this chapter.

Example: stack

A **stack** is a collection of data kept in sequence. Each item of data is of the same type. Data is added to and removed from the top of the stack. Operations which can be used on stack include:

(a) **top**- It returns the top item of the stack as result.

(b) **push**- Given an item and a stack, it returns a stack with that item inserted at the top.

(c) **pop**- It takes a stack and returns the same stack with its top item removed.

(d) **createstack**- It returns a new empty stack.

(e) **isemptystack**- It returns true if a stack is empty, false otherwise.

4.1. Motivations for Abstract Data Types

Commercial programs are often large and complex pieces of software written by many professional computer programmers. Each programmer or group of programmers will be responsible for part of the program in order to save time and human effort. For most of the time, each of these units will concentrate on their own

responsibility. In other words, complex tasks are sub-divided into parts which are themselves broken up until the pieces are sufficiently small to be handled by a small individual. This kind of technique is called **modularity**. A program is said to be modular if its various **modules** (components) are relatively independent of one another. That is, each module can operate correctly even in the absence of the others. Modularity is a procedural decomposition because it is the method by which a program achieves its purpose that is modularized. Apart from achieving modularity, a good design also needs to minimize the degree of *coupling* between modules and to maximize the *cohesion* of individual modules. Coupling is concerned with the inter-connections between modules; whereas cohesion is concerned with the responsibility of an individual module for the other modules or functions. Usually coupling cannot be avoided completely since the modules of a large program have to co-operate in some way. But the more tightly modules are coupled, the more difficult they will be to implement, test and modify individually.

Besides, even a perfectly working software system will become obselete as time goes by so it is essential to separate each stage of the *software life cycle* as shown in Figure 4.1. According to this diagram, if the requirements of the software are changed, it will change the specifications and which will change the design and the design will then change the implementation and so on. Thus, the whole process is **cyclic**.

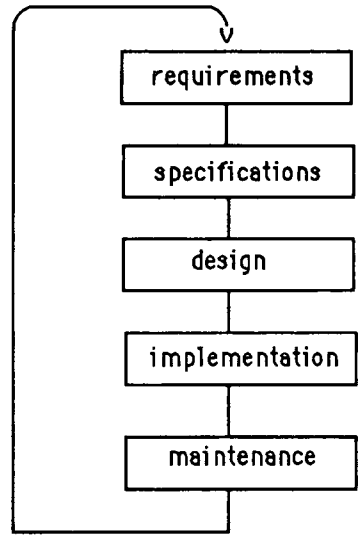


Figure 4.1 Software life cycle

Furthermore, software is an expensive commodity, especially when compared with hardware cost which is becoming cheaper all the time. This is partly due to software complexity and that means it is hard to produce a totally **correct** software. An aim to devise methodologies and tools which can reduce the incidence of errors in the final product is then required. The cost of a software can be divided into two parts:

(a) the cost of building the software in the first instance, i.e. the first four stages of the software life cycle.

(b) the cost of maintenance-the final stage of the life cycle.

The latter usually exceeds the former mainly because of the fast pace and change in industry and commerce. A technique is therefore needed to reduce errors and to enable software to be easily modified later.

Conclusively, the desperate demand in finding a way to help programmers to separate level of concerns, to reduce software production costs and to produce reliable software products leads to the tremendous development of **data abstraction.**

4.2. Formal definition of an ADT

Before defining the meaning of an ADT, some terms must be introduced first. The field of study involves the use and implementation of data objects is usually called **data structures**. Data structures serve to implement complex data objects and these objects are classified into types according to the way they are used.

Horowitz [49] put forward the view that a **data type** is a set of objects and an associated set of operations. For example, a stack is a last-in-first-out (LIFO) list which usually has those operations described at the beginning of this chapter. Another obvious example is the set of integers which is a data type with operations addition, substraction, multiplication and division etc. Moreover, one must also differentiate the idea of user-defined types such as the enumerated types in Pascal with ADTs. The

concept of ADTs aims to deal with types defined by users who can specify their own access operations as explained later.

Since the concept of ADTs is a modern one, several authors have attempted to define it. Horowitz has pointed out that the idea of ADTs in programming languages is to provide a mechanism whereby a type definition is isolated and protected from the rest of the program. Access to the data type may be made only in a very restricted manner. The *representation* of the data type will be hidden so that outside the definition of the type the programmer can rely on its representation. The representation of the data and the algorithms for manipulating them are encapsulated to protect the ADTs from being misused and to promote its independence. The power of the abstraction derives from the ability to make use of the ADTs via an externally defined *specifications* of the operations permitted on the type, in isolation from the *implementation* details.

However, Martin [50] has presented another definition for an ADT as following:

An **abstract data type** is basically a pair consisting of : a set of objects and a set of operations that manipulate the objects. Even more precisely, an ADT is a system consisting of three components:

(a) Some set of objects (given, for example, as Pascal type), referred as data.

(b) A set of syntatic descriptions of (primitive) functions, referred as the specification of the ADT.

(c) A semantic description namely the implementation of the ADT. That is, a sufficiently complete set of relationships that specify how the functions interact with each other. The technical term for these relationships is *axioms*. Thus, an implementation is the production of a program which carries out the operations defining an ADT.

This definition gives the overall structure of an ADT and it also addresses to a greater extent the mathematical aspects of an ADT; the previous definition places emphasis on the basic concepts of using ADT in programming.

Nevertheless, the key point of an ADT is that it is a collection of related operations and the behaviour of the ADT can only be seen by observing the results of applying the operations. In order to understand ADT fully, Figure 4.2 depicts a pictorial representation of an ADT.

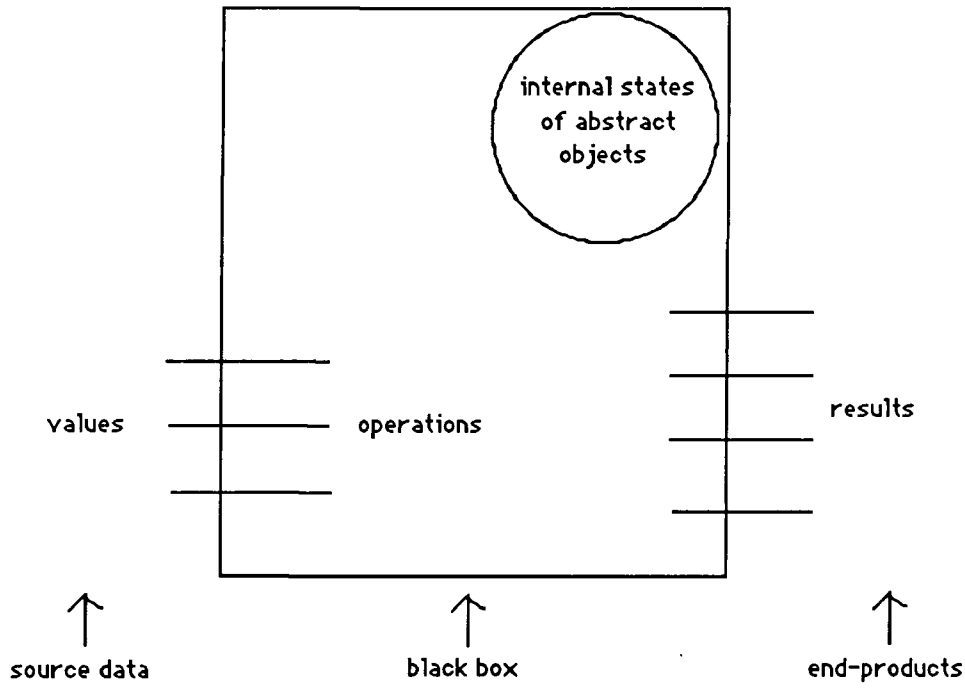


Figure 4.2 A pictorial diagram of an ADT

An ADT is the black box in the diagram which offers the choices of operations to users that define the ADT; users must select the operation and provide the correct source data and then the black box will compute the associated result.

4.3. Specification of an ADT

Like many other definitions in computer science, a language is needed to define a specification. Moreover, precision is required in a specification because it is a pre-requisite for the achievement of correctness and this kind of precision can be

obtained by a *formal language* usually based on some mathematical notations.

Thomas [51, pp. 13] said that "Any programming language is a formal language since a program written in that language has a single, fixed meaning. Utimately, this meaning is defined by the combination of compiler and machine used to execute the program. Repeated executions of the program yields the same result, given the same input, if any, to the execution". Unfortunately, these kinds of formal languages are not suitable for defining the specification of an ADT because they are languages for describing representation but not *abstraction*. So a formal language which encourages abstraction in order to express ideas about specification distinct from representation is required.

Two common approaches to solve the above problem, as noted by Thomas [51], are the **axiomatic** and **constructive** approaches. Some knowledge about the mathematical set theory is required in this particular section.

4.3.1. The axiomatic approach

It is also known as the **algebraic approach**. Before going into details of this approach, the distinction between the syntax and semantics of a formal language must be understood. The syntax of a language specifies all the legal expressions while the semantics concern with the meanings of the various forms of legal expressions in the language. Figure 4.3 shows the full syntax and semantics of the stack example.

Name :

stack(item)

Symbols :

$\forall$ - universal quantifier

i - an item

s - a member of stack

Sets :

S - set of stacks

I - set of items

B - set consisting of Boolean values true and false

M - set of message values consisting of the single member, the stack is empty

Syntax :

createstack: $\rightarrow$ S

top : S $\rightarrow$ I $\cup$ M

pop : S $\rightarrow$ S $\cup$ M

push : I X S $\rightarrow$ S

isemptystack : S $\rightarrow$ B

Semantics :

$\forall_s \in S$, $\forall_i \in I$

top(**push**(i,s))=i

isemptystack(**createstack**)=true

isemptystack(**push**(i,s))=false

pop(**push**(i,s))=s

pop(**createstack**)=the stack is empty

top(**createstack**)=the stack is empty

Figure 4.3 Axiomatic Specification of a stack

In the axiomatic approach, an ADT is defined by its properties as a set of axioms which relate meanings of operations to one another as in Figure 4.4. In Figure 4.4(a), the meanings of **push** and **top** are related to each other by asserting something about the application of one operation to the result of applying another.

if an item is **push**ed onto an existing stack
and
the operation **top** is applied to that stack

then the item that is returned is the one previously **push**ed onto the stack.

(a)

top(**push**(i,s))=i

(b)

Figure 4.4 Description of the stack operations
(a) in plain English text
(b) in formal axiom form

Figure 4.4(a) is an informal axiom whereas Figure 4.4(b) is a formal one. The latter expresses the meanings of **push** and **top** are related. Firstly, **push** is applied to the source data values, i and s. That is , the item i is **push**ed onto the stack s. Secondly, **top** is applied to the result of the **push** operation. Finally, the result of **top** is the value of i.

From this example, it can be realized that the semantics of an ADT rely totally on the precision of the set of axioms defining it. In other words, these axioms are a precise definition of the ADT if they are complete. Addressing completeness, Thomas [51] has suggested two properties:

(a) Complete in the sense that they define the outcome of all permissible applications of the operations of the ADT.

(b) Complete in the sense that they define operations that allow the construction of all permissible instances of the ADT.

In order to examine the issue of completeness, in the sense of defining the outcome of all permissible operations (the first property), consider the following case: according to the semantic definitions given in Figure 4.3, the expression

top(pop(createstack))

is invalid if **pop** is applied to an empty stack (from **createstack**), **pop** will return a value from M - the stack is empty - and the action of **top** will be undefined. One solution to such problems is to return a value from M, as a result, whenever it is used as the source data for application of such **pop** operation discussed above. This solution will lead to the introduction of new axioms as illustrated below:

top(the stack is empty)=the stack is empty
and
pop(the stack is empty)=the stack is empty

Unfortunately, if the expression is complicated or M has more than one member, the introduction of new axioms could lead to a specification that begins to exhibit the very verbosity it is intended to avoid in the first place. Therefore, Thomas suggested a more satisfactory solution called **invariant assertion** that is "Whenever an operation is applied to a value from M then the result of the operation is that same value from M". This specifies precisely what is required.

However, the axioms of Figure 4.3 are complete in the sense of the second property of completeness because there are only two operations, **createstack** and **push**, which allow stacks to be built. Although the other operation **pop** can

modify a stack, it does not produce any stack that could not otherwise built. Hence, it is intuitive that any permissible stack can be constructed as a composition of **createstack** and **push** alone. All expressions which involve these two operations are generally referred to as **reduced expressions**.

Having discussed the issues of the axiomatic approach, it is useful to give a summary of it as follows. An axiomatic specification of semantics involves the **implicit** definition of meaning by relating the semantics of operations to one another by use of axioms. These axioms typically involve the definition of the result of composing operations together; a set of axioms precisely and completely defines the semantics of an ADT. This kind of remoteness from representation makes it an excellent device for encouraging abstraction in a specification.

4.3.2. The constructive approach

This is also known as the **operational approach**. This approach defines the semantics of operations explicitly by relating each individual operation to an underlying model. That is, definitions are built on explicitly from the operations defined in the underlying model which have been precisely defined.

In order to ease the construction process, the principles of **pre-conditions** and **post-conditions** are used. For instance, in the **maximum** operation (written in Pascal) described below:

```
function maximum(x,y:integer):integer;
begin
    if x>=y then maximum:=x
    else maximum:=y
end;
```

The semantics of operation maximum can be given by the following two statements:

$$\text{pre-maximum}(x,y) ::= \text{true}$$

$$\text{post-maximum}(x,y;r) ::= (r \geq x) \ \Lambda \ (r \geq y) \ \Lambda \ (r = x \ V \ r = y)$$

The first statement specifies what must be true about the source data and the second one specifies what must be true about the relationship between the source data and the result. By using these pre-conditions and post-conditions statements, the specifiaction of an ADT can be precisely constructed.

A pre-condition has three components:

(a) The name of the pre-condition, eg. pre-maximum.

(b) Variables representing the input data, eg. variables x,y in pre-maximum.

(c) A declaration of the condition that must hold before the operation can be legally applied, eg. the condition true in pre-maximum.

Three components are also found in a post-condition:

(a) The name of the post-condition, eg. post-maximum.

(b) Variables representing the source data and the result, i.e. x,y and r in post-maximum repectively.

(c) A declaration of the relationship that holds between source data and result after application of the operation, eg. the right-hand side expression of post-maximum.

It is worth emphasizing that the pre- and post- conditons do not represent the applications of the operations they are defining. They represent what must be true before and after the operation is carried out.

The stack example shown in the axiomatic approach can be specified by use of the constructive approach as shown in Figure 4.5 below (all the operations in a combination of italic and bold styles are the operations of an underlying model which are well-defined in terms of an abstract list).

Name :
stack(item)

Symbols :
i - an item
s - a member of stack
r - result of an operation

Sets :
S - set of stacks
I - set of items
B - set consisting of Boolean true and false
M - set of message values consisting of the single member:
the stack is empty.

Syntax :
createstack : → S
top : S → I ∪ M
pop : S → S ∪ M
push : I X S → S
isemptystack : S → B

Semantic :
pre-**createstack**()::=true
post-**createstack**(s)::=s=***createlist***

pre-**top**(s)::=true
post-**top**(s;r)::= if s=***createlist***
then r=the stack is empty
else r=***first***(s)

pre-**pop**(s)::=true
post-**pop**(s;r)::= if s=***createlist***
then r=the stack is empty
else r=***trailer***(s)

pre-**push**(i,s)::=true

post-**push**(i,s;r)::=r=***make***(i)***concatenate*** (s)

pre-**isemptystack**(s)::=true

post-**isemptystack**(s;b)::=b=***isemptylist*** (s)

invariant assertion :

Whenever an operation is applied to a value from M then the result of the operation is that same value from M.

Figure 4.5 Constructive specification of a stack

The general style of this constructive specification is exemplified by the entry for **top**. The pre-condition states that **top** can always be applied to a stack. The post-condition of **top** states the fact that **top** will produce a result either an item or a message depending on the value of source data s. The inclusion of invariant assertion to define the outcome of all permissible stack expressions.

As a summary, the specifications produced by the constructive approach are easier to read and to write than axiomatic specifications. However, this ease of use implies that the underlying model must be capable of defining precise definitions. Lately, ADTs are usually defined axiomatically in the underlying model and then used constructively for more complicated applications. Therefore, the partnership of these two methods is vital.

4.4. Implementation issues of an ADT

This is a two-stage process:

First stage

(a) Investigate data structures for the representation of the ADT,

(b) Investigate suitable algorithms for the operations of the ADT,

Second stage

(a) Choose the most suitable programming language for the implementation of the two issues defined in the first stage.

(b) Turning the data structures and their associated operations into program codes of the chosen language.

One important point about the implementation of an ADT is that it can contain features which have been solely introduced from the representation but not in the specification stage such as the size of the stack.

4.4.1. Facilities possessed by a programming language to support ADTs

Ideally any (high level) programming language ought to have the following ingredients to implement ADTs:

(a) To separate the whole implementation of an ADT from an application program.

(b) To limit access to the implementation of an ADT to the headings of the routines which implement the operations.

(c) To allow the application program to manipulate more than one instance of the ADT without having to replicate the implementation.

(d) To compile separately the implementation of an ADT and make the implementation easily available to more than one application program.

(e) To be able to change the implementation of the ADT without in any way affecting an application program as long as the specification is met.

(f) To be able to define the type of the ADT item in the application program. That is the ability to define generic (i.e. type-free) ADTs.

All these items listed above are concerned with the technique of *information hiding*. This enables programmers to use an ADT without knowing the implementation details. Unfortunately, there are not many programming languages that can satisfy all these ideal requirements. Some of them will be discussed in section 4.5 shortly.

4.4.2. Advantages of Information Hiding

There are a number of key advantages of information hiding for system designs:

(a) An implementation can be changed without causing any functional effects on the application program.

(b) Maintenance of an implementation can take place without affecting the application and vice versa.

(c) Help to track down programming errors because if ADT implementations are proved to be correct, then if something is wrong, it must be in the other parts of the program.

(d) Users do not have to worry about the implementation details of an operation provided that they know the name of the routine; information about its parameters; a description of the purpose of the routine; and how to call the routine.

(e) Re-usable: Since users cannot change the implementation of an ADT, therefore it can be used by other different applications.

(f) Independent of any application: By looking at the implementation, an application programmer may be unduly influenced by what is there and write the application in such a way that it becomes dependent on the implementation of the ADT.

(g) Maintainable: Since only one copy of the implementation exists, consistency can be achieved during an update of the software.

4.4.3. Error detection in ADTs

Finally, this sub-section concludes the discussion on the implementation of ADTs by addressing errors appeared in an implementation. To be exact, errors found in an ADT means that an attempt has been made to violate a constraint. These are actually mistakes in application programs because in the specification stage of an ADT, these constraints have been taken into full accounts as shown in Figure 4.5. One classical example of such a mistake is attempting to use the **top** operation of a stack but having supplied an empty stack.

Two strategies has been proposed by Thomas to solve such constraint violation problems:

(a) Demand the application should detect a particular use of an operation would lead to a constraint violation, eg. **top**ping off an empty stack, then take evasive action to ensure that the event does not occur. Therefore, the only action required by the implementation is to detect when the application program fails to perform a test and take appropriate response such as printing out an error message.

(b) Allow the implementation to detect the constraint violation event and report it to the application program so that remedial action can be taken. This needs additional information to be passed between implementation and application, either via a new parameter (some sort of status flag) or by using one of the existing parameters to indicate the error.

4.5. Applications of ADTs in programming languages

Abstract data types used in programming languages provide programmers with an abstract view of data in which implementation details are separated from a specification of how the type may be used. Essentially, for any programming language to implement

ADTs is to support information hiding which can be achieved through **encapsulation**. The term encapsulation is used to describe those information hiding features exhibited in that particular language. The rest of this section will concentrate on encapsulation features of some programming languages. The main reason for choosing them is due to their popularity and pragmatic importance.

4.5.1. Ada

Ada is one of the modern languages that has been designed with information hiding in mind [52] and its major encapsulation feature is called **package.** In Ada, a program is viewed as a collection of similarly constructed parts - **subprograms**. An Ada package consists of two parts named **specification** and **body**. Diagrammatically, it can be viewed as follows:

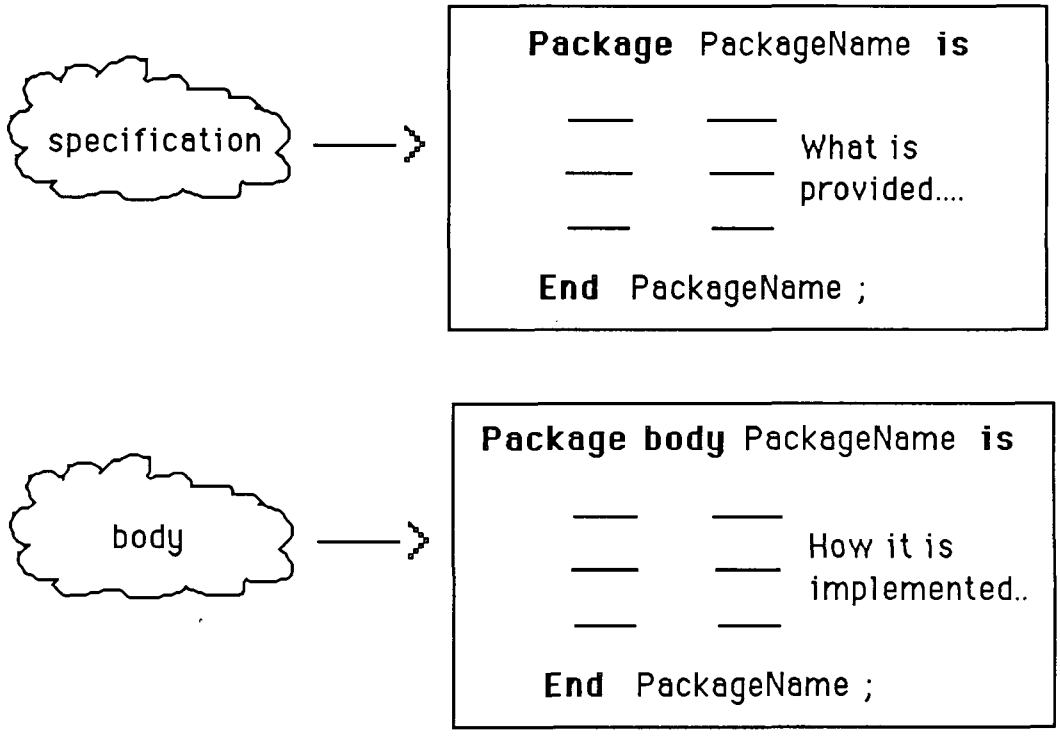


Figure 4.6 Pictorial representation of an Ada package

Both the specification and the body of a package can be constructed separately provided that the latter is conformed to the former. In use, the specification part is available to users and it contains the headings of the routines which implement the operations of an ADT. The body part which contains the implementation of the routines is then hidden from users. As an example, Figure 4.7 shows an Ada package which implements an integer stack by an array and a pointer to indicate the top of the stack.

```
package NumberStack is
    subtype itemtype is integer range 0..10000;
    type stack is private;      -- the function of private will be discussed shortly

    procedure CreateStack(S: out stack);
    function Isemptystack(S: in stack) return boolean;
    function Top(S: in stack) return itemytpe;
    procedure Pop(S: in out stack):
    procedure Push(item: in itemtype; S: in out stack);

  private        --this part defines the internal representation of the ADT
    maxsize: constant integer:=100;
    subtype cursor is integer range 0..maxsize;
    type store is array (1..maxsize) of itemtype;
    type stack is record
                    top: cursor;
                    item: store;
                  end record;
end NumberStack;
```

(a)

```
package body NumberStack is
      procedure CreateStack(S: out stack);
      begin
         S.top:=0;
      end CreateStack;

      function Isemptystack(S: in stack) return boolean;
      begin
         return(S.top=0);
      end Isemptystack;


      function Top(S: in stack) return itemtype;
      begin
        if Isemptystack(S)
        then
           put("The stack is empty");
        else
           return(S.item(S.top))
        end if;
      end Top;

      procedure Pop(S: in out stack);
      begin
        if Isemptystack(S)
        then
          put("The stack is empty");
        else
           S.top:=S.top-1;
        end if;
      end Pop;

      procedure Push(item: in itemtype; S: in out stack);
      begin
        if S.top=maxsize
        then
           put("The stack is empty")
        else
            S.top:=S.top+1;
             S.item(S.top):=item;
        end if;
      end Push;
end NumberStack;
```

(b)

Figure 4.7 An Ada stack package - based on an array and a pointer.

Among those five routines' headings in Figure 4.7, three Ada **modes** of parameters are present. The **in** mode is used for input only and cannot be modified by a routine. In constrast, the **out** mode is used for output only and can be used and modified by a routine. The last mode **in out** can be used for both input and output, and can be used and modified by a routine. These parameters are responsible for updating the stack. Besides these five routine headings, the user also need to know the two type definitions: itemtype and stack. The type stack shows an eexample of encapsulation in Ada. It is declared to be **private** which means its definition will be defined in the **private** part of the specification. Although the representation of this type is still visible to user as shown, the private declarations are effectively local to the package specification: the user will not be allowed to make use of these definitions in any other programs. Thus, the implementation is visible but not accessible. When the body of a package is compiled, the Ada compiler will make use of the type definitions held in the package's specification part and therefore the specification must be compiled prior to the compilation of the corresponding body.

Furthermore, another interesting Ada's encapsulation feature is the provision of libraries. For instance, the procedure ***put*** inside procedures Top, Pop and Push is one of the Ada standard library routines for output which is made available from an I/O package named TEXT_IO. However, since this output routine will not be required until the package NumberStack is invoked, so inside the body of NumberStack, there is no information about the location of ***put***. Thus, it is the user's responsibility to import (or make available) the package TEXT_IO into a new application sub-program before using the package NumberStack. Details of how this is done will be demonstrated in the next sub-section.

The use of the **private** type described earlier has two drawbacks. First of all, ideally the contents of the private part should be included within the package body. If an implementation is to be altered, the changes may be required in both the package body and the private part of the package specification, even though that part of the package specification accessible to users need snot be changed. Moreover, the user's program will need re-compiling since part of the package specification has been altered. Secondly, if the representation of the stack's items is to be changed, then the type definition of **itemtype** must also be changed accordingly. But this definition is

in the user-accessible area which means the user may allow to alter the original intention of the package completely. This could cause disastrous consequence such as insufficient storage spaces, two identifiers have the same name etc. Fortunately, Ada provides another tool called **generic package** (see section 4.5.1.2) which can tackle these problems.

4.5.1.1. Using Ada packages

Taking the package NumberStack as reference, this package can be used by placing the statement

with NumberStack;

in front of the subprogram heading which results in the importation of the NumberStack package into the new sub-program. Thereafter, references to all the visible resources (types, procedures, functions and so on) of this package can be made using the familiar dot-notation as follows:

```
NumberStack.Push(3, onestack);
currentvalue:=NumberStack.Pop(onestack);
```

The package name is pre-fixed by the procedure names Push and Pop so that, in situations where two or more packages have been imported into the same subprogram, no confusion between resources with the same name in different packages will occur. However, in cases where it can be absolutely certained that no such ambiguity will happen, the **use** statement is used instead in conjunction with a **with** statement:

with NumberStack; **use** NumberStack;

The Push and Pop operations can then be referenced without the package name. This **use** statement resembles Pascal's with statement but it is in force for the

whole of the subprogram that it precedes.

4.5.1.2. Special packages in Ada

The prime task of Ada's encapsulation features described so far, is to separate almost totally the implementation of the resources from its specification. In circumstances where there are different application programs need to manipulate several stacks with different types, package NumberStack may not be validated for all of them except the one in which the definition of type itemtype is still an integer. Ada provides a facility called **generic packages** to solve this problem. A generic package is used to create instance of the package known as **generic instantiations** as described below.

The basic idea of generic package can be thought as Figure 4.8.

```
generic
   type itemtype is private;

package Stackpackage is
   type stack is private;
    the five operations of NumberStack;

 private
   same as those of NumberStack;
end Stackpackage;
```

Figure 4.8 A generic stack package

This time the type definition of itemtype which comes right after the keyword **generic** is a so called **generic parameter**. Such a parameter is used to specify, outside the package in some other sub-programs, the type of certain *objects* used within the package. For example, if a stack called **CharStack** (a

stack of characters) is required in an application program, the application program will contain the declaration:

package CharStack **is new** Stackpackage(character);

The effect is to bring into existence a new package called CharStack which is a specific instance of Stackpackage where the type of itemtype is set to be character. Therefore, this generic facility enables generic packages to be written for which certain objects (with distinctive names) have their types defined in later stages.

4.5.1.3. Exception handling in Ada

Ada offers a method of dealing with constraint violations known as **exception handling** which enables information about the occurrence of constraint violations to be communicated between packages and application programs. This mechanism uses a special kind of objects called *exceptions*. To make use of exception handling, three components must be present in an Ada program:

(a) the declaration of exception variables,

(b) **raise** statements to set the exceptions,

(c) **exception handlers** (program codes) to be executed when exceptions have been raised.

For instance, the package NumberStack in Figure 4.7 could be made to incorporate exceptions with an Ada program through the following steps:

(a) by adding, in the package specification part, the declarations

StackFull, StackEmpty : **exception**;

immediately after the line

type stack **is private**;

(b) by replacing, in the package body, the statements

put("The stack is empty");
and
put("The stack is full");

with statements

raise StackEmpty;
and
raise StackFull;

respectively,

(c) By adding, at the end of an Ada program (which has imported the package NumberStack), the following pieces of exception code which are introduced by the keywords **when** and are executed only if their respective exceptions have occurred:

```
exception
   when StackFull =>
      --do something
   when StackEmpty =>
      --do something else
```

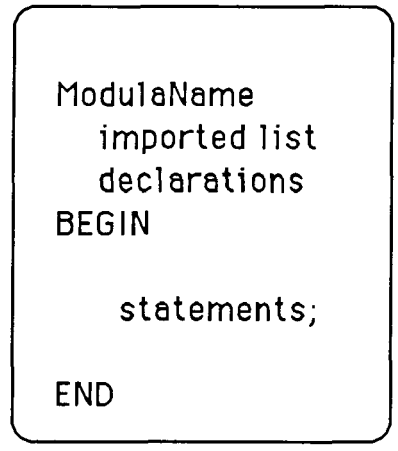
So, the overall effect of the exception handling process is that, whenever an exception is detected by the package NumberStack, the appropriate exception is raised and then control is passed to the corresponding handler in the Ada program. By removing the exception handling routine to a separate part of the program, the remaining part of the program can be left to deal with **normal** processing only. Although in principle, exception handling in Ada does not offer any new programming capability, this mechanism gives programmers the opportunity to deal with exceptions under their control.

However, in situation where an application procedure uses a package that includes exceptions but fails to provide the appropriate exception handlers, the Ada system will first look for an exception handler in the sub-program which invoked the application procedure. If no handler is found, a search will be made in the next enclosing sub-program. Eventually, the system will look in the main program and if a handler is still not found, the default actions have to be carried out.

Finally, the exception handling mechanism described above is applicable to all Ada sub-programs including generic packages.

4.5.2. Modula-2

Modula-2 [53] is a direct-descendant of Pascal and it was designed and developed by Niklaus Wirth who also invented Pascal. In Modula-2, a program is treated as a collection of **modules**. The module concept has much in common with **package** in Ada. The following diagram shows the general structure of a module in the langauge Modula-2:



Currently, there is no official standard in Modula-2, although one is being prepared. The module sample given below conforms to the requirements of the MacMeth computer (version 2.3, 1987), a system written for the Apple Macintosh computers, with all the keywords in upper cases.

4.5.2.1. Encapsulation in Modula-2

For the purpose of information hiding, a module is split into twoparts called a **definition module** and an **implementation module.** These are the direct equivalents of package specification and package body in Ada. Definition modules and implementation modules can be compiled separately provided that the former is compiled first. Figure 4.9 shows an example of a stack implemented as a linked-list.

```
DEFINITION MODULE stackops;

    TYPE itemtype=integer;
    TYPE stack;

    PROCEDURE CreateStack(VAR s: stack);
    PROCEDURE Isemptystack(s: stack): boolean;
    PROCEDURE Top(s: stack): itemtype;
    PROCEDURE Pop(VAR s: stack);
    PROCEDURE Push(item: itemtype; VAR s: stack);

END stackops;
```

(a)

```
IMPLEMENTATION MODULE stackops;

    FROM InOut IMPORT WriteString;
    FROM System IMPORT Allocate, Deallocate;

    TYPE
        Link=POINTER TO StackRecord;
        stack=Link;
        StackRecord=RECORD
                        item: itemtype;
                        previous: Link;
                    END RECORD;

    PROCEDURE CreateStack(VAR s: stack);
    BEGIN
        Allocate(s,size(StackRecord));
        s:=NIL;
    END CreateStack;

    PROCEDURE Isemptystack(s: stack): boolean;
    BEGIN
      RETURN(s=NIL);
    END Isemptystack;

    PROCEDURE Top(s: stack): itemtype;
    BEGIN
        IF Isemptystack(s)
        THEN
            WriteString('The stack is empty');
        ELSE
            RETURN(s^.item);
        END
    END Top;
```

```
    PROCEDURE Pop(VAR s: stack);
    VAR temp: Link;
    BEGIN
        IF Isemptystack(s)
        THEN
            WriteString('The stack is empty');
        ELSE
            temp:=s;
            s:=s^.previous;
            Deallocate(temp);
        END
    END Pop;

    PROCEDURE Push(item: itemtype; VAR s: stack);
    VAR p: Link;
    BEGIN
        Allocate(p,size(StackRecord));
        p^.item:=item;
        p^.previous:=s;
        s:= p;
    END Push;
END stackops;
```

(b)

Figure 4.9 A Modula-2 stack based on a linked-list
(a) definition module
(b) implementation module

Several points are worth mentioning from this example:

(a) Modula-2 uses Pascal's parameter mechanism. Parameters are passed by value or by reference and in the latter case the paramater is preceded by reserved word VAR.

(b) Functions are also termed as PROCEDURE in Modula-2.

(c) To make use of objects defined in other modules, they are brought into existence by the IMPORT statement. So the statement

FROM *InOut* IMPORT *WriteString*;

means to import the routine *WriteString* from the standard library *InOut*.

(d) All objects defined in the defintion module are exportable and can then be used in the implementation module.

(e) Pre-defined procedures *Allocate* and *Deallocate* are equivalent to **new** and **dispose** in Pascal respectively. Procedure *size* returns the amount of storage space needed for a new object.

(f) Modula-2 provides full encapsulation as illustrated by the type definition of object stack which is hidden totally from the specification part of the ADT. This technique is called **opaque export**. However, the objects are restricted to being of type pointer or to subrange of standard types.

4.5.2.2. The provision of ADTs in Modula-2

This can be done simply by placing the names of the required resources (or objects) appeared in the definition module of the ADT in an IMPORT statement immediately following the name of the main module as shown below:

```
Module Reverse;
From stackops IMPORT stack, itemtype, CreateStack, Push, Pop, Top;
```

Although Modula-2 supports total hiding of the implementation of an ADT, it does not provide any generic type facilities as in Ada which is a big drawback.

4.5.3. CLU

The language CLU [54] was developed at MIT by B. Liskov and her team. One of the initiatives for the design of CLU was to provide programmers with a tool that would enhance their effectiveness in constructing programs of high quality i.e. easy to understand, construct, modify and maintain. This goal is achieved by using *abstraction*. There are three kinds of abstractions in CLU - **procedural**, **control** and **data abstractions**. The only concern in this thesis is data abstractions which will be viewed as a new user-defined type. A data abstraction is implemented in one single program module by means of a **cluster**. Actually, the name CLU comes from the first three letters of the word CLUster.

4.5.3.1. CLU clusters

A CLU cluster contains the following three components:

(a) A header which gives the name of the type being implemented and the name(s) of the associated operation(s).

(b) A definition of the representation chosen in the type being implemented.

(c) Implementation(s) of the primitive operation(s) of the type. Additional routines (*CLU procedures* or *CLU iterators*) can also be included in the cluster but only those named in the header can be called from the outside world.

Figure 4.10 shows the features of a CLU cluster.

```
AbstypeName=cluster is          % list of names of the externally
                                  visible operations go here.

    rep= % definition of the representation used goes here.
     % Implementations of the operations
     % plus some useful local routines if desired.

end AbstypeName
```

N.B. % is just the comment symbol in CLU

Figure 4.10 CLU cluster template.

In order to illustrate how data abstraction is achieved in CLU, the stack example is used again as in Figure 4.11:

```
stack=cluster is CreateStack, Isemptystack,
          Topitem, Pop, Push              % heading of the cluster

   rep=array [int]          % definition of the internal representation

      % Implementation of all the stack operations are given below:
      % N.B.  All the pre-defined CLU routines are in italic style.

      CreateStack=proc () returns (cvt)
         return(rep$create(0))                    % create a new array with
       end CreateStack                            % low bound 0

        Isemptystack=proc (s:cvt) returns(bool)
          return(rep$low(s)=0)                    % low determines the
       end Isemptystack                           % lowest index of an array

        Topitem=proc (s:cvt) returns (int)
         if rep$low(s)=0 then
           signal failure("The stack is empty")   % failure is an exception
         end                                      % routine which serves as
         return(rep$top(s))                       % general error message
       end Topitem                                % which will terminate
                                                  % enclosing procedure
                                                  % after its execution.

       Pop=proc (s:cvt)
          if rep$low(s)=0 then
           signal failure("The stack is empty")
          end
           rep$remh(s)                            % remh removes the last
       end Pop                                    % element of an array.



      Push=proc (s:cvt, i:int)
         rep$addh(s,i)                              % addh adds an additional
      end Push                                      % element to the end of
                                                    % an array.
      choose=proc (s:cvt) returns (int)
         return (rep$bottom(s))                   % bottom returns s[low(s)]
       end choose

  end stack
```

Figure 4.11 A cluster called stack

There are several points worth noting about this cluster example:

(a) The line **rep**= array [int] serves the following functions:

It informs the CLU compiler that inside the stack cluster the representation is an array of integers and since it is an equate (a constant), it permits the reserved word **rep** to be used as an abbreviation for this array type throughout the body of the cluster.

(b) In any cluster, two types are always under discussion: the new abstract type, eg. stack, which is being implemented, and the representation type (or **rep** for short), that is the array type in (a). Inside the cluster, these two types must be visible. Furthermore, it must be possible to go back and forth betweeen them since users are only allowed to call the operations of the abstract type which are implemented in terms of the **rep** type. For instance, operation Topitem of cluster stack which receives an argument of type stack, but to implement Topitem, it must be able to make use of the array that represents stack.

(c) In order to achieve the objectives mentioned in (b), CLU provides two special operations called **up** and **down.** **up** takes a **rep** object as argument and produces an abstract object as the result; **down** performs the reverse transformation. Each cluster has its own version of **up** and **down** to accompany itself only. These two operations are defined automatically by the CLU compiler so that the actual representation is not visible to the users. For instance, the CLU compiler will probably provide the **up** and **down** operations for the cluster stack with the following headers:

up=proc (a:array [int]) **returns** (stack)
down=proc (s:stack) **returns** (array [int])

Operations **up** and **down** can only be used in clusters but independent of the location within the cluster. However, **up** and **down** are most often used in two circumstances. First, when an operation takes an abstract

object as an argument, it often uses **down** to convert that object to the **rep** type; second, when an operation returns a newly-created abstract object, it often uses **up** (just before returning) to convert a **rep** object to the abstract type.

(d) Although the **up** and **down** operations are very useful in the two cases described at the end of (c), CLU provides a special syntax - **cvt** (change of viewpoint) to simplify the convertions. The keyword **cvt** may be used as the type of an argument or the type of the result in the header of a cluster's operation. In the former , it indicates that the actual parameter is of the abstract type but the formal parameter is of the **rep** type, so **down** should be applied implicitly to the actual parameter immediately after the call and the resulting **rep** object should be assigned to the formal parameter. For example, in operation Push of cluster stack, **down** is called on the abstract stack object s, passed as an argument, but inside Push the type of s is array [int]. On the other hand, when **cvt** is used as the type of a result, it indicates that the result object is of the abstract type but the object being returned is of **rep** type, so that **up** should be applied implicitly this time to the returned object just before it is returned. Consider procedure CreateStack of stack, it returns an abstract object, that is a stack, to its caller but the **return** statement is apparently returning an array. As **cvt** is used in the header of this routine, **up** will be applied implicitly just before the object is being returned.

(e) It may have noticed that the two keywords **returns** and **return** appear frequently in cluster stack. **returns** is often found in the header of a routine while **return** is inside the body of a routine. Actually, they have different significant effects on the routine. **returns** is a CLU specific built-in predicate which asserts the fact that the routine will return in the normal way; that is, it does not abort, raise a signal or run forever. However, **return** is a statement to terminate execution of the containing procedure or iterator. So they must not be mixed.

(f) The position of a (local or external) procedure inside a cluster is not important, i.e. procedures can be referenced even before they are defined. Thus, it is only important in run-time but not in compile-time. To illustrate these points, consider a stack which provides an extra external procedure called member whose function is to check whether or not an item has already existed in the stack. If the answer is positive, the item will not be pushed onto the stack, otherwise it will be pushed. In conjunction with member is a local procedure named getindex. Since there is an additional routine, the new heading of cluster stack will be:

```
stack=CreateStack, Isemptystack, member, Topitem, Pop, Push
```

and procedures member and getindex are defined as:

```
member=proc(s:cvt, i:int) returns(bool)
    return(getindex(s,i)<=rep$high(s))
end member

getindex=proc(s:rep, i:int) returns(int)    % a local procedure of stack
    j:int := rep$low(s)
    while j<=rep$high(s) do
        if i=s[j] then
            return (j)
        end
        j:=j+1
    end
    return(j)
 end getindex
```

Operation member can then be accessed externally as usual or internally by a slight modification of the procedure Push:

```
Push=proc(s:stack, i:int)
    if ~member(s,i) then
        rep$addh(down(s),i)
    end
end Push
```

In this new version of Push, the abstract type stack is used instead of the keyword **cvt**. The reason for this is because Push uses the procedure

member (which may be defined later) that requires a stack argument, therefore **cvt** cannot be used to convert s to an object of the **rep** type. Furthermore, in the call to member, the prefix stack$ is no longer needed as the call is made within its own cluster. Also, since getindex is a local procedure to the cluster stack, no type conversion is needed and hence s can be declared as type **rep** directly.

(g) The final comment on this example is that operations *create*, *new*, *addh*, *top*, *high* etc, are all pre-defined routines in CLU for instances of type array whose size can grow or shrink dynamically i.e. no absolute boundary. However, different types will have different such pre-defined operations (see [54]).

4.5.3.2. Using a CLU cluster

Once a data abstraction has been defined , eg. a stack, it is no different from a built-in type and its objects and operations can be usable in the same way as those of a built-in type. In CLU, variables can be declared of the new type, as in

```
s:stack
```

and objects can be created and manipulated, as in

```
s:= stack$CreateStack()
stack$Push(s,3)
```

User defined types can also be used to represent other user-defined types. For instance, one could implement a hierarchy tree structure with the following type definition:

```
rep=record [item:int, left, right:stack]
```

where the ADT stack is employed to represent each node of the tree.

4.5.3.3. Parameterized cluster

Finally, CLU clusters can be parameterized. Parameterization provides the ability to define a class of related abstractions by means of a single module. However, parameters are restricted to the following types: int, real, bool, char, string, null and type. The type parameter is the most useful one. When a module is parameterized in this case, no knowledge of the actual parameter type is needed. Nevertheless, if the module is related to objects of the parameter type, certain operations must be provided by the actual type. Information about the required operations is described in a **where** clause which is part of the heading of a parameterized module. Considering,

```
set=cluster [t:type] is CreateStack, Isemptystack, Topitem, Push, Pop
    where t has equal: proctype (t,t) returns (bool)
```

defines a generalized set of abstraction. Sets of many different types can be obtained from this cluster, but the **where** clause states that the element type must provide an equal operation. To use the parameterized module, actual values for the parameters must be provided using the general form:

```
module_name [parameter_values]
```

4.5.4. A comparison between Ada, Modula-2 and CLU

Ada, Modula-2 and CLU are classical examples of programming languages that support information hiding as illustrated by those stack examples given previously. The main reason of using stack is partly due to its simplicity and familarity, and partly due to the intention of making a comparison between Ada, Modula-2 and CLU with regard to their information hiding capability as follows:

(a) **Transparency** :

Transparency is an essential requirement for any language to support information hiding as it shields users from misusing and yet enables easy maintenance of abstract objects. CLU and Modula-2 provides **total** transparency through clusters and opaque exports respectively; whereas Ada's private type mechanism only supports transparency **partially.**

(b) **Re-usability** :

All three languages encourage programmers to view their tasks as the construction of new libraries using, as far as possible, the facilities already available in existing libraries.

(c) **Generic support** :

Ada and CLU supports generic facility using generic packages and parameterized clusters respectively. On the contrary, Modula-2 does not possess any generic facility.

(d) **Ease of implementation** :

It is apparently easier to implement the ADT, stack, in CLU than in Ada and Modula-2 which reflects the more efficient data abstraction mechanisms provided by CLU. However, much of the abstraction is done by the language itself such as those standard array routines.

(e) **Program size :**

CLU has used the least amount of codes, compared to Ada and Modula-2, for the implementation of those stack's routines owing to the intensive use of pre-defined library routines.

(f) **Flexibility** : The discussion of this issue will be divided into two parts:

I/O facilities : Ada and Modula-2 allow programmers to select the most appropriate set of I/O routines subject to the requirement of the application via the **with** and **import** statements respectively. In CLU, this kind of flexibility is lost since all the I/O statements have been built internally such as the **failure** statement.

returned function values : There is no restriction on the type of values returned by an Ada or a CLU function, but in Modula-2, functions can only return values of simple types such as integer, character, real etc.

(g) **Separate Compilation:**

Ada's packages and Modula-2's modules have the same general structure. Both have two parts: one containing the definition (specification) of objects that can be used by an application program, the other containing the body (implementation) of those objects. Hence, the specification part is physically separated from the implementation part which enhances the use of separate compilation. Although CLU's clusters cannot be compiled separately, clusters can be parameterized to define a class of related abstractions using a single module which is an important feature of object oriented programming as discussed shortly.

(h) **Error handling :**

Ada and CLU have explicit routines to deal with constraint violations such as Ada's **raise** statements and CLU's **signal** statements. On the other

hand, since Modula-2 does not provide this facility, the application programmers have to be forced to accept the system's default action (usually to abort the whole process).

4.6. Object-oriented programming

The idea of ADTs is very useful in sub-dividing a big task into pieces as stated in section 4.1. When designing a large complicated program, everythings can be thought as an ADT or an *object* and the designer's task is to identify and define these objects. This leads to the technique of **object-oriented programming**. Problems solved by this technique is often easier than the traditional techniques such as *top-down* and *bottom-up*. However, as Thomas [51] mentioned that a combination of the two techniques: object-oriented programming and top-down will provide programmers the best way to tackle a really difficult program. The programmer can use the former to identify objects and their operations to develop the structure of the program as a collection of modules. Then the latter is used to evolve the more complex operations of the objects or even perhaps the main program body.

Thus, object-oriented programming design leads to easier maintainable programs for two main reasons. First of all, the procedures are usually short and readable as they perform only single and well-defined tasks. Secondly, it is easier to locate and modify an operation when it is confined to a single procedure. Recently, object-oriented programming has turned out to be a very efficient tool because it allows programmers to build new software on top of some existing objects which are reliable and usually well-documented.

From the above discussion, it can be realized that object-oriented programming is primarily a system building tool which puts **re-usability** at the centre of the software development process, making re-usability the usual way that new components are built. Since object-oriented programming is so useful, a lot of research efforts have been made to implant this concept on existing conventional programming languages. Apart from retaining the efficiency and compatibility of the base language,

the new language will also provide the re-usability power of an object-oriented programming language. Two typical examples of this kind of language are Simula-67 and Smalltalk-80. The main theme for the rest of this chapter is to investigate the object oriented programming features of these two languages.

4.6.1. Object-oriented programming languages

The programming language Simula-67 [55] is an early ancestor of object-oriented programming. It comes by this lineage due to its introduction of the **class** concept. In object oriented language terminology, a class is viewed as a generic ADT. The main role of a class with respect to object-oriented programming is its ability to create *objects* which are the means for communication between them. However, the real founder of object-oriented programming is the modern language Smalltalk-80 [18], developed at Xerox Palo Alto Research Centre, U.S.A. In fact, the term object-oriented programming arose from the enormous development of this language. Nevertheless, re-usability is the key issue in any object-oriented programming language. To enhance re-usability, the idea of **inheritance** was emerged.

In the following sub-sections, the basic principles of inheritance will be described, followed by an investigation of the inheritance power in Simula-67 and Smalltalk-80. Since object-oriented programming has been used widely for interactive and graphical applications, the examples given in these sub-sections will belong to this category.

4.6.1.1. Inheritance

The functions of inheritance in object oriented programming can be summarized as follows:

(a) It is a tool for organizing, building and using reusable classes.

(b) Inheritance links concepts (ADTs) together so that as a higher level

concept changes, the change will be automatically applied throughout the rest of the system.

(c) Inheritance also provides enormous simplification by relating objects as closely as possible to one another. As a result, this will reduce the total number of objects that must be specified and stored. In other words, inheritance has a two-fold effect, reducing code bulk by reducing the need to re-develop common functionality, and reducing surface area by enhancing consistency.

So, it can be seen that without inheritance, each class would be a free-standing unit developed from scratch, which will in turn increase the possibility of inconsistency.

4.6.1.2. Simula-67 class concepts

Simula-67 was a language developed in 1967 at the Norwegian Computing Centre in Oslo. The initial motivation for the development of this language was to provide an environment, based on the language Algol-60, for producing software models to simulate real processes such as a queueing process or an industrial process and so on. By running the model, the performance of the real processes can be assessed. The designers of Simula-67 introduced a new programming construct called **class**. A class (see Figure 4.12 below) consists of some variables, a collection of procedures and a piece of code (the class body) which is executed whenever a new object of the class comes into existence.

```
class heading;
class variable(s);
class procedure heading(s);
class body;
```

Figure 4.12 Components of a Simula-67 class

Each object of the class has its own copy of the variables and so has the ability to represent a dynamic state. The objects continue to exist until they can no longer be referred to by any means. For instance, the following class declaration expresses the concept of complex numbers:

```
class complex(x,y); real x,y;
begin real r,theta;
      r:=sqrt(x**2+y**2);
      theta:=arctan(y,x);
end complex;
```

In Simula-67, a block is a pattern of text:

```
begin declarations;
  statements;
end;
```

An **instance** is a representation of this pattern of data and actions in the computer memory. It comes to existence only when its call is executed. In Simula-67, the source and the object programs are equivalent. The purpose of a block (or procedures) is, for each activation of it, to perform a sequence of operations according to a given statement pattern, thereby producing a set of results. To achieve its purpose, each instance of the block may set up a suitable data structure for its own use. After the completion of its actions, the whole block instance is deleted including the local data structures; only the results remain as recorded in non-local variables or in the state of input or output devices etc. Thus, the apparent symmetry between the declaration part and statement part of a block (as shown above) is not matched by the behaviour of the kind of block instance. The declarations, at least those variables play a secondary role since they have strictly local significance.

A group of objects having the same heading, layout of variables and action pattern are said to be belonged to the same class. The class name is a generic name, describing what kind of object it contains. Objects belonging to the same

class are collectively described by a class-declaration. Class-heading and class-body are very similar to a procedure-declaration. An object may perform operation just as any other block instances. At the same time, it is an entity which has **attributes**. The attributes of an object of a given class are instances of the quantities declared locally to the class body and instances of the parameters listed and specified in the class-heading. The attributes of an object are accessible from outside the object by special language mechanism, eg. the dot-notation. For example, the following piece of program declares C to be a complex number and makes C refer to the object created by standard routine **new** in the second statement:

```
ref(complex) C;
C:-new complex(3.0,4.0);
```

The attributes of object C (x, y, r and theta) can then be accessed via the usual dot notation and so the statements print(C.x); print(C.y); print(C.r) and print(C.theta) will print the values 3.0, 4.0, 5.0 and 0.927 respectively.

Simula-67 also introduced the notion of **super-class** and **sub-class** as a way of representing the property of *inheritance*. For instance, if there is a class named list_element which is defined as :

```
class list_element;
begin
    ref(list_element) next;
    next:-none;

    ! class procedures insert, delete, push etc. which
    ! are operating on the class list_element go here.
end list_element;
```

then a list of complex numbers may be created by prefixing the name list_element to the declaration of class complex as follows:

```
list_element class complex (x,y); real x,y;
begin
    ! the same code as class complex;
end complex;
```

Here class list_element is the super-class of class complex or complex is the sub-class of list_element. The effect of the super- and sub- classes relationship is that, all the actions defined in the super-class are performed prior to those of the sub-class, whenever a new object of the sub-class is generated. In other words, the sub-class object has inherited all the properties of its super-class but not vice versa.

Although Simula-67 supports the definition of abstract data objects by representing the abstract operations of the type as **procedures** within a class, the dot notation provides access not only to the procedures of a class but also to all the local attributes like variables. Hence, it is possible to access a Simula-67 object in a way which violates the properties of the ADT. Nevertheless, the Simula-67's class concept was among the first programming languages to provide a general mechanism for programmers to define their own types and the representation of the hierarchy of type inheritance.

4.6.1.3. Smalltalk-80

Smalltalk-80 was a successor of the language Smalltalk-76. It was intended as a language for developing user interfaces on personal computers and has lead to the ideas behind the notion of *desktop*. However, it was felt that no programming language was really suitable for desktop applications. As a result, new programming approach was developed as in Smalltalk-80.

To be precise, Smalltalk-80 was a graphical interactive programming environment rather than a language. It was designed such that every components of the system that is accessible to the users can be presented in a meaningful way for **observation** and **manipulation**. With the aid of suitable hardware, the user can select information on the screen usually by means of an *icon* , and then invoke the appropriate actions to interact with that information. Although Smalltalk-80 was made up of many components, it was based on only a small

number of concepts [18, pp. 5-89] and they can be defined by five terms :

object, message, class, instance and **method**

Key concepts :

All the components of Smalltalk-80 (numbers, strings, programs, compilers etc) are represented as an object which is simply an ADT consisted of some private memory and a set of operations.

In Smalltalk-80, all the actions are performed via the technique of **message passsing**. A **message** is a request for an object to carry out one of its operations, but however, the message does not specify how that operation should be done. The latter would be performed by a **receiver**, the object to which the message was sent. The partnership between a message and the corresponding receiver is that: a message specifies the **type** of operation desired together with the required data; the receiver determines how to accomplish the operation. The kind of messages to which an object can respond is called its **interface** with the rest of the system. The only way to interact with an object is through its interface. Furthermore, in order to protect the internal implementation of an object, the private memory of it can only be manipulated by its own operations and these operations can only be invoked by the appropriate messages.

To make system management easier and better re-usability, a special kind of object called **class** (similar in concept to a Simula class) exist in Smalltalk-80. A class describes the implementation of a set of similar objects. The individual objects described by a class is called its **instance**. In other words, a class describes the form of its instances' private memories and how the instances carry out their operations. The instance of a class are similar in both their public and private properties. An object's public properties are the messages that make up its interface. All instances of a class have the same message interface since they represent the same kind of components. An object's private properties are a set of **instance variables** that make up its private memory and a set of **methods** (routines) that describe how to carry out its operations. The instances of a class

use the same set of methods to describe their operations. Each class has a name that describes the type of component its instances represent. Each instance variable in an object's private memory refers to one object, called its **value**. Each method in a class specifies how to perform the operation requested by a particular type of message. When that type of message is sent to any instance of the class, the method is executed.

Nevertheless, the best way to understand all the concepts described above is to present an application program written in Smalltalk-80 as shown in the next sub-section.

4.6.1.4. A Smalltalk-80 example

Smalltalk-80 often considers programming as instructing real objects to perform a sequence of actions, eg. drawing figures on the computer screen using a computerised pen (often termed as the **light pen**). So the pen is an object which can perform certain tasks. But as in real life, figures can take different forms and so does the pen. Therefore, pen is a generic concept. This leads us to define an object to be a specific instance of a generic description. As an example, Figure 4.13 shows how a square can be drawn with Smalltalk-80.

```
class name                          square
superclass                          apen
instance variable name              x
                                    y
                                    size
instance methods

    accessing

        create
            self new

        draw: ink

            super colour: ink
            super moveto: (ord: x     vert: y)
            super drawlineto: (ord: x+size      vert: y)
            super drawlineto: (ord: x+size      vert: y+size)
            super drawlineto: (ord: x      vert: y+size)
            super drawlineto: (ord: x      vert: y)
            super moveto: (ord: 0     vert:0)


        erase
            draw: background


        stretch: amount
            self erase
            size<- size+amount
            self draw
```

Figure 4.13 A class definition of drawing a square in Smalltalk-80.

Several points needed to be noted from the above example:

(a) Whenever a new instance of class square is created, the procedure **create** must be invoked with three parameters, eg. square create:(x:250 y:375 size:20), and the function of **self new** is to create a new initialized instance of the class itself.

(b) **Instance variables** are local variables and each instance of class square will have its own private set of instance variables.

(c) **Instance methods** are those procedures that describe the behaviour of the instances when they receive messages. They can be shared between all instances of class square.

(d) The word **accessing,** appeared just before the four procedures, indicates that all the procedures following it is accessible by users.

(e) The procedures **moveto** and **drawlineto** are routines that have been defined in the superclass **apen** as indicated by the word **super** in routine **draw**. This means that class square can inherit all resources of apen.

(f) The pseudo-variable **self** referred to the receiver of a message because in general there can be several squares in existence at any time, and it is important to identify them individually.

(g) The colour **background** is the colour of a blank screen and **ink** is an optional colour.

From this example, one can realize that object-oriented programming languages, such as Smalltalk-80, provide a high degree of information hiding. Users are only allowed to know about the messages that can be sent to an object via those defined operations and about what additional information may be required i.e. the nature of the parameters of the operations. No visibility of private data or of the implementations of the operations. Thus, objects are the sole inhabitant of the universe. Moreover, objects are treated uniformly. For instance, they all have inherent processing facility and they are all referred in the same way. They also all communicate via the technique of message passing. Also, though integers are provided in the language, they are not considered primitively or specially in any way. User defined objects are on the same status level as system defined objects. Another interesting feature is that an object cannot be opened up; the inside of it is hidden. Of course, if the object wants to permit its insides to be examined, it can

be done. The only restriction in Smalltalk-80 is that messages are the only way to communicate between objects, so they must be very versatile. A message may be parameterized to include one or more objects' names along with its message. Then the receiver object responses to the message with a reply. The message may initiate computation and the reply confirms that the activity has been successfully accomplished. At this point, one may think that message passing or sending sounds like procedure invokations. This may be right in some sense but there is a difference. In procedure calls, the caller is in control, waiting for the procedure to return. Also the caller and the receiver of the call usually share the procedure interface and may be some global variables' data spaces. In object-oriented programming, once a message is sent, the sender trusts the receiver to accomplish its goal. Thus the sender relinquishes control to the receiver.

Among those features of Smalltalk-80 discussed so far, the most important one is the mechanism of inheritance. It can be viewed as a way to define some useful constructs in a central place and then automatically broadcasting that construct to all places where it is required as indicated by the keyword **superclass**. The advantage of this approach is that new functionality is no longer developed by coding from scratch, but by inheriting some superclasses and describing how the new one differs. For instance, suppose an object called rectangle for which operations draw, erase and stretch have been defined. it would be simpler to define the class square as a rectangle whose sides are equal. Square can then inherits all the operations of rectangle. This means that no new procedures is needed except changing the values of some local variables in Figure 4.13. So inheritance has advanced the cause of **re-usability**. Note also that Smalltalk-80 supports only single inheritance. That is, an object can only be an instance of one class.

4.7. Conclusions

Abstract data types have been used in almost all stages of software development, particularly in the specification, design and implementation stages of the software life cycle. This chapter has discussed the important aspects of ADTs that make them so useful.

Basically, an ADT is composed of a set of objects and a set of operations that manipulate the objects, and the behaviour of the ADT can only be noticed by observing the results of applying the operations. This concept can be used to separate the level of concerns of large and complicated software programs by defining every resources (objects) as an ADT. As a result, it can reduce the software production costs (which is an expensive commodity) and to produce reliable software products. Two methods for the specification of ADTs, the axiomatic approach and the constructive approach, are described. The former specifies an ADT by defining its properties as a set of axioms (rules) while the latter builds an ADT from objects that are already well defined. Also, issues for the implementation of ADTs such as information hiding, error detection are discussed.

In order to illustrate the power of ADTs, three modern programming languages, Ada, Modula-2 and CLU are chosen as examples. Each of them has used a different method to achieve information hiding. They are packages, modules and clusters respectively. A comparison between them with respect to the issue of information hiding is also given.

The idea of abstract data types has lead to a new programming technique known as object oriented programming. This technique was originated from the class concept of Simula-67 in which programmers are allowed to define complex objects (ADTs), made up of attributes that have already been defined. These attributes may be of the standard Simula-67 types or may be reference types defined by the users. The main objective of this approach is **reusable software**. That is, a new piece of software is created from an existing piece of similar software. In this sense, the new software derives or inherits its properties from exsting software. The object oriented programming language Smalltalk-80 has greatly enhanced the cause of software re-usability as

illustrated by the square example in section 4.6.1.4. To be precise, object oriented programming is primarily a system building tool which arose from the awareness that ambitious software systems are generally too expensive, of sufficient quality and hard to schedule reliably. Object-oriented systems should prevail well into the future with the exponential improvement in hardware capability.

Chapter Five -- Type theory and data persistence in high level languages

5.1. History of types in programming languages

In the early stage of programming, numerical computation was the only concern and therefore all values were viewed as having a single arithmetic type. However, with the advent of high-level programming languages such as Fortran in 1950s, it was found convenient to distinguish between integers and reals (represented by floating point numbers) for two reasons:

(a) In computer memory, all data are represented as a string of bits of fixed size: characters, integers, reals, records and arrays etc. Thus, there is no way of differentiating what is being represented by a piece of raw memory. However, integers can be represented in an exact form compared to reals due to the rounding and truncation errors of the latter. Therefore, integer computations and real computations must be considered separately.

(b) Also, use of integers for iteration and array computations was logically different from the use of floating point numbers for numerical computations. For example, elements of an array are totally distinct from each other, so integers must be used as the index.

This provided the initial indication of requiring a type system. The best way to organize such a system is to classify objects in terms of the purposes for which they are used. Some of these typed languages are described briefly in the following paragraphs.

Fortran [56] used the first letter of a variable to distinguish between integers and reals; all identifiers starting with letter I, J, K, L, M or N are integers whereas the others are reals. Later, Algol-60 [57] introduced identifier declarations for

integer, real and boolean variables, eg.

real x,y; **integer** z;

The language Algol-60 was a pioneer in the notion of type and associated requirements for compile-time type checking. The block structure requirements of Algol-60 allowed not only the type but also the scope (visibility) of variables to be checked at compile-time. The ideas of type checking will be discussed shortly.

In 1960s, the concept of types was developed further. The Algol-60 type notion was extended to richer classes of values. There were some new typed languages during this period, eg. PL/I [58], Pascal [59], Algol-68 [60] and Simula-67 [55] etc. PL/I attempted to combine features of Fortran [56], Algol-60 [57], COBOL [61] and LISP [62]. Its types include typed arrays, records and pointers. However, it has certain type loopholes especially in pointer types as illustrated by the following PL/I program (all keywords are in upper cases) :

```
DCL 1  structA     STATIC,
DCL    06   x      char(6),
DCL    06   y      char(20);
DCL 1  structB     AUTO,
DCL    06   a      char(6),
DCL    06   b      char(21);
DCL    (pt1,pt2)   POINTER;
          :
          :
pt1=addr(structA);
pt2=addr(structB);
          :
          :
pt1=pt2;
          :
          :
END;
```

Several points are worth noting in this example :

(a) The number to the left of each identfiier (i.e. a, b, x and y) is called a **level number** which is used to indicate relative levels of names. All level numbers must be a decimal number and they are usually appeared in a declaration (the DCL statements). By default, level number 1 is always assumed which also signifies the beginning of a new structure (eg. structA, structB).

(b) The two keywords STATIC and AUTO refer to the storage class attributes, **static** and **automatic**, in PL/I respectively. Storage is allocated permanently to variables having storage class **static** throughout the execution of a PL/I program. However, for variables having storage class **automatic**, storage is allocated only when the block in which the variable declared is active and then it will be released after the block has terminated.

(c) Pointer variables in PL/I are declared by the keyword POINTER (or PTR for short). These variables can be used to point to the base address of any structure as in the statement,

pt1=*addr*(structA)

where *addr* is a built-in function which locates the starting memory address of a variable. Also, a pointer variable can be assigned to any other pointer variable. Hence, PL/I does not offer much type-checking facility on pointers except that a pointer cannot be assigned to a non-pointer variable. However, this is potentially very dangerous as PL/I's pointers are merely memory addresses and therefore it is the programmer's responsibility to ensure that he or she is accessing the right parts of the memory. Consider the assignment statement:

pt1=pt2

in the above example, pt1 is pointed to the base address of structA which occupies 26 number of bytes whereas pt2 is pointed to the base address of structB that occupies 27 number of bytes in memory. In other words, some previously

defined memory values have been overwritten without informing the system. This is the reason that Pascal and Ada have to associate each pointer variable with objects of one type only.

In addition to arrays, records and pointers, the language Pascal [59] provides even more types: sets, variant records as well as user-defined types. Unfortunately, Pascal also contains some type loopholes; for instance, it includes array bounds as part of the array type specification which makes procedures that operate uniformly on arrays of different dimensions impossible to be defined. Also, Pascal did not require the full type (names and definitions) of procedures/functions passed as parameters to be specified. For example, if there is a procedure defined as:

```
procedure test1(var a:integer);
begin
      a:=a+1;
end;
```

it can be passed to any procedure with heading:

```
procedure try(procedure t(var x:integer));
```

but so does procedure test2:

```
procedure test2(var a:integer);
begin
      a:=a-1;
end;
```

However, the results after the invokations of these two procedures are completely different. Moreover, Pascal allows the tag field of **variant records** to be manipulated independently.

Consider the following type definitions:

```
itemclasses=(Book,Recording);
Libraryitems=record
                Title: packed array [1..30] of char;
                Author: packed array [1..16] of char;
                case class:itemclasses of
                    Book: (Edition:1..50;
                           Year: -200..1999);
                    Recording: (Performer: packed array [1..20] of char)
            end;
```

The usage of the tag field of the variant record, class:itemclasses, is absolutely free to the user. If the current variant of class is Book but the user thinks it is Recording, he or she will then try to access the field Performer which does not exist. Serious errors will occur that may not be detected easily in the later stage of a program. Finally, Pascal does not define type equivalence either, so the question of when two type expressions denote the same type is implementation dependent.

Algol-68 [60] has a more rigorous notion of types than Pascal, with a well-defined notion of **type equivalence** which enables a compiler to check the type of a variable before performing any operations on it. Generally, there are two ways to determine whether or not variables belong to the same type. They are known as **name equivalence** and **structural equivalence**. In the former, two variables are considered to be of the same type only if they are declared using the same type identifier. For instance, in the following piece of Pascal program,

```
type  T=array [1..10] of integer;
var   C : array [1..10] of integer;
      D : T;
      E : T;
```

variables D, E are of the same type but variable C is not.

On the other hand, in structural equivalence (the one used by Algol-68), two variables are considered to be of the same type whenever they have components of the same type structure. Using this definition, all the variables in the previous Pascal program will have the same type. Unfortunately, structural equivalence causes a logical problem as demonstrated in the following declarations,

```
var K : (Male,Female);
    I : (Female, Male);
```

it is difficult to decide whether or not variables K and L are of the same type without the help of a sophisticated compiler.

Furthermore, the notion of type, called **mode** in Algol-68, is extended to include procedures as *first class values*. Primitive Algol-68 modes are **int**, **real**, **char**, **bool**, **string**, **bits**, **bytes**, **format** and **file**, whereas **mode constructors** include **array**, **struct**, **proc**, **union** and **ref** for constructing array types, record types, procedure types, union types (variant records) and pointer types respectively. Algol-68 has well-defined rules for coercion, using dereferencing, deproceduring, widening, rowing, uniting and voiding to transform values to the types required for further computations as described in [60]. Algol-68 also provides type-checking facilities but those algorithms are so complex that it will be a very time-consuming task for users to check them. Thus, later languages such as Ada had a simpler notion of type equivalence and type checking with severely restricted coercion.

In chapter four, the langauge Simula-67 has been addressed. It was the first object-oriented programming language and its notion of types includes classes whose instances may be assigned as values of class-valued variables and may persist between the execution of the procedures they contain. Procedures and data declarations of a class constitute its interface and are accessible to users. Subclasses inherit declared entities in the interface of superclasses and may define additional operations and data that specialize the behaviour of the subclass. Instances of a class like ADTs in having a declarative interface and a state that persists between invokations of operations, but lack the information hiding power of ADTs.

Modula-2 was another language mentioned in chapter four which was the first widespread language to use modularization as a major structuring principle as well as the concept of ADTs. Typed interfaces specify the types and operations available in a module; type pointers and subrange of standard types of an interface can be made **opaque** to achieve data abstraction and also an interface can be specified separately from its implementation as illustrated by the stack example (Figure 4.9). Module interfaces in Modula-2 are similar to class declarations in Simula-67 except for two scope rules:

(a) The scope of visibility of a module's identifiers can be extending by listing identifiers in the module's export list. Then identifiers will be visible in the surrounding scope.

(b) An identifier visibles in the surrounding scope is also visible inside the local procedure. But it is not visible inside a local module unless the identifier is included in the module's import list.

The above two scope rules can be summarized by the following example (all keywords in capital letters):

```
VAR a,b:CARDINAL;
MODULE m;
      IMPORT a;
      EXPORT w,x;
      VAR u,v,w:CARDINAL;
      MODULE n;
          IMPORT u;
          EXPORT x,y;
          VAR x,y,z:CARDINAL;
          (*      u, x, y, z visible here      *)
      END n;
      (*      a, u, v, w, x, y visible here      *)
END m;
(*     a, b, w, x visible here    *)
```

If an identifier is to cross several scope boundaries, it has to be listed in all the IMPORT lists or the module must be EXPORTed as a whole. Extending visibility from an inner module to the outside is achieved by export, while extending from an outer

scope to the inside by import. The rules are completely symmetric. Unlike class instances in Simula-67, module instances are not *first class values*. A linking phase is necessary to interconnect module instances for execution; this phase is specified by the module interfaces but is external to the language.

Finally, ML [63] was one of the newly developed interactive programming languages which has introduced the notion of parametric polymorphism (see section 5.1.3.1) in languages so that variables are instantiated to different types in different contexts. Hence, it is possible to specify type information partially and to write programs based on partially specified types that can be used on all the instances of those types. However, ML omits type declarations; the most general (less specific) type that fit a given situation is then automatically inferred. Another interesting type mechanism in ML is that type specifications omitted by the user may be re-introduced by a technique called **type inference** which will be described shortly. For example, if the user enters 3+4, the system will respond 7:int and inferring that the operands and the value of the expression are of the type int. If the user enters the function declaration **fun** f x=x+1, the system responds f:int->int, defining a function value for f and inferring that it is of type int->int. ML supports type inference not only for traditional types but also for parametric (polymorphism) types, such as the following length function for linked lists. Supposing

```
fun rec length x=if x=nil then 0
                          else  1+length(tail(x));
```

is entered, ML will infer that **length** is a function from lists of arbitrary element type to integers (length: a list->int). If the user enters length[1;2;3] subsequently, ML infers that length is to be specialized to the type int list->int and then applies the specialized function to the list of integers.

5.1.1. Objectives of a type system

Cardelli [64] has suggested a definition for **type** as follows:

type of an object= a set of data + operations allowed on the set of data

and he also stated that the major task of any type system is to prevent data inconsistency in two senses: it avoids embarassing questions about the representations of objects and it prevents inconsistent interactions between objects, which may result in an un-determined situation. However, one could always provide an additional third goal for a type system: to reserve the right amount of storage spaces for objects of the appropriate type. The easiest way to achieve the first two goals is to impose some suitable constraints on objects while the last one is totally machine dependent. Since different objects often have different properties, a unique representation is required for each kind of objects so that the expected operations can be performed more easily on the corresponding instances of these objects. In other words, a type can be thought as a set of covers that protect an underlying untyped representation from arbitrary or un-intended use and at the same time it contraints the way objects may interact with other objects. Violating a type system involves removing the protective covers and operating directly on the underlying representation of objects, i.e. objects could be manipulated in some un-intended ways with potentially disastrous results such as using of an integer as a pointer can cause arbitrary modifications to programs and data.

In order to prevent type violations (usually due to the result of misusing objects of a given type), a static (formatted) type structure is often imposed on programs. One strategy is to associate types with items like constants, operators, variables and function symbols by means of redundant declarations as in Pascal and Ada. Whenever these items appear in the program, they must be defined so that the compiler can check the consistency of the item's definition and its use. Another strategy is that explicit declarations are avoided whenever possible, but a **type inference** system is employed which can infer the type of expressions from local contexts while still establishing consistency as in the language ML. The following sub-sections discuss the basic concepts of these different strategies.

5.1.2. Type checking in programming languages

Generally, the process of checking the type of every expressions in a program is termed as **type-checking**. Type-checking can be done either at compile-time or at execution-time. As mentioned by Cardelli et al [64], languages in which the type of every expressions can be determined at compile-time is said to be **statically typed** (sometimes referred as **static typing**). Languages in which all expressions are *type consistent* are called **strongly typed**. Pascal possesses both these properties. Noted that every statically typed language is also strongly typed but the converse may not be true.

Static typing allows type inconsistencies to be discovered at compile-time and guarantees that executed programs are type consistent. Static typing facilities early detection of type errors and allows greater execution time efficiency. It also enforces a programming discipline on the programmer which makes programs more structured and easy to read. Although static typing has greatly reduced the possibility of type violations, it may lead to a loss of flexibility and *expressive power* by prematurely constraining the behaviour of objects to a particular type. This is the reason that Pascal does not have generic procedures such as Ada's generic package. Type-checking has been greatly enhanced by the notion of **polymorphism** as illustrated below.

5.1.2.1. Polymorphism

Conventional typed languages like Pascal, are based on the idea called **monomorphism**. That is every value, variable and expression can be interpreted as a unique type. Monomorphic programming languages constrasted with polymorphic languages in which some values, variables or expressions may have more than one type. There are two commonly used terms associated with polymorphic languages: **polymorphic functions** and **polymorphic types. The former** are functions whose operands (formal parameters) can have more

than one type whilst the latter are types whose operations are applicable to operands of more than one type. Currently, several types of polymorphism exists as shown in Figure 5.1.

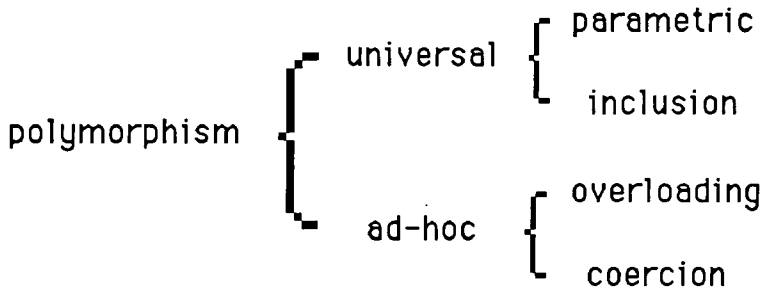


Figure 5.1 Kinds of polymorphism

5.1.2.1.1. Universal Polymorphism

The concept of achieving type uniformity in more than one way is called **universal polymorphism**. Universally polymorphic functions normally work on an infinite number of types (all types having a given common structure) and one can assert with confidence that some values have many types. In terms of implementation, a universal polymorphic function will execute the same code for arguments of any permissible type. Furthermore, universal polymorphism is sub-divided into two kinds: **parametric** and **inclusion** polymorphism. They are the two major ways that a value can have many types.

Parametric polymorphism is so called because the uniformity of the type structure is achieved by the type of a parameter. When a function works uniformly on a range of types (usually exhibiting some common structures), then parametric polymorphism is said to be achieved and the function is referred to as a **generic function**. Generic functions have an implicit or explicit type parameter which determines the type of argument for each

application of them. But these functions are actually doing some kind of work independent of the argument type. This may sound a bit confusing. Consider the case when a stack is being implemented with the three usual operations: push, pop and empty. In Pascal, there is no way to construct procedures for these three operations such that they can be used for a stack of integers or a stack of reals, even though the kind of work involved is the same. Fortunately, this is not the case for all programming languages, for example, the Ada's generic facilities, described in chapter four, allow those stack operations to be written in such a way that the type of the object involved is not defined until the procedures are called. The only additional requirement is a type parameter in the heading of each procedure. However, the Ada's generic procedures/functions are special cases of parametric polymorphism because the actual parameter must be **instantiated** before they can be used. Thus, the actual type values have to be determinable at compile-time and then code is generated for each particular type. This constrasts with a true parametric polymorphic language like ML in which code is only generated once for every procedure. ML also allows procedures to be passed as parameters. Finally, parametric polymorphism can be classified as the purest form of polymorphism because the same object or function can be used uniformly in different type contexts without changes or any kind of run-time tests or special encodings of representations. However, this uniformity of behaviour requires all data to be represented and dealt with uniformly.

The other kind of universal polymorphism is the inclusion polymorphism which is used to model **subtypes** and **type inheritance** in object-oriented programming languages. The concept of inheritance (see chapter four) is very important as it allows relations among types to be specified. Inheritance may also be viewed as a type composition mechanism which permits the properties of one or more types to be re-used in the definition of a new type. Hence, the specification "A inherits B" may be viewed as an abbreviation mechanism that avoids re-defining the attributes of type A in the definition of type B. Nevertheless, inheritance is more than a shorthand since it imposes structure upon a collection of related types. This can reduce the conceptual complexity of a system specification like the hierarchy structure of Smalltalk-80.

Moreover, in inclusion polymorphism, an object is viewed as belonging to many different classes that need not be disjoint, i.e. there may be inclusion of classes. To understand inclusion polymorphism better, the language Simula-67 is re-visited. Simula-67's classes are user-defined types organized in a simple inclusion (or inheritance) hierarchy in which every class has a unique immediate superclass except at the root. Instances of a class may be assigned as values of class-valued variables and may persist between executions of procedures they contain. Procedures and data declarations of a class constitute its *interface* and are accessible to users. Sub-classes inherit declared entities in the interface of superclasses and may define additional operations and data that specialize the behaviour of the sub-classes. Therefore, Simula-67's objects and procedures are polymorphic because an object of a sub-class can appear whenever an object of one of its superclasses is required.

Although instances of a Simula-67 class are similar to data abstractions in having a declarative interface and a state that persists between invokations of procedures, they lack the information-hiding power of data abstractions. This leads to subsequent object-oriented programming languages like Smalltalk-80 which combine the class concept derived from Simula-67 with a stronger notion of information-hiding.

5.1.2.1.2. Ad-hoc polymorphism

In constrast to universal polymorphism, ad-hoc polymorphism is obtained when a function works, or appears to work, on a finite set of types (which may not exhibit a common *structure*) and may behave in (potentially) un-related ways for each type. Also, an ad-hoc polymorphic function may execute different codes for each type of argument. Same as universal polymorphism, two kinds of ad-hoc polymorphism exist called **overloading** and **coercion** (see Figure 5.1).

In overloading, the same variable name can be used to denote different functions. However, the choice of which function is denoted by a particular instance of the name is decided by its context, i.e. overloading is just a convenient syntactic abbreviation. On the other hand, coercion is a semantic operation that is needed to convert an argument to the type expected by a function. Coercions can be provided statically by automatically inserting them between arguments and functions at compile-time or dynamically by run-time testings on the arguments. The distinction between overloading and coercion is sometimes very confusion as illustrated below:

```
2+3
2.0+3
2+3.0
2.0+3.0
```

The ad-hoc polymorphism of the operator + can be interpreted in one of the following ways:

(a) The operator + has four overloaded meanings, one for each of the four combinations of argument types.

(b) The operator + has two overloaded meanings for integer and real addition respectively. Except that if one of the arguments is of type integer and the other is of type real, then the integer argument is coerced to the type real.

(c) The operator + is defined only for real additions and integer arguments are always coerced to the type real.

In this example, one may consider the same expression to exhibit overloading or coercion or both, depending on the implementation.

The definitions of polymorphism described so far is applicable only to languages that have a clear notion of both type and value. However, monomorphic languages such as Pascal and Ada have ways of relaxing strict

monomorphism in the sense that polymorphism is an exception rather than a rule. Real and apparent exceptions to the monomorphic typing rule in conventional languages include:

(a) overloading: Integer constants may have both the types integer and real. Operator + can apply to arguments of both types.

(b) Coercion: For example, in Pascal, an integer value may be used where a real is expected but not the reverse case.

(c) subtyping: Elements of a subrange type also belong to the super-range types.

(d) value sharing: For instance, the value NIL in Pascal is a constant that is shared by all pointer types.

The first two cases are typical examples of ad-hoc polymorphism. Subtyping is an instance of inclusion polymorphism. Every object of a subrange can be used in a supertyped context so the subtype objects can be operated on all the operations of the supertypes. Cardelli et al have pointed out further that value sharing is a special kind of parametric polymorphism because one could think of the symbol NIL as being heavily overloaded but it is open-ended since NIL is a valid element of an infinite collection of types which have not been declared yet. Moreover, all the uses of NIL denote the same value which is not a common case for overloading. One could also think that there is a different NIL for every types but all the NILs have the same representation and can be identified, which is a characteristic of parametric polymorphism.

5.1.3. Type inference algorithms

In conventional typed languages, the compiler assigns a type to every expression. However, in some of these languages, the programmer does not have to specify the type of every expression; type information need only be placed at some critical points in a program and the rest is deduced from the context by the compiler. This deduction process is called **type inference**. Typically, type information is required for local variables, constants, function arguments and function results, then the type of all expressions and statements can be inferred.

Type inference is usually done in a *bottom-up* fashion. Given the types of each variables and/or constants, and the type rules for the ways of combining expressions into bigger expressions provided by the language, it is possible to deduce the type of a particular expression. The language ML introduced a very sophisticated way of performing type inference. Consider the mathematical function f(x)=x+1, the ML type inference algorithm will work bottom-up as follows:

In x+1 above, x would initially have the type **a** where **a** is a new type variable (introduced by the system), then the **int** operator will **force a** to be equivalent to **int**. This instantiation of type variables is done by Robinson's unification algorithm [65], which also takes care of propagating information across all the instances of the same variable, so that incompatible usages of the same variable can be detected. Also, this type inference algorithm is not limited only to polymorphic languages . It could be added to any monomorphic typed language with the restriction that at the end of type checking, all the type variables should disappear. However, expressions like fun x=x would be ambigious, therefore it is better to write fun (x:int)=x to disambiguate them. Nevertheless, the ML type inference algorithm is the best known one recently.

Finally, it is worth noting that type inference will be reduced to type-checking when there is so much type information in a program that the type inference task becomes trivial. More precisely, type-checking is the case when all the type

expressions involved in checking a program have explicitly contained in the program text. In other words, there is no need to generate new type expressions during compilation and all have to do is matching existing type expressions.

5.2. Persistent data in programming languages

5.2.1. Background

Databases and programming languages have developed nearly independently over the past few decades. Since almost every program needs to access some form of permanent data, therefore enormous efforts have been spent on developing particular database for specific applications or in using an interface between a database and a programming language. However, difficulties are frequently found due to the lack of enough programming tools for creating databases, or an interface does not exist and it would be practically impossible to re-structure the data to conform to the programming environment. Therefore, the idea of developing an integrated programming environment of databases has stimulated research works on combining databases and programming languages. This leads to the recognition of the need to provide an integrated system for programming and data management. Two approaches [66] are usually used to produce such a system: either by writing a completely new programming language or by enhancing an existing language with some form of database management. In either case, the usual strategy is to combine an existing language with an existing data model. These integrated languages are called **database programming languages** (or DBPLs for short). Particularly, DBPLs focus on the problems of providing a uniform typed system usually by the technique of polymorphism, and on mechanisms for data to persist. The former has just been discussed so the rest of this chapter will concentrate on the data persistence issue.

5.2.2. Motivations for data persistence

The persistence of a data object is the length of time that the object exists. In traditional programming languages, every object has a well-defined lifetime. A variable declared in a block or procedure will persist (exist) during the activation of that segment of code and thereafter it will be inaccessible. This mechanism is called **the scope rule**. Some langauges allow explicit control of persistence through the use of storage allocation procedures like the C's library functions **free** and **malloc**. These functions can apply to the full range of types. Without explicit de-allocation, the objects persist until the program terminates. In other words, data cannot last longer than the activation of a program without the explicit use of some storage agency such as a file system or a database management system (DBMS).

However, for some applications, data are needed to last longer than the duration of one program execution, eg. the personal record of a British citizen. Unfortunately, in most of the traditional langauges, the only objects with long-term persistence are files; in Pascal, files may also be parameterized by other types, eg. file of integer, file of real etc, except pointer types. But such file systems have the following disadvantages:

(a) It does not allow random access of data since searching is always started from the beginning of a file, i.e. all the previous data must be examined before getting to the target.

(b) No structure which means no security is imposed on the stored data and therefore it is up to the users to interpret the data.

These kinds of persistence problems lead to the development of the so called **persistent programming languages.**

5.2.3. Principles of persistence

Three principles are suggested by Cardelli et al [64]:

(a) Persistence should be a property of arbitrary values and not limited to certain types.

(b) All values should have equal rights to persistence.

(c) While a value persist, so should its description.

The first two principles state that a value's persistence should be regarded as a property of data orthogonal to its type. A corollary is that the code used to manipulate a value should not depend on its persistence. The third principle emphasizes the fact that it should not be possible to write out a value as one type and then subsequently read it in as another type. This final principle is often regarded as a property of program environments rather than a property of programming languages.

Nevertheless, one counter-argument to providing persistence exists - it is difficult to find good engineering techniques to support an arbitrary persistent structure. Certainly, the mechanisms for some types, such as those constructed as relations (relational databases), are better understood because of the substantial research efforts spent on them. To compete with these existing technologies, the general purpose methods will need to be as efficient in placing data and avoiding the transfer of irrelevant data. This, in turn, will require adequate mathematical models of data and program behaviour, and interpretation of these models to control the data collection and to select representations, access methods and so forth. Additional problems may arise over implementing concurrency, transactions and failure recovery. A sensible refutation of these counter-arguments will be the advent of some practical persistence systems which have performed functions for the arbitrary collection of data they run.

The following sections describe one of the data persistent languages called **PS-algol** with particular interest to its ability to provide data abstraction, data protection, modularity, separate compilation, binding and data persistence. The main reasons of choosing PS-algol rather than the other persistent programming languages are:

Practically : PS-algol is developed and most widespread in U.K. Therefore, it is easier to install; easier to obtain documentation; easier to make inquiries via public communication medium such as telephone, national mail systems etc.

Technically : PS-algol adopts a uniform approach to persistence-any value may persist independent of types including procedures. It also provides mechanisms for dynamic binding and type-checking . In each stage of the design, PS-algol designers have tried to permit eager type-checking so that the majority of code is **statically type checked.**

Since PS-algol is such a powerful and convenient tool, it is employed in conjunction with the language C as the basis to implement a distributed system as described in the next chapter.

5.2.4. The persistent programming language PS-algol

The language PS-algol [67] is a derivative of S-algol [68]. It is an experimental language designed to show that it is possible to provide persistence regardless of type [69] by means of a **persistent database system** and also to illustrate that graphically based human computer interaction can be supported by language features (such as bitmap images and line drawings). However, the latter property will not be discussed in this thesis.

PS-algol uses the notion of **first class procedures** (FCP), to provide all the features mentioned in section 5.2.3 (data abstraction, data protection, modularity etc), as described in the following sub-sections.

5.2.4.1. First class procedures

Most conventional programming languages provide facilities like functions and procedures as the only mechanisms for abstraction over expressions and statements, so that a user of a procedure/function (these two terms will be used interchangeably unless they have to be differentiated) only needs to know its effect, but not the details of how the procedure is executed or implemented.

As Morris [70] and Zilles [71] have pointed out, to exploit the use of procedural abstraction mechanisms to their full potential, it is necessary to promote **procedures** to be full **first class data objects** (referred as **first class procedures**). That is, procedures should be allowed the same rights as any other data objects in the language such as being persist, being assignable by the results of expressions, by the results of other procedures or by the elements of structures etc, not just being declared and being passed or executed as in Pascal and Algol-60. However, the most important concept in understanding first class procedures is that of **closure** [72]. The closure of a procedure includes all the information required to execute the procedure correctly. A closure has two parts: the code to execute the procedure and its environment which contains the local variables of the procedure. To understand more about FCP, consider the following PS-algol program:

```
let sum=proc(*int A->int)
begin
    let total:=0
    for i=lwb(A) to upb(A) do
        total:=total+A(i)
    total       ! return value of the procedure
end

write "How big is the vector"
let n:=readi()
let this.vector=vector 1::n of 0
write "Please enter",n," elements"
for i=1 to n do  this.vector(i):=readi()
let result:=sum(this.vector)
write  sum(this.vector)
```

Figure 5.2 An example of PS-algol's first class procedure

The program in Figure 5.2 reads in the size of a vector, followed by the value of each of its elements and finally prints out the sum of the vector's elements. Several points are worth noting:

(a) The dot in identifier such as this.vector is just part of the identifier, not an operator.

(b) All identifiers are introduced by the **let** declarations. This **let** is followed by := or = which signifying a variable or a constant, and then an expression. The type of the identifier is deduced from an initializing expression which must be present.

(c) Declarations and statements can be freely mixed in a program.

(d) The * symbol in the very first **let** declaration represents a one-dimensional vector in PS-algol.

(e) Two characteristics of FCP can be found in Figure 5.2. First of all, procedures can be assigned as in **let** sum=**proc**(***int->int**). Secondly, procedures can be passed as parameters, eg. procedure sum has been passed to the pre-defined library routine **write**.

(f) A value can be returned from a procedure via an identifier name or an expression by placing it in a single line such as the variable **total** inside the procedure sum.

5.2.4.2. Using FCP to implement ADTs

The notion of ADTs has been discussed in the previous chapter. As a recapitulation, ADTs provide programmers an abstraction mechanism and a protection mechanism. An ADT defines the operations available on the data object while only allowing the definition of the type to manipulate or access the representation.

In order to explain the data abstraction mechanism in PS-algol, a program is given in Figure 5.3. The task it sets out to solve is to define an abstract object for a complex number and to allow only the opeations of creation, addition and printing on the complex number.

In PS-algol, a structure class is a tuple of named fields with any number of any type. The **structure** statement adds to the current environment a binding in the closest enclosing scope for the class name (structure complex in Figure 5.3) and a binding for each field name (ipart and rpart in Figure 5.3). When an instance of a structure class is created (eg. using complex(i,r)), it yields an object of that class which may be assigned to an object of type pointer (**pntr**). However, the value of the pointer is not determined until run-time. Hence, the combination of the two classes structure and pointer give PS-algol certain degree of polymorphism because the pointer declaration in a procedure is just a place holder which can be replaced by any structure class.

Moreover, the fields of a structure are accessed using a pointer expression followed by the structure's field name in parenthesis, such as c(rpart) and c(ipart) in Figrue 5.3, provided the required structure class, complex in this case, is in scope. In PS-algol, the scope of an identifier starts immediately after the declaration and continues up to the end of a block. A block is either a procedure or statements within a **begin..end** construct or statements within {..}. However, if the same identifier name appears in two different blocks, then while the inner one is in scope, the outer one is not.

```
structure complex.pack(proc(real,real->pntr)new.complex;
                       proc(pntr,pntr->pntr)add.complex;
                       proc(pntr)print.complex)

let complex.package=proc(->pntr)
begin
  structure complex(real rpart,ipart)

  let n=proc(real i,r->pntr); complex(i,r)

  let a=proc(pntr c1,c2->pntr)
  begin
    complex(c1(rpart)+c2(rpart);  c1(ipart)+c2(ipart))
  end

  let p=proc(pntr c)
  begin
    write c(rpart)
    if c(ipart)<0 then
      write "-"
    else write "+"
    write "i"
  end

  complex.pack(n,a,p)

end

! Main program body

let cpack=complex.package()
let new=cpack(new.complex)
let add=cpack(add.complex)
let print=cpack(print.complex)
let c1=new(3,4)
let c2=new(2,7)
print(add(c1,c2))
```

Figure 5.3 Complex number package in PS-algol

Since the representation of a complex number (structure complex) is encapsulated in a block, it will be inaccessible to other parts of the program. Also, since the field names of the representation of the complex number are only local to a particular block, therefore only those procedures defined in that block can use these names. Thus, the representation of the data object is completely separated from its use which is one of the aims of an ADT. Indeed, the block could

be written in other co-ordinate systems, say the polar co-ordinates, without changing the external meaning of the abstract object **complex**. However, there is still a big drawback that is the lack of *information-hiding* mechanism. Because the details of the representation is visible to users, one can then change the structure complex or modifies its three procedures. The way PS-algol deals with such problem is to treat procedures as first class procedures and allows them to persist, so that the implementation details of an ADT can be compiled separately and stored in a PS-algol database with an associated entry name. If one wants to use the ADT, an application progam is needed to load the definition of the ADT from the database. The full details of this technique will be presented in the separate compilation section later.

5.2.4.3. Data protection

Morris [70] has specified three ways in which a data object may be used in a manner not intended:

(a) Alteration: An object that involves references may be changed without use of the primitive functions provided for the purpose.

(b) Discovery: The purpose of an object might be explored without using the primitive functions.

(c) Impersonation: An object, not intended to represent anything in particular, may be presented to a primitive function expecting an object representing something quite specific.

Since the names of the fields in a PS-algol structure class are only known to the primitive procedures by the scope rules, the objects can never be accessed except by those primitive functions, therefore the first two problems are eliminated. But, impersonation is really a serious problem in PS-algol because a pointer may point to a structure of any class and this will not be checked until a

program is being executed which may cause a serious failure. For instance, in procedure a of the complex number example shown in Figure 5.3, the two parameters c1 and c2 are expected to be of structure class complex but it would not be identified at compile-time. In order to make sure that c1 and c2 are bounded to the correct type before allowing any operations on them, the relational operator **isnt** is applied, as shown in Figure 5.4, to define the appropriate actions to deal with impersonations, but it is totally relying on the designers.

```
let complex.package=proc(->pntr)
begin
    structure complex(real ipart,rpart)

    let error=proc(pntr item->bool)
    begin
        if item isnt complex then
        begin
            write "Error"
            true
        end
        else false
    end

    let a=proc(pntr c1,c2->pntr)
    begin
        if error(c1) or error(c2) then nil
        else
              complex(c1(rpart)+c2(rpart), c1(ipart)+c2(ipart))
    end
     :
     :        Procedures n and p as in Figure 5.3
     :
end
```

Figure 5.4 Impersonation problem in PS-algol

5.2.4.4. FCP as Modules

The concept of **modules** has been introduced in many programming languages such as Ada, CLU, Modula-2 etc. Atkinson and Morrison [69] have pointed out that modules aim to serve the following functions:

(a) To provide a mechanism for the module's own data, that is, data bound with the module over the scope of lifetime of the module, rather than only for individual applications of the module.

(b) To be the unit of program building, being used in system construction as a unit of specification and as unit of a compilation, testing and assembly.

(c) As a localization or hiding of certain design decisions.

In conjunction with persistence as an orthogonal property, FCP performs all the above roles. The first function can depend on either on **partial application** or can be obtained in conjunction with program building facilities, that is, programs may use procedures which other programs have left in a database; the second function will be discussed in the next sub-section; and the last function has been already demonstrated in Figure 5.3.

To explain how FCP achieves the first function of a module using the technique of partial application, consider the procedure make.list.pack in Figure 5.5 which maintains a list of things to do for different people in different contexts.

```
structure list.pack(proc(string)add; proc()clear; proc()print)

let make.list.pack=proc(string person,context->pntr)
begin
  structure cell(string item; pntr next)

  let list.start:=nil


  list.pack(

  proc(string s);  list.start:=cell(s,list.start)        ! procedure add

  proc();   list.start:=nil                              ! procedure clear

  proc()                                                 ! procedure print
  begin
    write "'n list of tasks for",person,"doing",context
    let l:=list.start
    while l~=nil do
    begin
      write "'n",l(item)
      l:=l(next)
    end
    write "'n"
  end

  )
end
```

Figure 5.5 Procedure to implement a list package

Suppose a person has tasks in a number of contexts, procedure make.list.pack can be applied partially to yield procedures for each person as defined in Figure 5.6 and then those returned procedures can be used as shown in Figure 5.7 to obtain the list of tasks concerning the particular person. From this example, one can realize that make.list.pack originally requires two parameters supplied at the same time but now these two parameters are separated. Furthermore, each procedures yielded by the calling functions have data associated with them (the lists, the tasks and the persons), so the first requirement for modules has been met by FCP.

```
let make.list.for=proc(string person->proc(string->pntr)
begin
  proc(string context->pntr)
  begin
    make.list.pack(person,context)
  end
end
```

Figure 5.6 Partial application of make.list.pack

```
let Rons.list.maker=make.list.for("Ron")

let Malcolms.list.maker=make.list.for("Malcolm")

let MPA.paper=Malcolms.list.maker("First Class Finish Paper")

let MPA.shopping=Malcolms.list.maker("shopping")
```

Figure 5.7 Using the partial applied procedure

5.2.4.5. FCP in relation to persistency, separate compilation and binding

This section demonstrates how FCP can be used as the unit of system construction and the unit of definition. Suppose a system is to be built out of a telephone and address list for employees of a company. In order to separately compile the list maintainer, one could write a PS-algol program such as that shown in Figure 5.8 :

```
structure person(string name,phone.nb; pntr addr,other)
structure address(string nb,street,town; pntr next.addr)

let pw="ok"
let db:=open.database("list",pw,"write")
if db is error.record do
begin
   db:=create.database("list",pw)
   if db is error.record do
   begin
      write "The database cannot be created because",db(error.explain),"'n"
      abort
   end
end

write "Name          ?";      let this.name=read.a.line()
write "Phone number?";        let this.phone=read.a.line()
write "House number?";        let this.house=read.a.line()
write "Street        ?";      let this.street=read.a.line()
write "Town          ?";      let this.town=read.a.line()

! Construct the address record
let this.addr=address(this.house,this.street,this.town,nil)

! Construct an employee record
let this.person=person(this.name,this.phone,this.addr,nil)

! Look for an entry called employee.info in the database
let addr.list:=s.lookup("employee.info",db)

! If it is not the first ever entry then create a new table for it
if addr.list=nil do
begin
   addr.list:=table()
   s.enter("employee.info",db,addr.list)
end

! store the list into the database
s.enter(this.name,addr.list,this.person)
let committed=commit()
if committed=nil then write "Data recordeed'n"
else write "Data not recorded because ",committed(error.explain),"'n"
```

Figure 5.8 A program to add one employee to a company's address list

Before proceeding any further, the persistent mechanism of PS-algol is reviewed at this point.

5.2.4.5.1. Persistent data in PS-algol

The persistence of data is the length of time that the data exists. In PS-algol, any data item (including procedures) has the rights for the full range of persistence (i.e. from temporary results in evaluating expressions to data which may outlive the program). Therefore, PS-algol provides mechanisms for the programmer to identify which data is to persist and in which database it should persist, and also provides mechanisms for the storage and retrieval of persistent data. All these mechanisms are available via a set of standard procedures in a PS-algol library as described shortly. The procedures are divided into two groups: the group concerned with identifying the relationship between data and the database and the implementation of transactions; the group concerned with providing new data structures.

Database procedures : All data which persists longer than the execution of a program is held in a database. In order to inform the database system that a new database is required, the database must be created first using the standard procedure **create.database** which is defined as:

let create.database=**proc**(**string** dbname,pw->**pntr**)

then the newly created database can be opened using procedure **open.database** defined as:

let open.database=**proc**(**string** dbname,pw,mode->**pntr**)

and finally the database can be used. In the definitions of these two procedures, dbname is the name of the requested database, pw is a password to prevent un-authorized user and mode is either **read** or **write**. Note also that these two

procedures both return a pointer to a table (see below). Whatever operations are performed on the database, no changes are recorded unless the program has executed a call of **commit**:

```
let commit=proc(->pntr)
```

This procedure commits the changes made so far to the database opened by the program. Either all or none of the changes will be recorded (atomic transaction), so the programmer can ensure that all the databases are consistent. It is only after a commit that the changes made can be observed by other programs. Note that only the changes prior to the last commit of a program, if any, are recorded in the database. In the case of an error, an instance of the class **error.record** will be returned which is defined as:

```
structure error.record(string error.context,error.fault,error.explain)
```

where **error.context** is the name of the procedure called by the user in which the error was detected, **error.fault** is a short constant string defining the error suitable for testing and **error.explain** is a readable explanation of the fault that a simple program can print. However, if no error occurs, the result of a commit is the **nil** pointer. Finally, there is a special routine called **abort** which may be used to abort a PS-algol program whenever something is wrong during the transaction of a database.

Tables : Tables are a system supported data structure in PS-algol. They are commonly used and needed for building databases, but may also be used for temporary structures. A table stores an updatable mapping from keys to values. The keys may be integers or strings and the values are pointers to instances of any structures. These mappings are usually implemented by binary-trees or some adaptive hashing techniques such as hashed trees. The standard procedure **table**:

```
let table=proc(->pntr)
```

creates a new empty table and returns a pointer as a token for it, which is an instance of the structure class **Table**. The keys to values mapping can be done via one of the following ways:

(a) Two procedures are used to modify the entries in the table given as the parameter table below:

```
let s.enter=proc(string key; pntr table,value)
let i.enter=proc(int key; pntr table,value)
```

A table may contain entries whose keys are integers or entries whose keys are strings, a key of one type never matches a key of the other. However, when new association is recorded in the table between the key and the value, this supersedes any previous association for that key that was held in the table. If the value is **nil**, the effect is to remove any existing entry for the given key from the table.

(b) Two procedures are used to return value associated with a given key from the given table. If there is no entry for that key, the result is **nil**.

```
let s.lookup=proc(string key; pntr table->pntr)
let i.lookup=proc(int key; pntr table->pntr)
```

(c) The two routines:

```
let s.scan=proc(pntr table; proc(string,pntr->bool)user->int)
let i.scan=proc(pntr table; proc(int,pntr->bool)user->int)
```

apply the function **user**, provided by the programmer, to the entries in the stored table given as the first parameter. The given function is applied to every elements with a string key by s.scan and to every elements with an integer key by i.scan. The function will be repeatedly called with the key as its first parameter and the associated value as second parameter. However,the function is called with keys in ascending numerical order by

i.scan and with keys in ascending lexical order of ASCII strings by s.scan. Repetition continues until either all the entries for the specified type of key have been parameters to function **user** or until it returns **false**. Each scan functions returns the number of times **user** were called.

5.2.4.5.2. Separately compiled code

Having understood the persistent mechanism in PS-algol, consider the following example which uses the address list created by the program in Figure 5.8 to look up the phone number of a particular employee as in Figure 5.9 :

```
! open the appropriate database
let pw="ok"
let db=open.database("list",pw,"read")
if db is error.record do
begin
    write "Cannot open database because ",db(error.explain),"'n"
    abort
end


! locate the entry position of the required data in the database
structure person(string name,phone.no; pntr addr,other)
let addr.list=s.lookup("employee.info",db)
if addr.list=nil do
begin
    write "No address list yet'n"
    abort
end


! Find the phone number of a particular employee
write "Name   ?"; let this.name=read.a.line()
let this.person=s.lookup(this.name,addr.list)
if this.person=nil then write  this.person,"not known'n"
else write "Phone number of ",this.name,"is",this.person(phone.no)
```

Figure 5.9 An example of using separately compiled code in PS-algol

Careful examination of Figure 5.9 shows a number of interesting features. First of all, previously stored information is obtained from a database (called list in this example). Secondly, a structure class (person) is present which indicates the internal structure of the information stored in the database. The existence of this structure class is vital because it regulates the way how those information can be accessed later in the program. However, the field names of this structure definition are not as important as their types .

The above example has illustrated the basic idea of a technique called **procedure library** in PS-algol. Since PS-algol allows procedures to persist, so instead of storing ordinary data objects, a table of procedure names associated with structures containing the procedures are stored in a database (as a library) to allow users to construct a program out of these separated compiled routines. Consequently, the effect of modularity can be achieved by storing procedure in this way for two reasons:

(a) The module is used without its implementation being seen by the programmer.

(b) Modules can be synthesized using other modules, allowing construction of large programs, while the individual program text that has to be read to understand the program at a given level is kept small.

This gives the basis for constructing a variety of software construction tools within PS-algol.

5.2.4.5.3. Binding

Binding is a method which maps variables used in a program to the corresponding physical values in the computer memory (may be primary or secondary). PS-algol uses both dynamic and static binding techniques. Variables within program text are statically bound, but values of pointer

variables are dynamically bound to instances of structure classes that is the *reason why pointer variables are determined at run-time.*

The dynamic binding feature does not prevent programs being strongly typed, but it implies that some type-checking is dynamic (occurring near in time to the final evaluation) rather than static (occurring at initial program analysis). So partially analyzed programs may be left for further analysis and evaluation when data and resources become available. The consequences may be that checks can be factored out of the internal operations. This dynamic evaluation has inevitable costs. The program, that depends on it, may contain errors which could have been detected at compile-time and hence reported more opportunately. Also, the program which uses it, may execute more slowly because of run-time checks. But many programs even in a statically checked language, may need equivalent run-time checks, for example, to check the tag field of a Pascal case statement matches the field being used in a record. However, denying dynamic binding, the class of programs that can be written is reduced; for instance, without dynamic binding, it would not be possible to write the associative structure in PS-algol that maps to members of a mixture of structure classes including those yet to be declared when the map is created. This lost opportunity can lead to other costs : either these general purpose components cannot be written within the language so that some other languages have to be used, or the programmer simulates types and languages within the original language. The former incurs the cost of linking between languages, loss portability and increased programming difficulty. The latter incurs the cost of an extra layer of interpretation, mapping and obscure convention to trap people who subsequently modify the program. So the choice of static or dynamic checking is a matter of choosing the appropriate tradeoffs between **security** and **flexibility**.

However, even if dynamic checking is adopted, the language should enable programmer to choose the points at which dynamic binding is required since to bind all names dynamically would be very un-economical. Therefore, one would typically expect small units to be statically bound and to form larger units dynamically bound together. In PS-algol, the choice is offered via the

constancy mechanism. That is, if an identifier is bound constantly (using the symbol =), the system may assume that it will not change. But if it is an variable (using the sequence :=), its value may be replaced. In either case, since many instances of a class may exist, an instance which implements an ADT in a way believed to be appropriate to that particular usage can be bound. Subsequently, if another implementation appears to be more suitable, the new value can then be assigned. Finally, the binding mechanism of PS-algol, in which the type of a referend is checked when it is first brought from the database, has provided incremental type checked linking and loading.

5.2.4.6. Comparison of the PS-algol and INGRES database systems

The PS-algol database system has been used mainly for academic research studies. This section presents a summary of the major difference between the PS-algol database system and one of the most sophisticated commercial database system called INGRES [73]. INGRES is a distributed relational database management system which can be used on a wide range of microcomputers, minicomputers and mainframes to define, manipulate and protect user data and to develop new applications for users. The INGRES database system is superior to the PS-algol one with respect to the following aspects:

(a) Since INGRES is a distributed database management system, a remote database may include data residing in INGRES databases on several other computers. This distributed database capability is achieved via INGRES/STAR.

(b) INGRES can retrieve data stored in a database interactively using a structured format known as **form** which is the electronic equivalent of a paper form; it is used for data input and data display. With forms, INGRES allows queries to be made on data stored in the different entries (tables) of a database.

(c) A **pull-down** menu is supported which allows users to work with the various pre-defined INGRES tools (modules) to manipulate information in a database.

(d) INGRES provides a **help** facility with information about the usages of each INGRES tools so that users can understand what is happening at every stages.

(e) INGRES supports two query languages, **Structured-Query-Language** (SQL) and **QUEry Language** (QUEL), to enhance data transactions.

(f) A language called INGRES 4GL which provides a tool for establishing new applications using INGRES forms, eg. create a new module for accessing data on a particular database.

(g) A tool called **Visual-Query-Language** (VQL) which lets user reformat data from INGRES database tables into other Personal Computer productivity software such as LOTUS 1-2-3.

(h) Another distinctive feature of INGRES is its ability to set levels and types of locking on a database using SQL's **setlockmode** command. There are two levels and two types of locking. They are **defaults, sessions, shared** and **exclusive** locks respectively. Unfortunately, locking introduces deadlocks even though they enable better concurrency control. However, once a deadlock is detected by INGRES, one of the transactions will be chosen as the victim to be aborted and then an error message will be sent to it. Thus, the user has to be awared of this situation in order to take the appropriate remedial actions.

Although INGRES is so powerful, it does not support un-structured data access (i.e. the row-of-byte strategy) since data can only be stored in terms of forms (structured items) which makes it un-suitable for the purpose of this thesis.

5.2.4.7. Criticisms of PS-algol

Having experienced with some PS-algol programs with the aid of the two manuals [74] and [75], the following deficiencies are found in the current implementation of PS-algol:

(a) PS-algol is a one-language (standalone) system i.e. software written in other languages cannot be used. This is very inconvenient in the sense that the users have to write all their own routines apart from the standard ones.

(b) There is no clear distinction between **reserved words** and **identifiers**. For instance, all the reserved words must be in lower case letters but identifier can be in any combination of upper and lower case letters. There is a cruel way to solve this problem by under-lining all the reserved words using the directive %ul after %list but there are some additional characters eg. hypen, quotes appearing in the output listing of the compiled program which may confuse users.

(c) Slightly complicated I/O processes from the programmer point of view. For example, once a read statement is executed, i.e. the program is in read mode, all the subsequent write operations will be suspended until a RETURN key is pressed. Therefore, programmer must have a good plan in constructing the output format.

(d) All the various read operations are implemented in terms of function calls rather than procedure calls which results in a lot of PS-algol assignment statements.

(e) If a string is required as input, it must be enclosed in double quotes, e.g. "hello". The quotes can be omitted if the function **read.a.line** is used instead of the function **reads** with the expense of losing certain degree of data integrity because all the printable characters can be accepted as the input in this case.

(f) There is no type called **char** as in Pascal and therefore the type **string** has to be used all the times.

(g) Since all the structures are referred as type **pntr**, there is a possibility for type mis-understanding and impersonation problem as discussed in section 5.2.4.3.

(h) Nested procedure constructs are not allowed.

(i) One procedure can call another procedure if and only if the latter has already been defined.

(j) Each statement of a PS-algol program usually occupies only one line of text. However, in some circumstances, it is necessary to construct a single statement with two or more lines of text such that the program is more structured and hence people can understand it easier. Consider the following cases:

(1) If a, b, c, d, e and f are very long boolean expressions, it will be more understandable and natural to construct the statement

```
if (a and b and c and
    d and e and f) then do something
```

than the **nested** if statements

```
if (a and b and c) then
    if (d and e and f) then do something
```

In this example, the logical operator **and**, which must be situated at the end of a line, is served as a line separator. In fact, all PS-algol operators (eg. or, not, +, -, *, /, etc) can be used for the same purpose.

(2) Another commonly used line separator is the symbol comma as in the following write statement:

```
write "The total number of input characters are",
      10000000000000000001
```

(k) Some comments about the PS-algol database system are:

(1) Although the present PS-algol system does not seem to have a limit on the total number of databases available to a user, all the information about the databases can be only retrieved via application programs. For example, in order to obtain a directory of the existing databases, a program contains only the function **list.database.dir** is required. Hopefully, there should be a better interface between the PS-algol and the operating system in the future.

(2) No database can be used unless it has been created explicitly. Therefore, it is invalid to open a non-existing database. Unlike in Pascal, if a file does not exist, it will be created automatically before opening it.

(3) In the present PS-algol system, databases can only be removed by the routine **remove.database**. Unfortunately, the removed database still occupies an entry on the database directory (as proved by the result of calling **list.database.dir**) which means the database is just removed logically but not physically. This is due to the absence of a PS-algol garbage collector which is claimed to be available very soon.

(l) PS-algol communicates with Unix via files to receive data and send results. The function **create** produces a new object of type **file** called a **file descriptor**.

```
let create=proc(cstring name; cint flag->file)
```

The parameter flag indicates the access mode of the file descriptor. For instance, to create a file for write only, the decimal number 128 is used; for read and write by the owner only, the decimal number 384 is used. From these examples, it can be realized that the effect of flag is to set the respective status bits of the newly created file.

5.3. Conclusions

Traditionally, the interface between a programming language and a database has either been through a set of relatively low-level subroutine calls, or it has required some form of embedding of one language in another. Recently, the necessity of integrating database and programming language techniques has received much attention. As a result, a new range of programming languages, namely the database programming languages (DBPLs), were evolved. Two major design issues of a DBPL are the provision of a uniform typed system and mechanisms for data to persist. This chapter has presented a detailed discussion on these two issues.

A uniform typed system is often accomplished by using some kind of type checking or type inference mechanism. In the former case, the types of each data objects and expressions are checked before a program is executed whereas the latter can infer the types of objects and expressions from local contexts. The notion of polymorphism has enhanced the task of type checking. There are two kinds of polymorphism: universal and ad-hoc polymorphism. Mechanisms for these two kinds of polymorphism such as parameterization, inclusion, suntyping, type inheritance, overloading and coercion are examined, followed by the discussion of the type inference algorithm used by the language ML.

Having discussed the requirements for a uniform typed system, the issue of data persistence is addressed. The aim for persistent data is to support applications that require data to be last longer than the duration of a program execution. This leads to the development of persistent programming languages. The distinctive features of one of these languages called PS-algol have been described with particular interest to its

notion of first-class-procedures (FCP). The main reason of choosing this langauge is due to its ability to support both the structured and un-structured data access methods (which is the heart of this thesis) via FCP as illustrated by the PS-algol programming examples given in this chapter.