THE UNIVERSITY OF AALBORG

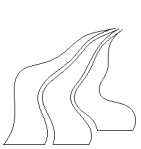
Design and Implementation of a HyperBase

by

Uffe Kock Wiil et al.

IR 90-03 September 1990

Institute for Electronic Systems
Department of Mathematics and Computer Science
Fredrik Bajers Vej 7 — DK 9220 Aalborg Ø — Denmark
Tel.: +45 98 15 85 22 — TELEX 69 790 aub dk



Design

and

Implementation

of a

HyperBase

AUC S9D Gr. 424

January 1990

AALBORG UNIVERSITY CENTER

INSTITUTE FOR ELECTRONIC SYSTEMS

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Title:

Design and Implementation of a HyperBase

Made by:

Uffe Kock Wiil
Claus Bo Nielsen
Carsten Ruseng Jakobsen
Finn Sølvsten
Per Magnus Petersen
Hans Mejdahl Jeppesen
Poul Larsen

Supervisor:

Kasper Østerbye

Number printed: 100

Pages: 53

Project period: 1/9 '89 – 5/1 '90

Abstract

Task: The overall task of this terms project has been to design and implement a HyperBase, which can be used as a tool for HyperMedia applications. HyperMedia application programmers should be able to construct their own special purpose HyperBase, by using the facilities provided by the HyperBase. Therefore the major design goals was to make the HyperBase as general as possible, by giving a small set of general operations on the HyperBase, and to make the HyperBase easy to use and easy to extend.

Solution: To implement the HyperBase, the system was divided into five separate blocks with distinct functions and interfaces. The five blocks are: UNIX File System Interface, Operations, Event/Lock Handling, Network Communication, and Client Interface. The idea was that these blocks could be implemented and tested alone. To test the total system an application using the HyperBase was created.

State of the project: Each of the above mentioned five distinct blocks are implemented and tested. The design and implementation is documented in this report. A manual containing a users and a programmers guide is also written.

Preface

This report is written by 7 computer engineer students on the 9. term in the fall 1989. The purpose of this terms project has been to create a tool to be used as a basis in our graduation projects and in the TILDA group at our university.

This report contains the following 4 chapters: Introduction, Design, Implementation, and Conclusion, and an appendix containing a description of an application using the HyperBase. Furthermore a manual for the HyperBase tool is written and supplied as an appendix.

A user of the HyperBase should read the Design chapter and the users guide in the Manual. A programmer, who wishes to extend the HyperBase, should read the Design, the Implementation chapter, and the programmers guide in the Manual.

Aalborg University Center January 5, 1990.

Uffe Kock Wiil

Carsten Ruseng Jakobsen

Claus Bo Nielsen

Hans Mejdahl Jeppesen

Finn Sølvsten

Poul Larsen

Per Magnus Petersen

Contents

1 Introduction **5**

1.1 The HyperText idea 5

1.2 Basic elements and operations 6

1.3 HyperText as a file system 7

1.4 Using HyperText 8

1.5 HyperBase 9

2 Design **10**

2.1 Design decisions 11

2.2 HyperBase model 12

2.3 Basic entities 13

2.3.1 General choices concerning nodes and links 13

2.3.2 Key fields in link nodes and data nodes 16

2.3.3 File organization 18

2.4 Basic services 18

2.4.1 Structure commands 18

2.4.2 Read and write commands 19

2.4.3 Event and lock commands 19

2.4.4 Connection and disconnection commands 20

2.5 The configuration file 20

2.5.1 Design of the HyperBase Language 21

3 Implementation **23**

3.1 Introduction 23

3.2 UNIX File System Interface, block 1 25

3.2.1 Storing and retrieving data and link numbers 26

3.3 Operations on the HyperBase, block 2 28

3.4 Events and Locks, block 3 29

3.4.1 Main part . . . 30
3.4.2 Event part . . . 30
3.4.3 Lock part . . . 31
3.5 Network Communication Block, block 4 . . . 31
3.5.1 The function of block 4 . . . 31
3.5.2 The classes of block 4 . . . 32
3.6 The User Interface, block 5 . . . 32
3.6.1 C-interface . . . 33
3.7 The configuration program . . . 33

4 Conclusion **34**
4.1 Design considerations . . . 34
4.2 Summary of the work . . . 35
4.3 Evaluation . . . 36
4.3.1 Design . . . 36
4.3.2 Implementation . . . 36
4.4 Future work . . . 36
4.5 Final conclusion . . . 37

A HyperMedia Application **39**
A.1 Design . . . 39
A.1.1 The primitives . . . 40
A.2 The program . . . 40
A.2.1 The configuration file . . . 41
A.2.2 Implementation . . . 42
A.3 Conclusion . . . 43

B Manual **45**
B.1 Users guide . . . 45
B.1.1 What is HyperText . . . 45
B.1.2 What is a HyperBase . . . 45
B.1.3 The HyperBase operations . . . 46
B.1.4 Configuration of the HyperBase . . . 49
B.1.5 Error codes . . . 50
B.2 Programmers guide . . . 50
B.2.1 Extra key fields . . . 50
B.2.2 Extra operations . . . 52

B.2.3 Modularity 53
B.3 Lacks 53

Chapter 1

Introduction

Solving problems using computers is a process of transforming a description of the problem from a natural language to a more formal description based upon the computer environment. It is obvious that a good correspondence between these descriptions, will be the basis for good solutions. The development of high level programming languages was one of the first improvements used to achieve part of this correspondence. When the principles of high level languages was well founded, an interrest in the surrounding environment evolved. Today the need for a total environment supporting the developers natural way of thinking, is widely accepted.

Recently the interrest has focused on the possible ways of structuring documents. This has resulted in the concept of HyperText. HyperText is normally viewed as a general way of representing text documents, but it can also be applied to many other fields like structuring sound, maps, or pictures. The latter is often called HyperMedia systems. There is one restriction to the term HyperMedia, the substance of the structured data has to be in binary form.

The prefix Hyper is used for any system which can be diveded in:

- A window system performing the user interface.
- A database system supporting storing and retrieval of data organized in a graph structure.

Most often the term is used in connection with a HyperText system. The following section will give an overview of HyperText, which is general for most HyperMedia systems.

1.1 The HyperText idea

It was Vannevar Bush [Bush 1945] who developed the idea of HyperText, and described it in his article "As we may think" in 1945. The ideas presented here inspired Douglas Engelbart [Engel 1963] to make AUGMENT, described in his article "A Conceptual Framework for the Augmentation of Man's Intellect" in 1963. Bush created a system MEMEX which handled large amounts of notes, pictures and scientifical literature in a graphical system with browsing, implemented with microfilm and photocells. Engelbart created a similar system on a computer workstation with a device he named *the mouse*. Bush's and Engelbart's view of HyperText systems can almost be read from the titles of their articles. They focused on the way we store and retrieve informations in the mind, and wanted to create a system which could support this process, and in this way

augment the human intellect. Engelbart viewed the workstation with a mouse and a HyperText system like AUGMENT, as an extension of the tool "pen and paper" supporting intellectual creative processes.

While AUGMENT still was being developed Ted Nelson started a similar project called XANADU. Ted Nelson was the first who used the term HyperText, for the special way text was organized in MEMEX, AUGMENT and XANADU. Nelson was inspired by AUGMENT, but he's interest was literary. He wanted to implement a system for storage and retrieval of linked and windowing text. From this point many have contributed to the idea of HyperText and many systems have been implemented.

HyperText is a way of handling large volumes of text in an organized manner. HyperText provides facilities for structuring text in several ways and to link documents so that related information is easily found.

The simplest definition [Fischer 88] of HyperText is *nonlinear text*. Text in an ordinary book is arranged in a linear fashion and is supposed to be read in a linear way. Exceptions from these rules exist, such as a dictionary, a thesaurus, and so forth. In HyperText, information is stored as a directed graph, with *nodes* holding pieces of information and *links* connecting the pieces.

In most conventional paper documents, physical structure and logical structure is closely related. HyperText is a way of freeing the logical structure from the physical. HyperText can be viewed as a generalization of ordinary text. The nodes in a HyperText system can be linked in sequence, creating an ordinary, linear text document. HyperText can also be structured as a *tree*, providing an outline, or in any other way. The collection of all nodes and links will in general result in a graph.

Most HyperText documents have several kinds of links. Some links form the *structure* of the document while other links provide *cross references* to related documents or parts of documents. The structure links might organize the document as a tree, forming an outline of the document. Cross references might point to definitions of terms, further readings, other topics of relevance, and so forth.

1.2 Basic elements and operations

Links and nodes are the two most important keywords, when describing HyperText systems. The term entity will be used to describe either a link or a node, and entities will refer to both. To classify HyperText systems, the definition of entities often characterizes the derivation between different implementations. However the following will in general be true for most HyperText systems:

- The nodes hold the material stored in the HyperText system. This can be text in a pure HyperText system, or it can be any other kind of material in a generalized HyperMedia system.
- The links connects the nodes to form the HyperText graph. A link can be regarded as a *pointer*, leading the reader from one node to a node with related material.
- Both nodes and links can be typed. This can be used to mark specific kinds of nodes, so that a browser might treat specifications, design documents, and source code differently, and so that structure links can be treated differently from cross reference links.

Nodes and links are the basic entities in a HyperText system, and it is upon these, the basic operations of a HyperText system is build. A HyperText system must provide a browsing mechanism. Browsing is the process of following a path through the graph. In most cases the traversal through the graph will be forth and back, e.g. when finished reading a reference, it is natural to continue reading the text from the point where the reference was embedded in the text. Browsing can be provided in different ways like:

- Interactive reading: By following links and opening windows successively displaying the contents of the node referenced.
- Information retrieval: The ability to search through the HyperText database for some string in the data, or match a search string to the contents of an attribute in any of the entities.
- Graphical Overview: A way to display the entire or parts of the HyperText system graphically. An important issue is the ability to select between different types of links and or nodes if these are typed.

The HyperText system must provide operations which easily creates nodes and links. Further must it provide a number of operations to manipulate the windows in the user interface. Aside from these operations, we have found it important to apply the event mechanism known from the server and client model to HyperText. It is our belief that a HyperText system must provide a mechanism to subscribe to an event. The event handling could be defined as a set of basic operations that the HyperText system performed on the entities. This definition can easily be improved to more sophisticated event mechanisms. We will discuss this in chapter 2.

1.3 HyperText as a file system

HyperText can be applied as a file system, where files are organized like a directed graph. Each node in the graph contains data, e.g. part of a text file or program, the links constitute different ways of structuring the data stored in the nodes. One path through the links might result in a program listing or a design document. Another path through the nodes, could be from a design node to a specification node to a source code node to a binary version of the program text. A HyperText based file system is very powerful, and other well known file organizations like trees or flat structures turns out to be special cases of the HyperText organization. This gives the user a very flexible way of storing files or documents.

Historically limitations of computers have always been enforced to the humans using the computer. In the time of CP/M the only way of structuring files was in a flat structure. Other systems provided a 2-level organization, with all system files at the top and user files in a flat structure below (e.g. NOS/TELEX). Later the tree structure organization of files was designed in UNIX. HyperText file systems can be viewed as another extension to the existing way of organizing files.

Using HyperText as the concept for a file system also provides new ways of handling problems like mutual exclusion, by using the event mechanism. Instead of a brute lock excluding the latest process, the file system detects that two processes wants to change the same node, and informs both of them that they are in conflict. It would then be possible for the two users to negotiate about the right to change the node.

HyperText could be provided in other ways; e.g. as a library, as part of a programming language, or as an integrated part of any environment.

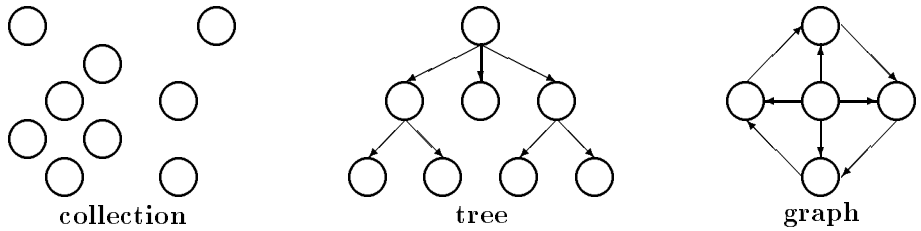


Figure 1.1: *Different forms of file organization.*

1.4 Using HyperText

The concepts of HyperText turns out to be a natural way of structuring many problems. In the following two of them will be introduced.

The idea of version control systems has traditionally supported a linear or a tree structured development process. Experience have shown that very often the development process is more appropriate described as a graph. It is not unusual that system development goes through a process like the one shown in figure 1.2.

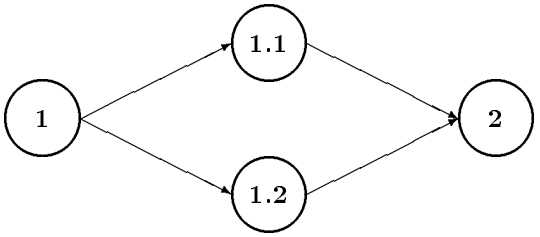


Figure 1.2: *Using HyperText in a version control system.*

The first release (1) is derived in two directions (1.1 1.2), and later these two form the foundation for a new version (2) based on both of them. There might be examples of even more complex relations between different versions of a system, where it will be necessary to express these by graphs. One way to implement such version control systems could be by the means of HyperText, since the problem is described in terms of a graph which HyperText supports.

Libraries are examples from real life where things actually is structured very heavily. Most operations performed on a library relies on references of different types. All these references structures the content of the library in different ways. In an ordinary library such structures could refer to indexes with author, title, or subject respectively as key. Another structure could be a book, and all references from that book, and references from the books referenced, and so forth. A system handling all these structures at the same time, could clearly take advantage of a HyperText system.

Lately a group at the University of Aalborg has investigated how to create and maintain a library of source code. They investigated methods to classify and structure such a library, and realized that they were having almost the same problems as an ordinary library. They also found that HyperText would be an appropriate tool to solve the problems at hand.

1.5 HyperBase

From the above it can be seen that HyperMedia can be applied to different problems. However different problems often desires different implementations of entities. This suggests the need for a general tool supporting HyperMedia developers. Such tools should capture the general idea of HyperMedia and provide basic tools operating on entities, and further it should be easy to extend and adapt to a certain field. The term HyperBase will be used to denote that kind of tools. HyperBase is the database part in a HyperMedia system.

Chapter 2

Design

In this chapter the design of the HyperBase will be presented. The HyperBase provides the most basic services of a HyperMedia system, like nodes, links, storing, locking, and event handling.

The intention of this design is to make a HyperBase, which can be used as a tool for creating different HyperMedia applications. HyperMedia application programmers should be able to construct their own special purpose HyperBase, by using the facilities provided by the HyperBase. It is therefore our goal in the design:

- To make the HyperBase as general as possible, by providing a small set of general operations on the HyperBase.
- To make the HyperBase easy to use and easy to extend.

A HyperBase designed and implemented with the above mentioned general design decisions in mind, should be able to support every imaginable HyperMedia application. That is, the HyperBase is thought of, as a configurationable foundation (a file system), usable for any imaginable HyperMedia application. A collection of different HyperMedia applications (a HyperMedia system) should be able to use the same special purpose HyperBase as file system, which is shown in figure 2.1.

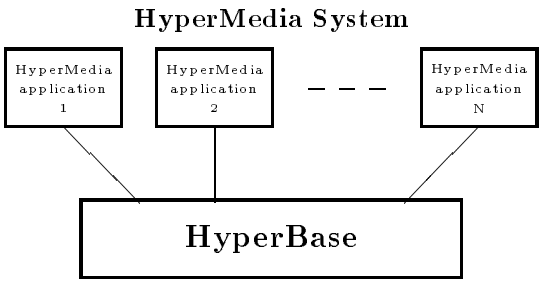


Figure 2.1: *A HyperMedia System in connection with a special purpose HyperBase.*

2.1 Design decisions

The HyperBase should be able to operate in a multiuser environment. To support the multiuser situation and continual keep the data in the HyperBase consistent, the HyperBase must provide a possibility of protecting the different entities in the HyperBase. This protection is obtained with the locking mechanism. The fact that the HyperBase is operating in a multiuser environment makes it a shared resource. To keep consistency in the HyperBase, it must be assured that there is only one HyperMedia client accessing the HyperBase at a certain time. Because of this demand and because a real distributed HyperBase system is to complicated for a one term project, the HyperBase is thought implemented as a server, making it possible for several users to access the HyperBase alternating. The relations between the server and the HyperMedia clients is depicted in figure 2.2.

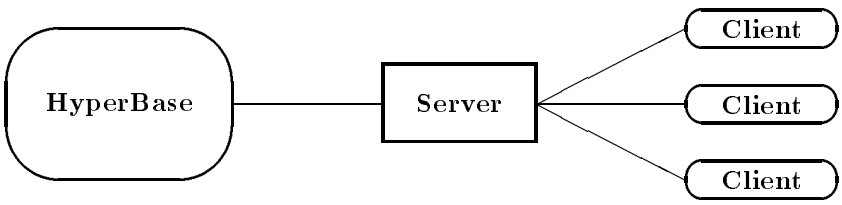


Figure 2.2: *The client - server model. The server has direct access to the shared resource (the HyperBase), and all clients must access the resource through the server.*

In this way it is up to the server to ensure mutual exclusion on the HyperBase. Since the server is the only process which can access the HyperBase, it must be guaranteed, that the services provided by the server is strong enough to support all possible requests from the clients. But there is one drawback by introducing locks. When introducing a locking mechanism, problems with different states in the HyperBase server arises. When a server with states crashes, all locks must be released, and all clients set back to some initial state, before it is possible to restart the server.

It is possible for a user to open several connections to one HyperBase, or connections to different HyperBases on different machines at the same time. It therefore should be possible to run more than one HyperBase at the same time, but each HyperBase has only one server located on exactly one machine. This causes that there must be a way to tell which HyperBase the user works in.

If the HyperBase user should be able to find his way in the HyperBase graph or even just to find the graph, there must be some initial node or initial link, so that every time the user connects to the HyperBase, he always starts out at the same place (node or link) in the graph. Different users could have different initial nodes or links.

In order to be able to use the HyperBase in different applications, the concepts of a HyperBase Language and a configuration file is introduced. The use of the configuration file in the HyperBase, is based on the idea that some of the essential HyperBase parameters can be defined at installation time. The HyperMedia application programmer writes a configuration file in the HyperBase Language, specifying how he will shape the entities in the HyperBase. The configuration file is used as input to a HyperBase Language parser, which generates a header file to be included in the HyperBase code. Then the HyperBase is compiled, and now the HyperBase structure is **static**.

The HyperBase is chosen to be static of simplicity reasons. If all data nodes and all link nodes have the same size and form they are easy to manipulate efficient.

2.2 HyperBase model

The ideas about HyperText were evolving before workstations with Ethernet was developed, and hence implemented on expensive mainframes. But most developers believed that within one or two decades, that kind of equipment would be spread to offices. In a time where this belief has become reality new and more specialized views of HyperText has evolved. One of these is the idea of creating tools in the programming environment for programmers based upon the concept of HyperText. The major point is that the process of writing programs and their documentation involves large amounts of interrelated data. The programming process could be supported by a HyperText system storing these data and relations, and supply the programmer with tools to get the information about either data or relations as needed.

In order to do experiments with such programming environments, different HyperText systems must be implemented and analyzed. This results in a natural demand for a tool supporting programmers writing HyperText applications. HyperBase is such a tool, and can be thought of as a "file system" for HyperText. The intention of HyperBase is to capture the major concepts of HyperText, so generally that any HyperText system can be build upon HyperBase. Therefore HyperBase supports directly the notion of links and nodes. It must be observed that links and nodes, doesn't contain information, but only structure. The information is stored as a chunk of data underneath each node, and the HyperText system programmer, can use this chunk, just as he pleases. This organization makes very dynamic systems possible, since all informations are maintained by the HyperText system interpreting the data stored below a node. If the user decides to create a new attribute to nodes at HyperText level, this information is stored in the data field at HyperBase level. The HyperText programmer extends the HyperText system to reflect this new change, but is still using the same HyperBase (file system). This organization is illustrated in figure 2.3.

Figure 2.3: *The relation between HyperText and HyperBase. HyperBase implements the two bottom levels.*

In the following a model of the HyperBase is presented. The intention of this model is to show how the different parts of the HyperBase are correlated.

As shown in figure 2.4 the HyperBase consists of three parts:

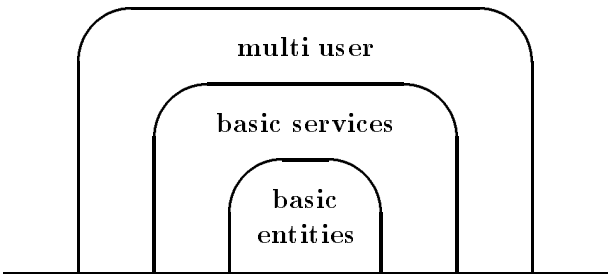


Figure 2.4: *The different parts of the HyperBase in connection with each other.*

- basic entities, like nodes and links.
- basic services, like building graphs, lock and event handling.
- a multiuser part.

The two basic levels could operate alone with just one user, so it could be said that the multiuser part of the HyperBase logically provides all the facilities necessary for the multiuser environment.

In the following sections the design of the basic entities, the basic services, and the design and structure of the configuration file will be presented.

2.3 Basic entities

This section describes the basic entities, e.g. how they are correlated, and what informations they hold.

2.3.1 General choices concerning nodes and links

In a simple model the structure of the HyperBase could be represented as a directed graph, where each node have a reference to its data (see figure 2.5).

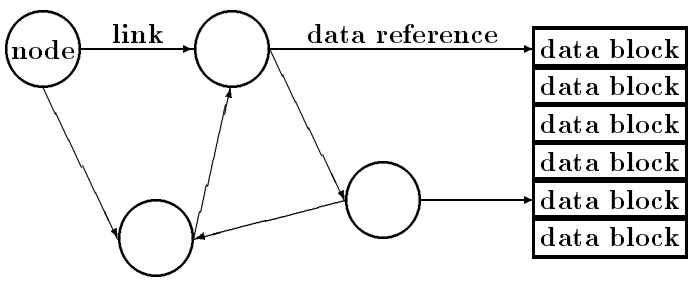


Figure 2.5: *A simple model of the HyperBase structure.*

The model (figure 2.5) of the structure is a bit to simple to describe our HyperBase, since the model only shows the nodes, and doesn't show that also links can contain information, e.g. about owner. One way to extend this, is to make a general node containing information about both the nodes and the links. But this construction causes a lot of unused attributes in the general node. We therefore include a new node in our design called **link node**, and the previous node is renamed to **data node**, see figure 2.6. In the following figures a data node is represented by a circle, and a link node by a square.

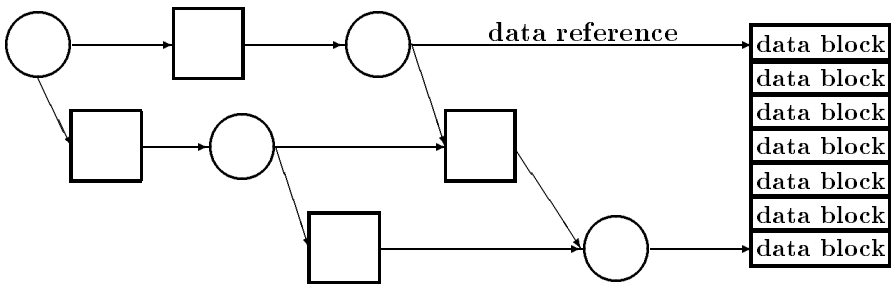


Figure 2.6: *A model of the HyperBase structure showing the different kinds of nodes (data node and link node).*

The idea that a link node is carrying informations about which data node it points to, is essential when we are talking about the type of graph we are choosing. A link node which are nondirected, have to carry information about which data nodes it is pointing to, and all the necessary key fields to describe both the forward and the backward nodes. In this way the link node would be unnecessary huge, and very hard to administrate, see figure 2.7

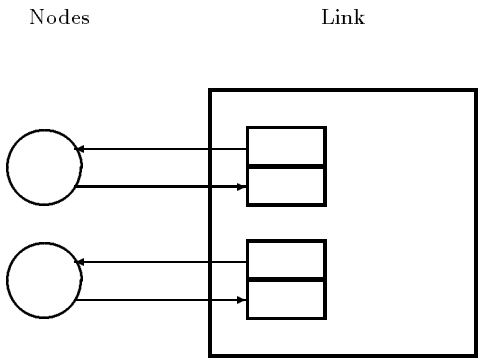


Figure 2.7: A link node in the nondirected graph.

Having made these decisions about the basic entities, we need to decide how the link and data nodes can be interconnected. There are four different ways to have references between nodes (both link and data nodes). These are shown in figure 2.8.

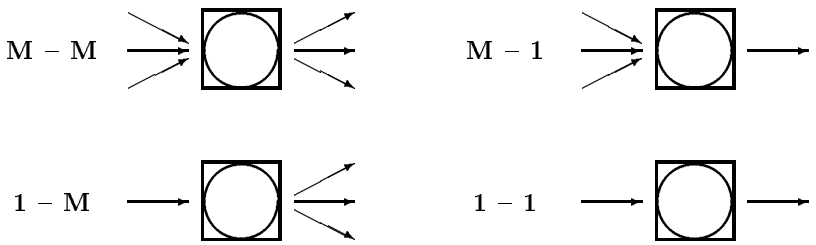


Figure 2.8: *Four different ways of having references between nodes (both data and link nodes).*

The multi to multi (M–M) is the most general way of having references between nodes, and of cause the one to one (1–1) is the most specific way – this is also shown in figure 2.9.

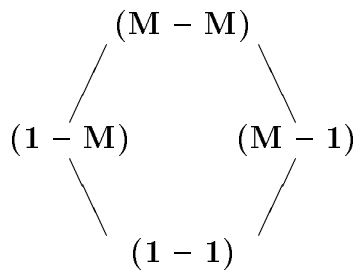


Figure 2.9: *The hierarchy of the four different ways of having references between nodes. (M–M) the most general and (1–1) the most specific way.*

In our design we have chosen that data nodes can have (M–M) references. This solution is obvious, because it's the hole idea of a HyperMedia graph to have the possibility of many references to the same node (data node), and to have many references to other nodes (data nodes) from a node (data node). Regarding the link nodes, we have chosen to have (M–1) references. This choice is made, because it makes it simpler for the user to find the next data node. With (M–M) references in link nodes the user first has to decide, which link he wants to follow, then he has to decide which of the data nodes the link points to, he wants to operate on. In a (M–1) link node the user only have to decide what link he wants to follow. The (M–1) construction also have the advantage, that it can save quite some "move link operations" as shown in figure 2.10.

All references to data node **i** are moved with just one "move link operation" instead of a "move link operation" for each data node having references to data node **i**. If the need for a (M–M) link should arise for a HyperMedia application, it is possible to build a (M–M) link node reference from the already existing "constructors". This is done (see figure 2.11) by letting the (M–1) link node point to an empty data node. This data node can, as explained earlier, have references to many link nodes.

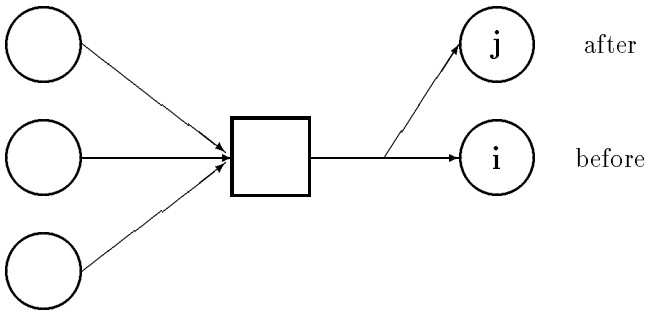


Figure 2.10: *The advantage of a (M–1) link node reference over a (1–1) reference. All the references to node i is move to node j with one operation.*

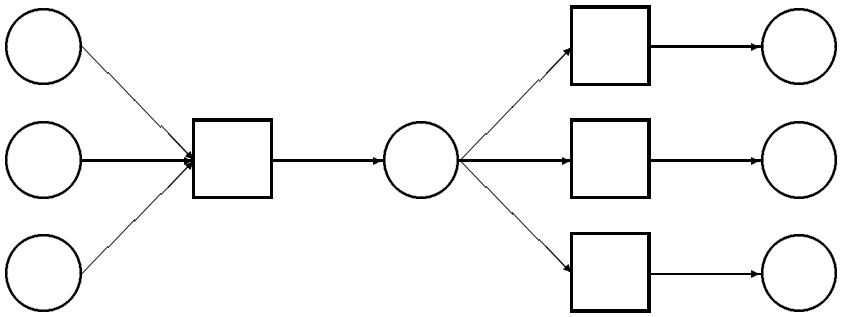


Figure 2.11: *Constructing a (M–M) link node reference from the existing constructors.*

2.3.2 Key fields in link nodes and data nodes

It is one of the features in the HyperBase that the user decides for himself what keys he wants in the entities. But some keys are used by the HyperBase itself, and has to be there. These keys are described in this section.

The node number

Every entity has a number, that uniquely identifies it. For security reasons it is decided that there isn't a link node and a data node that has the same number. In link nodes the key is named **link node number**, in data nodes **data node number**.

Number of references to a node

In order to ensure that there isn't any entity, which has references to a non existent entity, every entity has a key that holds the number of references there is to the entity. An entity can only be deleted when this number is 0. In link nodes this key is called **use count**, in data nodes it is called **links to me**.

Data in data nodes

Data are stored in blocks outside the data node. In a data node there are keys, **block pointers**, that contains pointers to the blocks that contains the data. The user decides for himself how many **block pointers** he wants. The size of the data is held in the key **size**. In addition to direct pointers to blocks there are two keys, **data block pointers** and **double data block pointers**. **Data block pointers** points at a data block that contains block pointers, and **double data block pointers** points at a data block containing **data block pointers**. This structure, that is inspired by UNIX, gives a flexible system, that efficiently stores both large and small volumes of data (see figure 3.3).

References

Link nodes has only one reference to a data node, this reference is held in the key **to data node number**. Data nodes can in theory have any number of references to link nodes. Therefore it is decided that references to links can be stored in the same way as data. That is, in a data node there are 3 kinds of keys that are used to hold references: **link numbers**,**data link pointers** and **double data link pointers**.

In figure 2.12 the entities are shown.

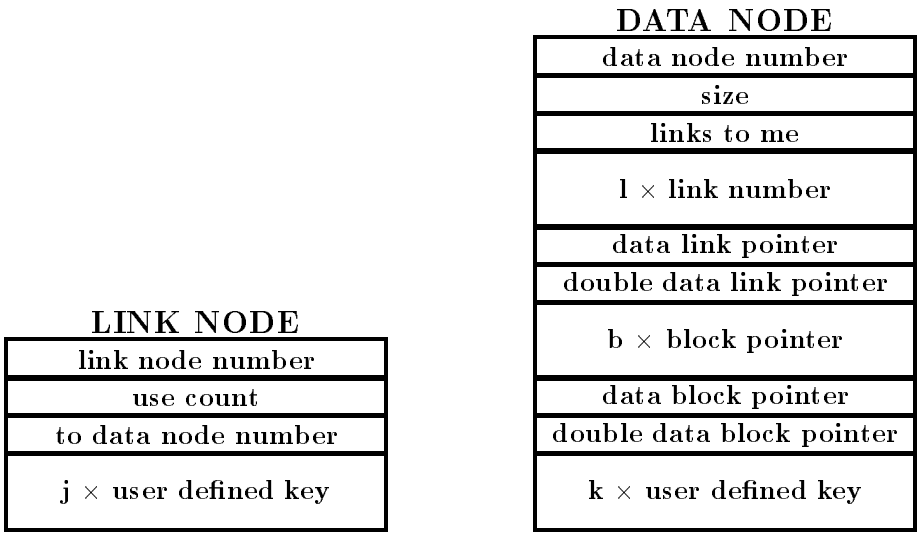


Figure 2.12: *The structure of data and link nodes.*

Both data and link nodes can contain a number of user defined attributes – here called *optional* attributes. Examples of these could be:

- *Owner*, which could be used by an application concerned with ownership of data and link nodes.
- *Date*, to keep information of the construction date.
- *Version*, which could be used by an application concerned with version control.

- *Name*, if the node should have a name, for some reason.

2.3.3 File organization

It would be obvious to save each entity in the HyperBase in a separate UNIX file, but this isn't a bright solution, because the server will spend to much time opening and closing files. It's relative slow to open and close files compared to searching in a single file. Therefore the HyperBase data is saved in 3 files. One file for data nodes, one for link nodes, and finally one for data blocks. With just 3 files, it is possible to open the files and keep them opened while the server operates. This solution also minimizes the search time for entities and data blocks, because the elements of each of the three files have the same size.

2.4 Basic services

The basic services provided by the HyperBase can be divided into four different types of commands. Commands concerning:

1. Making the HyperBase structure.
2. Reading and writing in the HyperBase.
3. Event and lock handling.
4. Connection and disconnection.

In the following all the commands provided by the HyperBase will be described. All the commands returns an integer, which reveals whether the operation succeeded or failed. If the operation failed the value of the integer will describe what went wrong.

2.4.1 Structure commands

The following commands are available when making the HyperBase structure:

CreateNode(ent_no) creates a new data node. The function returns the number of the new data node in **ent_no**.

Delete(ent_no) deletes the node (data or link node) with number **ent_no**.

Link(from_ent_no,to_ent_no) has two different functions. If the second parameter is a data node number, then **Link** creates a new link node and a reference from the new link node to the specified data node (to_ent_no), and returns the number of the new link node in **from_ent_no**. Else **Link** creates a reference from a data node (from_ent_no) to the specified link node (to_ent_no).

MoveLink(link,new_node) moves the reference in the specified link node (link) to the specified data node (new_node).

RemoveLink(from_ent,to_ent) removes the link in the specified data node **from_ent**, which points at the specified link node **to_ent**.

With this set of functions, it should be possible to make all kinds of operations on the HyperBase structure. To move the reference in a link node requires exactly one operation, the MoveLink operation. To move one of the references in a data node requires two operations, first a RemoveLink operation, and then a Link operation. This difference arises because there can be more than one reference to link nodes in a data node (link nodes only have one reference to a data node), and in the MoveLink operation you need to know exactly which of the references that is going to be redefined, so this operation can't be used for data nodes.

2.4.2 Read and write commands

There are four commands to read and write in the HyperBase, the operations are universal in the sense that they operates on all keys in the entities. That is, it will not be possible to use the write operations on the system keys, such as the entity number.

- Write(Type,ent_no,key,value)
- Read(Type,ent_no,key,value)
- EWrite(Type,ent_no,value)
- ERead(Type,ent_no,value)

The first two commands operates on fields in the entities, while the last two works on whole entities. **Type** declares whether the operation should be performed on a data node or a link node. **ent_no** is a number which unambiguous determines an entity. The type is only used as an extra error check, and to mark clearly to the user what operation is made. **key** is one of the keys (attributes) in the entities (data or link node). **value** is the address of the data which will be transferred in a write, and a pointer to the stored data in a read operation.

2.4.3 Event and lock commands

It is possible for a user to subscribe to events, if he wants to be informed about certain actions made in the HyperBase. It should be possible for a user to know when an operation is performed on a certain entity or a certain key in an entity. The user can get this information by using the event mechanisms that supports these special desires. These features are provided by the HyperBase by giving a set of simple commands. The commands can then be used to build a stronger event mechanism if necessary. The event handling commands are:

Event(ent_no,operation,key) The user wants to subscribe to events on a specific operation on the entity or a specific key in the entity.

UnEvent(ent_no,operation,key) The user wants to unsubscribe to a previous subscribed event.

ShowEvent(UsersId,ent_no,operation,key) Gives a list of the clients (**UsersId**), who has events on the specific operation on the entity or the specific key in the entity (**ent_no**).

It's not enough to have the possibility to subscribe to, unsubscribe to, and show events on entities. To be able to use the event mechanism, you must have a way to read the events, that are coming back from the server. There a two different ways to ask for events:

- GetEvent(UserId,time_limit,ent_no,operation,key)
- ReadEvent(UserId,ent_no,operation,key)

GetEvent waits for an event to arrive from the server. In the **time_limit** field it's possible to specify, the number of seconds to wait for an event to arrive. If **time_limit** is set to 0, the function will wait until the arrival of an event, or forever if no events arrives. The function returns the UserId of the user that has performed the operation on the entity (**ent_no**) or a key in the entity. ReadEvent gets the first arrived event from the server, if no events has arrived the function returns with empty arguments, else the function returns the same arguments as GetEvent.

To make sure that it will be possible to keep the HyperBase consistent in the multiuser environment, there are given 3 commands to handle locks on the entities:

Lock(ent_no) The client wants an entity locked.

UnLock(ent_no) The client wants to unlock an entity.

ShowLock(UserId,ent_no) Shows the client (**UserId**), if any, who has locked the specific entity (**ent_no**).

When an entity is locked, no other user can perform read or write operations on the entity.

2.4.4 Connection and disconnection commands

The following commands are available when a user wants to connect to or disconnect from the HyperBase:

- Connect(server_name,user_id);
- Disconnect();

When a client connects to the HyperBase server, the server name (machine – e.g. "dirac") and a user identification (e.g. user_name@machine – "kock@banach") is required. The user name is only used to identify the user for other users in event operations, so he can call himself what he wants to. A disconnection from the server requires no arguments.

2.5 The configuration file

The use of the configuration file in the HyperBase, will be based on the following method, as mentioned in section 2.1:

1. The application programmer writes a configuration file in the HyperBase Language.
2. The configuration file is used as input to a HyperBase Language parser, which generates a C++ header file.
3. The header file is included in the HyperBase code, and the HyperBase is compiled.
4. The HyperBase structure is static after the compilation.

The following HyperBase parameters can be user specified (see sections 2.3.2 and 2.3.2):

- Optional attributes in data nodes (key,value).
- Number of direct links in data nodes.
- Number of direct block pointers in data nodes.
- Optional attributes in link nodes (key, value).
- Max. size of user defined attributes in nodes (key, value).
- Size of data blocks in the UNIX data block file.

In the following section a simple HyperBase Language will be designed.

2.5.1 Design of the HyperBase Language

The HyperBase Language should only exist of a few keywords, so that it's easy to design, easy to remember (read use) and of course easy to parse. A list of the HyperBase parameters that can be user specified is shown in the previous section. The keywords in the HyperBase Language will be chosen with these parameters in mind.

A configuration file consists of a *must* part and an *optional* part. Parameters such as the number of *Link number*'s and *Block pointer*'s, and the size of data blocks should be in the *must* part, and the optional fields in the link nodes and data nodes can be in the *optional* part. If there is specified an optional key in either a link or a data node, the size of the value field should also be specified. The structure of a configuration file is shown in the following example:

```
CONFIG

  set number of direct links in datanode to <integer1>
  set number of direct blockpointers in datanode to <integer2>
  set number of direct links in linknode to 1
  set size of datablocks to <integer3>

OPTIONAL

  include key <identifier1> in datanode
  include key <identifier2> in linknode
  set size of <identifier1> to <integer4> in datanode
  set size of <identifier2> to <integer5> in linknode
```

In the above example all the HyperBase Language keywords are correctly used and are shown in their proper context.

Chapter 3

Implementation

This chapter describes how the HyperBase is implemented, that is how the overall structure of the server and the client library is made. The chapter is divided into seven sections; first there is an introduction which describes how the HyperBase is divided into five separate blocks. The next five sections describes the implementation of each of the five blocks in detail. The last section is about the configuration file.

3.1 Introduction

In order to simplify the implementation, the HyperBase is separated in functional blocks. When the actual code is to be written, there is a number of separate blocks with known functions and known interfaces. With such stand alone blocks it is relatively easy to write the code, and later test it by making stubs/drivers at each side of the implemented block. Each block are implemented as a C++ class. In figure 3.1 the HyperBase blocks are shown.

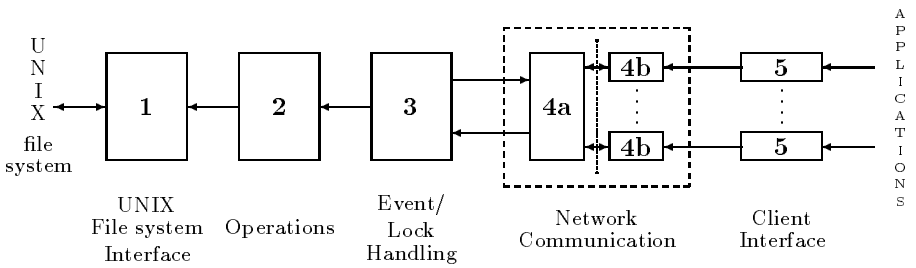


Figure 3.1: *The HyperBase divided into five logical blocks*

As seen in figure 3.1 the HyperBase server is block 1 to 4a, this part is running as a single process at the server computer. Block 4b and 5 is the client library, that will be a part of an application program (e.g. a HyperText application), running at each client computer. The blocks 1 and 2 are user independent, that means that these blocks do not care about which client who is asking for the service. Block 3 and 4a however are user dependent. The dotted line between block 4a and 4b illustrates the physical network, e.g. an Ethernet.

The five blocks have the following functions and interfaces:

Block 1 The function of block 1 is to take care of the interface between the UNIX file system and the rest of the HyperBase. The block provides a set of low level functions that operates on the UNIX file system. This means that the rest of the blocks do not operate directly on UNIX files, but uses the functions in block 1. Block 1 have to keep track on which entities and data blocks that is in use and which that are free. To make the searching for entities and data blocks efficient, it is necessary to built search trees in the computers main memory. The interface between block 1 and block 2 consists of the following low level operations:

OpenE(ent_no, **place)

CloseE(*place)

OpenNewE(Type, **place)

DeleteE(ent_no)

OpenData(**data)

CloseData(*place)

OpenLink(**place)

CloseLink(*place)

OpenE() opens an entity and places it in a buffer in memory. **CloseE()** updates the entity by writing the modified buffer to the physical file. If an **OpenE()** operation is made before a **CloseE()** the content of the buffer is overwritten. **OpenNewE()** returns a new buffer containing an empty entity of the specified type (data or link node). **DeleteE()** is used to remove an entity. **OpenData()** reads the data from the files, and **CloseData()** writes the data back to the files. The link operations works in a similar way.

Block 2 This blocks main function is to provide the operations on entities. The operations can be divided into two groups: operations on the HyperBase structure, i.e. primitives that create/change the structure of the HyperBase (delete, link, movelink, etc.), and operations on entities, i.e. functions such as read and write. The interface between block 2 and block 3 consists of the following operations:

Write(ent_no, *key, **value)

Read(ent_no, *key, *value)

EWrite(ent_no, **value)

ERead(ent_no, *value)

Delete(ent_no)

CreateNode(*ent_no)

Link(*ent_no, Toent_no)

MoveLink(LinkNo, NewNode)

RemoveLink(ent_no, Link)

These functions is described in sections 2.4.1 and 2.4.2.

Block 3 All events and locks is to be handled by this block. The block builds tables keeping information about who has subscribed to the different operations and who has locked the different entities. When an operation is performed on the HyperBase, block 3 sends events to those clients, who has subscribed to this operation, telling who performed the operation and what the operation was. The interface between block 3 and block 4 consists of the operations from block 2 and the following operations, which is special for event and lock handling:

Event(UserID, ent_no, operation, key)

UnEvent(UserID, ent_no, operation, key)

ShowEvent(UserID, **UsersID, ent_no, operation, key)

Lock(UserID, ent_no)

UnLock(UserID, ent_no)

ShowLock(UserID, *UserId, ent_no)

These extra operations are also described in section 2.4.3. This interface is special in the sense, that block 4a also provides an operation to be used in block 3:

SendEvent(UserID, ExecUserID, ent_no, Operation, Key)

This operation is used by block 3 to send events to the users, that has subscribed to operations in the HyperBase.

Block 4 This block has to deal with all network communication, i.e. this blocks main function is to make an abstraction from the "real" network, so that the network is transparent for the other blocks. If this implementation of the HyperBase is to be moved to another network architecture it is only this block that has to be rewritten. The interface between block 4b and block 5 consists of the operations from block 2, block 3 and the following operations:

Connect(*serverhost, *username)

Disconnect()

GetEvent(**ExecUserID, Timelimit, *ent_no, *Operation, *Key)

ReadEvent(**ExecUserID, *ent_no, *Operation, *Key)

These operations are described in section 2.4.4.

Block 5 This block provides the user interface, so the function of this block is to check both the syntax and the semantic of the operations, that the user wants to carry out. By doing this block 5 is able to make the network communication as minimal as possible, which will make the HyperBase faster.

The last "block", which is not a block like the others, is the configuration program. This program creates the hb_config.hh and hb_config.cc file, by reading a user written specification. In the specification the user can tell how he wants to shape his HyperBase.

In the next five sections the implementation of the five blocks is described in more detail. The last section of this chapter contains a description of the implementation of the configuration program.

3.2 UNIX File System Interface, block 1

Block 1 provides the interface between the rest of the HyperBase and the UNIX file system. Therefore block 1 manages three different files – one for data nodes, one for link nodes, and finally one for data blocks. These files are named *datanodes.hb*, *linknodes.hb*, and *datablocks.hb* and are located in the directory in which the server is started. By manage we mean keeping informations about locations (addresses) of used parts and unused parts of the files, so that we are able to find used parts (retrieve data) and to reuse deleted parts of the files (store data).

The file management for both the data nodes and the link nodes are implemented as follows:

- There is a balanced search tree in memory, keeping the node number and the address of the node in the belonging file. So when a node is to be opened, the node number is looked up in the search tree and the file address is returned. We use balanced search trees, because the search complexity for this kind of trees are as low as $log_2(n)$.
- There is a single linked list in memory, keeping addresses of unused parts of the file. We use a single linked list, because we don't have to look for specific addresses. When we make a look up in the list, every unused file address is as good as the others.

When a node is deleted, the number and the address are removed from the balanced tree (and if necessary the tree is rebalanced), and the address is inserted at the end of the single linked list. Then the node number on the file is set to zero. The number zero indicates that this part of the file is momentarily unused. Likewise, when a new node is created, the first address (the head) is found and removed from the single linked list (or if there is no unused parts in the file, the end of file address is found), and the node number and the found empty file address are inserted in the balanced tree, which might cause a rebalance of the tree. Then of cause the new node is written to the file.

To keep information about which node numbers (both data and link nodes) that are used (remember, nodes have unique numbers), a bit string (bit map) is used. Data nodes are given **even** numbers, starting from number 2, and link nodes **odd** numbers, starting from number 1 (the initial node). Bit number 0 is always set, because this is used to indicate unused file parts. When a node is created the next even (in case of a data node) or odd (in case of a link node) unset bit is found and set, and the belonging number is returned. When a node is deleted the belonging bit is unset (set to 0).

The file management for data blocks are a lot easier, because all we need is a bit map, keeping information about which blocks that are used and which blocks that are unused, and a single byte at the beginning of each data block in the file, telling whether the block is used (1) or unused (0). So when the user specifies that he wants a data block size of, say 64 bytes, then the actual file block size is 65. This is of cause made transparent for the user.

The files, the balanced search trees, and the single linked lists are implemented as classes in C++, with general member functions. The special use of these classes are obtained by using inheritance to more specific classes, like class uselist, usetree, nodefile, linkfile, and datafile. Figure 3.2 shows the structure of the class Filehandler, which provides block 1's member functions mentioned in the introduction of this chapter.

Class Filehandler operates on different instances (objects) of the specific classes (uselist, usetree, ...). The constructor of class Filehandler takes care of the initialization of the balanced search trees, the single linked lists, the bit maps, and the files by calling a function InitFileHandler(). As an example, the data node file is read from one end to the other, and the numbers and addresses of used parts ($number \neq 0$) are inserted in the data node tree, the belonging bits plus bit 0 in the number bitmap are set, and the addresses of the unused parts ($number = 0$) are inserted in the data node list.

The initialization of the link node tree, link node list, and data block bitmap are made alike.

3.2.1 Storing and retrieving data and link numbers

As mentioned in the design chapter the structure of the data area and the link number area in data nodes are inspired by the structure of I–nodes in UNIX. The structure of the data node with direct pointers, indirect pointers, and double indirect pointers are shown in figure 3.3. In this

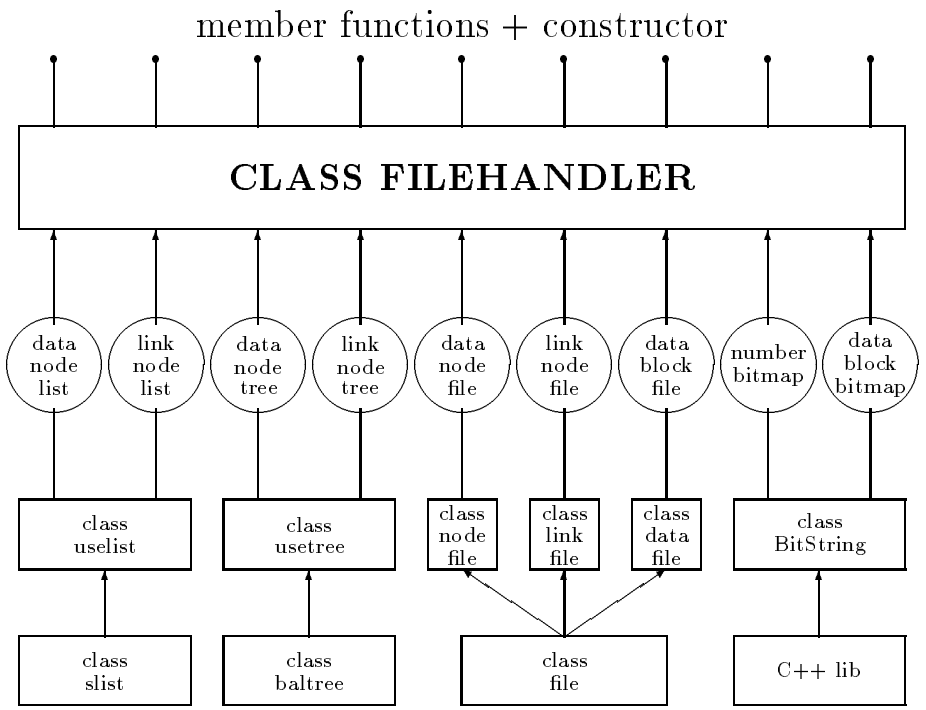


Figure 3.2: *The inheritance structure of block 1. In the figure, the boxes are classes and the circles are instances of these classes.*

chosen example the data node contains two direct link numbers and two direct data pointers, and the data block size is set to 8 bytes.

The size of data blocks and the numbers of direct data pointers (called blockptr[i]) and direct link numbers (called linknum[i]) in data nodes, can be user defined along with the user keys in both data and link nodes. If we have l direct link numbers, b direct data pointers, and a data block size of s bytes, a data node can contain:

$$linknumbers = l + (\frac{s}{4} \cdot \frac{s}{4}) + (\frac{s}{4} \cdot \frac{s}{4} \cdot \frac{s}{4}) \tag{3.1}$$

$$datasize = (b \cdot s) + (\frac{s}{4} \cdot s) + (\frac{s}{4} \cdot \frac{s}{4} \cdot s)\ bytes \tag{3.2}$$

4 is the *sizeof(long)*. "Longs" are used both to contain link numbers and addresses of indirect and double indirect data blocks. If we have e.g. 10 direct link numbers, 10 direct data pointers, and a data size of 64 bytes, the data node can contain:

$$linknumbers : 10 + (\frac{64}{4} \cdot \frac{64}{4}) + (\frac{64}{4} \cdot \frac{64}{4} \cdot \frac{64}{4}) = \underline{4.362} \tag{3.3}$$

$$datasize : (10 \cdot 64) + (\frac{64}{4} \cdot 64) + (\frac{64}{4} \cdot \frac{64}{4} \cdot 64) = \underline{18.048\ bytes} \tag{3.4}$$

Special routines are made to store and retrieve data and link numbers from this data node structure. These routines automatically allocates the needed new data blocks or deallocates remaining unused data blocks. An error is returned if there is an attempt to store more data or link numbers than the data node can contain.

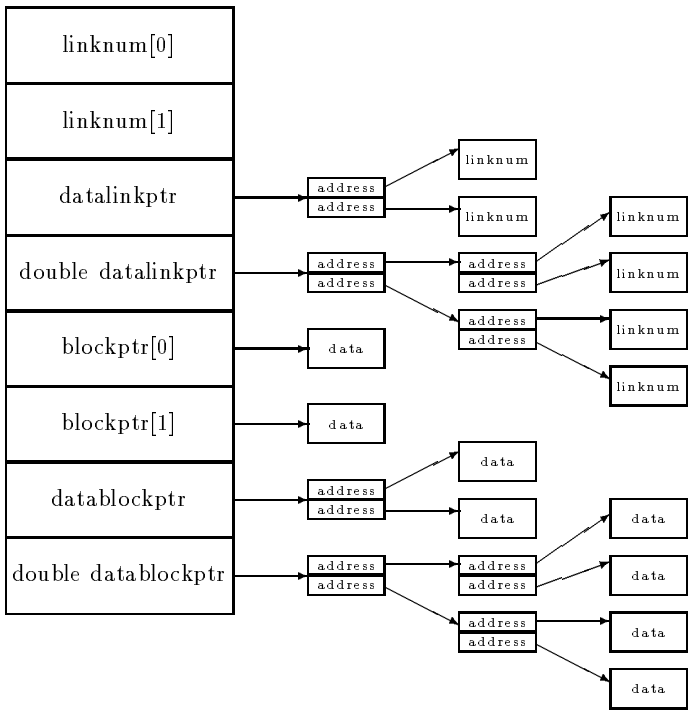


Figure 3.3: *The structure of the data area and link area in data nodes.*

3.3 Operations on the HyperBase, block 2

In this block of the HyperBase the different operations on the HyperBase are implemented. The way this part of the program works is to open/make entities in the HyperBase by calls to block 1, then make the changes on the entity and afterwards put it back with a new call to the block 1. This block is implemented in one single class where the public functions are the operations given to the user.

The functionality of this class is to give the user some HyperBase operations which keeps the base consistent, i.e. keeps the pointers and the pointer counters up to date. To ensure this consistency it will not be allowed for the user to change all the entity keys directly. Figure 3.4 illustrates how the pointers and counters are correlated.

The different public procedures will shortly be described in the following, because it is the most basic operations on the HyperBase:

Delete: Deletes an entity. If the entity is a link node, the data node it points to, will have the "links_to_me" counter decremented. If the specified entity is a data node the counter updating gets a little more complicated. In a data node there can be several pointers to link nodes, and all these link node shall have their "use_count" decremented.

CreateNode: Creates a new data node by a call to the Filehandler object, and returns the data node number.

Link: Creates a pointer between two entities and updates the necessary counters. If the entity, the pointer will point to, is a data node, there is created a new link node pointing to that

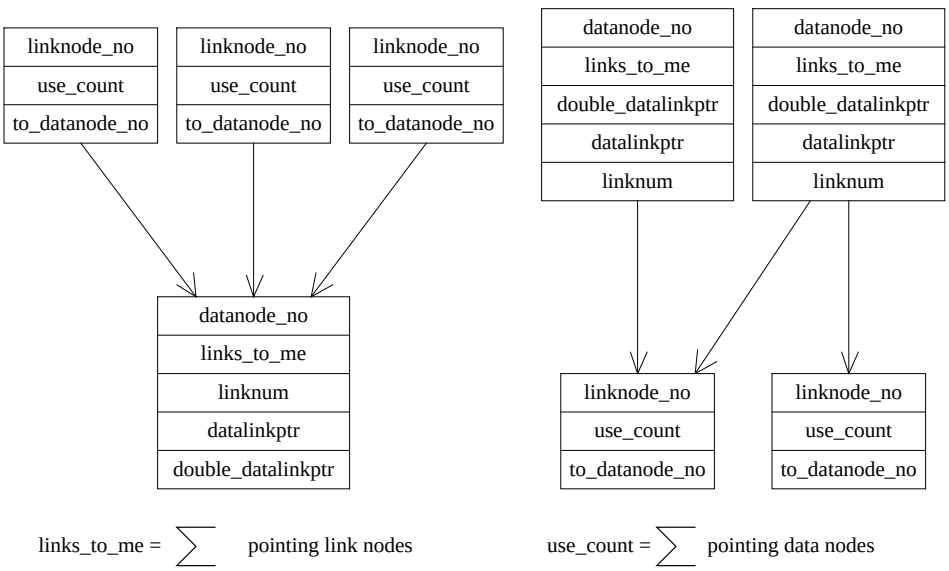


Figure 3.4: *This figure shows how the pointer counters and pointers are correlated. The structures are simplified and shows only the system keys important for this block. The 2 equations below the figures must always be true.*

data node. This is in fact the only way to create a link node. If the entity, the pointer will point to, is a link node, the pointer value will just be inserted in the data node.

MoveLink: Moves the pointer in a link node to another data node, and updates the counters in the involved data nodes.

RemoveLink: Removes a pointer in a data node, and updates the "use_count" in the pointed link node.

Read: Reads the value of the specified key in the given entity.

Write: Writes a value to a key field in the specified key. It will only be allowed to write directly to the user defined fields and the "DATA" field.

ERead: Reads a whole entity, except the "DATA" and "LINK" field.

EWrite: Writes the user defined keys into an entity read by "ERead".

3.4 Events and Locks, block 3

The function of block 3 is to take care of all operations concerning events and locks. It has to "listen" to all operations that has to be passed to block 2. If anyone has subscribed to that operation, then the event is send to the subscriber and block 2 is called.

To make sure that there is no locks or events if the server crashes, all locks and events are stored in the computers memory, that is, not in the entity in the physical files.

The implementation of the block is divided in to 3 parts, see figure 3.5.

Main part Handles the communication with block 2 and 4 and calls the Event and Lock part when necessary.

Event part Handles all concerning events.

Lock part Sets and unsets locks, and checks if the entity is locked.

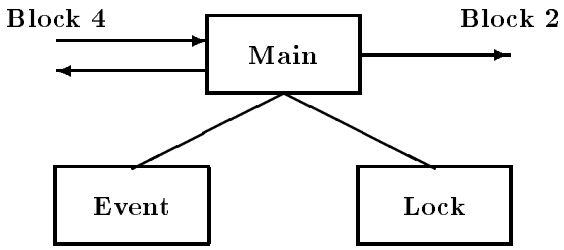


Figure 3.5: *Block 3 implementation.*

3.4.1 Main part

All operations that affects block 2 (read, write etc.) is first checked to see if the node is locked. This is done by using the functions defined in the Lock part. If it is not locked, a check is done to see if there is anyone that has made a subscription on this event. If this is the case, block 4 is activated, and the event is send to the user, who had subscribed to that event. When there is no more events to send, the operation is passed to block 2.

3.4.2 Event part

To set, unset and check for events, a balanced tree and a variable (AllNodes) is used, see fig.3.6.

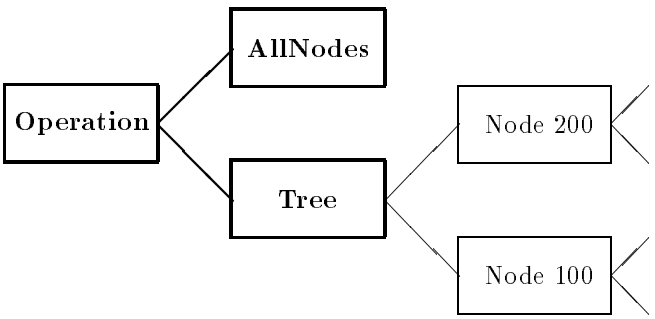


Figure 3.6: *Events for each operation.*

The AllNodes variable is used if the event is on all nodes. If it is a specific node, the balanced tree is used. Every node in the tree and AllNodes is a bit matrix of users and keys. The bit is set if there is a event on that user and key.

3.4.3 Lock part

The nodes that is locked is placed in a balanced tree. This means that it is easy to lock, unlock and check for locked entities in the tree. In every node in the tree, the number of the user that has locked the entity is placed.

3.5 Network Communication Block, block 4

The main function of block 4, the Network Communication Block, is to provide the ability to communicate with more than one client, you might say that it is in block 4 the HyperBase becomes a multiuser HyperBase. In the blocks 1 through 3 you can not talk about a multiuser HyperBase, more a user dependent and a user independent part.

The HyperBase is designed to be used in a computer network, that means that one computer in the network is the HyperBase server, and the clients can be located on both the server computer and the rest of the computers in the network. The way to communicate between the server and the client(s) is by using the computer network. The UNIX operating system designed at University of California at Berkeley, the BSD4.3, supports a way to communicate between computers using *sockets*. UNIX versions developed upon the BSD4.3 also includes the socket abstraction. At the University of Aalborg a computer network of Sun workstations running SunOS 4, which is based on the BSD4.3, is available. The implementation of block 4 will be done on these Sun workstations.

3.5.1 The function of block 4

As seen in the figure 3.1, block 4 has to communicate with block 3, at the server, and a block 5 at each client. Block 4 can be divided in two blocks, one on each side of the network; one block, 4a, placed in the server, and one block, 4b, placed in each client.

The function of block 4a is:

- If there is no incoming messages, the server goes to sleep. When a message is comming, the server wakes up, examines who "called", and reads the message. This sleep/wake routine will be the main loop of the server.
- Unpack the incoming messages, and "pull" in the right function with the right parameters i block 3. Upon return from the function call in block 3, send the data, entities, values, etc. back to the client, by first packing the message and then sending.
- Upon a call from block 3, send the event to the client specified by block 3, after sending, return to block 3 without waiting for the client to read the event.

In block 4b the function is:

- Take care of a connect (disconnect) call, that is to make the connection with the appropriate HyperBase server.
- When block 5 is calling a function at the server, pack the function, with the right parameters, and send it. Wait, and then buffer the reply (data, entities, event, etc.).

- If block 5 is calling the function GetEvent, block 4b will sleep until an event arrives or a specified timelimit has been exceeded. The ReadEvent does not sleep, it is supposed to be used as a polling function.

To take care of the communication there are used 3 sockets, one for reading, one for writing, and one for events. One more socket is used for connection. When a client wants to connect, he creates a socket with a unique number, known by all the clients (this number is specified in the configuration specification), and waits for the server to give him his 3 socket numbers. After receiving the 3 numbers he disconnects from the unique socket, so that other clients can connect to the HyperBase. In the connect call the client specifies on which physical computer the server is running (e.g. "aiken") and the user name of the client (e.g. "kasper@church"). The user name is only used when clients gets informations about other users. When getting events from the server, the client gets the user identification (e.g. "kasper@church" or "the first man on the moon") of the user who made the operation. The server identifies the clients by a number, and associates that number with the user name string by maintaining a list.

3.5.2 The classes of block 4

There are two classes in block 4a: one named "hb_client", which is the class where the selection of functions take place, and a class named "hb_block_4a", which takes care of the send event calls from block 3.

Block 4b contains one class, named "hb_block_4b", where all the functions to the HyperBase is. These functions take care of the network communication by calling the server, and waits to get the answer from the server.

3.6 The User Interface, block 5

The purpose of block 5 is to stop all operations on the HyperBase, that doesn't make sense, before they are send trough the network. Block 5 will only check the redundant information, and not make instance check, e.g. if a node really exist or not. Examples of redundant information, are that you can tell what kind of entity you have, if you have the number of the entity. This is possible because links have odd numbers and nodes even. You can also identify an entity on what kind of key fields it have, and in some operations the user have to specify what kind of entity, he is working on. Block 5 checks the following things:

- if an operation exists (in event operations).
- if a key exists.
- if a key field exists in the type of entity in question.
- if an operation is legal on the entity in question.
- if the user connects to the same HyperBase twice.

In order to keep track on which HyperBases the user is connected to, all instances of block 5 have a common list (a class variable) with all the HyperBases the user is connected to. The function Connect, calls a function newBase which checks if the HyperBase already is opened by the user, if not the base is inserted in the list and the connect call is passed to block 4b. Elements in the

list can be removed by the function removeBase, which is called by Disconnect, or by block 5 if the user forgot to disconnect from the HyperBase before deleting block 5.

The pointer Baselist is a class variable that points at the global list, every instance of block 5 have a pointer thisBase that points at their own element in the list.

3.6.1 C-interface

Because it should be possible to use the HyperBase in normal C-programs, block 5 also has a C-interface as an extension. The C-interface has a function for every public member function in block 5 (except constructor and destructor of course). These functions all have the same parameters plus one more. This extra parameter is a number used to identify each HyperBase.

This C-interface uses a list, where each element contains a pointer to a block 5 and a number associated with the base in question. All functions, calls getBase with the hb-identifier and gets a pointer to the correct instance of block 5. The list is maintained by connect and disconnect. The only check the C-interface makes is to prevent the user to use the same number to identify HyperBases twice. The rest is still taken care of by block 5.

3.7 The configuration program

The configuration program *hb_config* is implemented as a general Context Free Grammar, by the use of the compiler tools lex and yacc. *hb_config* builds a symbol table, which is used when generating the output files. Besides from the tests done in the grammar, some further checks are made by going through the symbol table and checking that the values seems appropriate. The code which maintains the symbol table and performs these last checks and generates the output files is stored in the file symbol.c. The lexer and parser is in the files config.y and config.l. The most important detail to recognize is that the numbers used in the grammar must be integers, and the identifiers must be a sequence of lower or upper case letters followed by any mix of letters and numbers. The names used in the implementation of the structures of data and link nodes cannot be used as identifiers, since that will result in C++ code which can not be compiled.

Chapter 4

Conclusion

In this chapter we will draw conclusions on the work that has been done, described in the previous chapters. We will also evaluate the implemented HyperBase server from a users point of view.

4.1 Design considerations

The design presented in the previous chapter, often leads to a choice between different design ideas. Some of the ideas which were rejected will be described in this section.

A major topic which was discussed very intensively in the beginning of the design, was how to make the HyperBase easy to extend and shape to a specific purpose. In particular it was discussed how attribute fields could be added to the structure of entities, and some interrest concerned the possibility of extending the number of operations available on the HyperBase, by adding new operations.

Two different approaches were discussed:

1. Implementation of a general language, which were called HyperBase Language, to support dynamic changes of the structure of entities while the HyperBase server was running
2. A static specification of the structure of entities in a configuration file, which is read at compile time. When the HyperBase server is compiled the structure of the entities is static and can not be changed

The latter of these two approaches was chosen for several reasons. Nobody had an overview wide enough to answer essential questions like:

1. Would we have time enough to implement both a general language, and the HyperBase?
2. Could we find some code from another languages, which cheaply could be hacked to our purposes?
3. Would the dynamic solution result in exceeding the deadlines for the project?

The schedule for this project has been very tight, and we wanted to be sure that we had a running HyperBase when the project finished. This combined with a general view of HyperBase as a file

system for HyperText, talked in favor of a static structure of entities. If the HyperBase today have a number of attributes, e.g. name, type, owner, group, creation date, and size, and a fixed number of operations manipulating these attributes, then it certainly will be expected that these attributes are the same tomorrow. Such a static structure is simpler and easier to maintain. Sometime in the future there might arise a need for a new set of attributes or functions. At that point, the existing HyperBase is backed up, the new HyperBase is compiled, and the HyperBase is restored with the new structure. It is a one time effort to convert the file format, and when the HyperBase is recompiled it has static entities, which is simple to handle in an *efficient* way. These considerations motivated the choice of the static specification of the structure of entities.

Another idea which evolved concurrently with the discussion of dynamic versus static structure, was the idea of typed data nodes and links. This was contrary to the notion of general nodes. It was clear to us that links and nodes were not carriers of information, but represented the structure of the data. This is also reflected in our implementation where we use three files to store all the information. One file for link nodes, one for data nodes, and finally one file for all data.

At this point we also discussed the need for a backup/restore application, and of ways to recover from crashes. These ideas were never followed up by detailed design due to lack of time.

When the structure was decided, it was time to discuss which operations would be adequate. The general structure that the HyperBase must support is a graph, so this part of the design concerned specifying a general abstract data type for graphs. There exists some mathematical algebraic specifications for graphs, but this subject was never investigated. Instead an ad hoc specification of a general graph and it's operations were made. In this phase we decided that we would not allow nodes to reference another node directly, a node must make the reference through a link. It was decided not to support links which pointed to links. This was done to keep the operations as simple as possible, which hopefully would lead to a better implementation. But the heaviest argument against these two special cases, was that they did not give any useful meaning to us.

4.2 Summary of the work

In this report the design and implementation of a basis (file system) for HyperMedia applications, called HyperBase, is described. The HyperBase is implemented as five blocks with distinct functions and interfaces. The implemented HyperBase provides facilities, that makes is useful as a tool for HyperMedia application programmers, who wants to create a special purpose HyperBase supporting exactly their needs concerning nodes, links, storing, locking, and event handling. The HyperBase is a part of a larger system, a HyperMedia system, with different application using the same special purpose HyperBase.

The implemented HyperBase can be used as it is, by writing a configuration file specifying how the configurationable parts of the HyperBase has to look like. If a HyperMedia application programmer needs facilities beyond the facilities provided by the HyperBase, these can be implemented on top of the HyperBase using the already implemented general operations on the HyperBase, which is the easiest way, or by extension of some of the five blocks that implements the HyperBase. How this can be done is written in the Manual.

4.3 Evaluation

The evaluation of the work is diveded into two parts, one concerning the design and one concerning the implementation of the HyperBase.

4.3.1 Design

Our background to the design consisted mainly of our earlier experience from the computer engineering studies, but also of reading articles and earlier reports concerning the HyperText idea. Throughout the term we followed parts of a course in data bases.

The special concepts in the design is the multi user ability, the locking mechanism, and the event mechanism. The concept of having a server taking care of the file system interface is well known and generally accepted (e.g. the Network File System – NFS). The locking mechanism is equally well known and accepted as a needed feature, when operating in a multi user environment. The event facility is a quite new but very useful concept in connection with the clients – server relations (e.g. in updating a browser).

Concerning the more specific details of the design, it can be argued that the locking mechanism is limited in the way that it is only possible to lock whole entities (nodes and links) and not certain fields in the entities. Furthermore it could be useful to have both a read and a write lock, making it possible to read write locked fields and entities. Also a special "browser" function could be useful.

4.3.2 Implementation

The programming environment, in which the HyperBase was implemented, consists of a SUN 3 and SUN 4 workstation network running UNIX (SUNOS 4) and a C++ compiler (GNU g++) with GNU's C++ library. In spite of these specific machines, the C++ source code should be portable to any machine running Berkeley UNIX.

As earlier mentioned the HyperBase implementation is divided into five blocks (see figure 3.1): UNIX File System Interface, Operations, Event/Lock Handling, Network Communication, and User Interface. This structure makes it easy to locate the individual functions in the HyperBase, which is useful when maintaining the system. If the "limited" locking mechanism should be expanded as mentioned above, then this can relatively easy be done by changes in the Event/Lock Handling block. Including extra functions in the HyperBase is described in detail in the Manual.

In the present implementation the operations `ERead()` and `EWrite()` isn't fully implemented. These functions can easily be constructed on top of the HyperBase using the operations `Read()` and `Write()`.

4.4 Future work

The present implementation of the HyperBase can be used as it is, but it would be a good idea to make the changes concerning the locking mechanism, to implement the browser function, and to implement the functions `ERead()` and `EWrite()` in the Operations block. These extra facilities would save a lot of extra work for the HyperMedia application programmers.

This implementation of the HyperBase is thought of as a basis for our graduation projects. The

idea is that the three different graduation groups in the spring term 1990, each makes a special HyperMedia application. Then these application could be assembled to a HyperMedia system.

4.5 Final conclusion

If a comparison is made between section 4.2, that sums up the work done, and the design goals listed in chapter 2, we find that these goals have been achieved. We have a developed HyperBase system with a small set of general operations, that seems easy to use and easy to extend. Finally we conclude that the developed HyperBase server, in spite of the implementation as five blocks, seems consistent.

Unfortunately our g++ compiler did not work at the end of the project. Because of this the implementation has not been fully tested, but we hope to get this done in the near future.

Bibliography

[Fischer 88] Lars Fischer
Programming Environments, Based On Hypermedia
Masters Thesis
Aalborg Universitetscenter, november 1988.

[Christensen 88] Jens Peter Christensen
Hyperbase
Masters Thesis
Aalborg Universitetscenter, august 1988.

[Tanenbaum 87] Andrew S. Tanenbaum
Operating Systems - design and implementation.
Prentice-Hall International, 1987.
ISBN 0-13-637331-3

[Engelbart 68] Douglas Engelbart
A Research Center for Augmenting the Human Intellect.
AFIPS Press, 1968.

[Bush 45] Vannevar Bush
As We May Think
Atlantic Monthly, pp 101-108
july 1945
Reprinted: Microsoft:'CD-ROM: The New Papyrus'

[Conklin 87] Jeff Conklin
Hypertext:An Introduction and Survey.
Computer,20(9):17-41, September 1987.

Appendix A

HyperMedia Application

The following appendix is an example of the use of our HyperBase. The purpose of this application is to illustrate a way to use our HyperBase. To do so, the application "Ymer" was born. The name Ymer are from nordic mythology, and is the name of the giant, who was killed by Odin and his brothes. Ymer's skull was used to make the firmament, and this firmament entwined the world, just as this application will do with our HyperBase.

A.1 Design

To design an application useful for test and verification, and perhaps finding one or two problems in the design of our HyperBase, the following criterions are important:

- Creating the structure.
- Maintaining the structure.
- Using user defined attributes in entities.
- Handling many users.
- Using more than one HyperBase in an application.
- Using the event and locking mechanisms.
- Moving data blocks of different size in and out of the HyperBase.

To fulfill these purposes we will construct Ymer as a simple graphic interface in NeWS, where it is possible to construct and maintain a HyperBase, containing a simple text file in the data key of the data node. As a tool to help the users of Ymer, a special window displaying the events from other users, e.g. constructing or maintaining the structure in the HyperBase, will be visible. Furthermore, it will be possible to subscribe to read/write in the data key in the data node, and to lock a node.

A.1.1 The primitives

To build the structure of the Ymer HyperBase, the user has three sets of primitives, one set at the main window menu (canvas, in NeWS terms), one set in the data nodes menu, and finally one set in the link node menu, this is made, because the nodes are different in there context, and behavior. First we have the main window:

New-datanode: Create a new data node at the user specified position.

Refresh: Refresh the data on the screen from the HyperBase information.

Quit: Quit the program.

When a data node is created, it will be possible to construct the whole graph structure, and use the file transporter in the data nodes:

Link to .. If the user creates a link to a data node, a new link node will be created, and the user can specify the position, then links between the three nodes are made. If the choice is a link node just one link is made between the nodes.

Move node Move the position of the node.

Delete node Delete the node.

Move link Move link from a one node to another node.

Lock Set or unset lock at node, and if the node locked by another user then show the user id of that user.

Events Same as lock, but for all keys in the node.

Transport Make it possible to transport a data key from the HyperBase to a text file in the UNIX system, and vice versa.

The link node functions is only for maintaining the structure.

Move node Same as data node.

Delete node Same as data node.

Move link Same as data node.

A.2 The program

The program will be split into two parts, one part in NeWS made in PostScript (PS), and one part made in C++. The PS part will contain all function to support the graphics, while the C++ part will be the front end to the HyperBase and the manager for the program, and will react on every event from the HyperBase and every event made by the user. These user events can be e.g. an operation to make a new node.

The connection between the HyperBase, Ymer, C++, and NeWS is shown in the class and instance hierarchy (see figure A.1 and figure A.2).

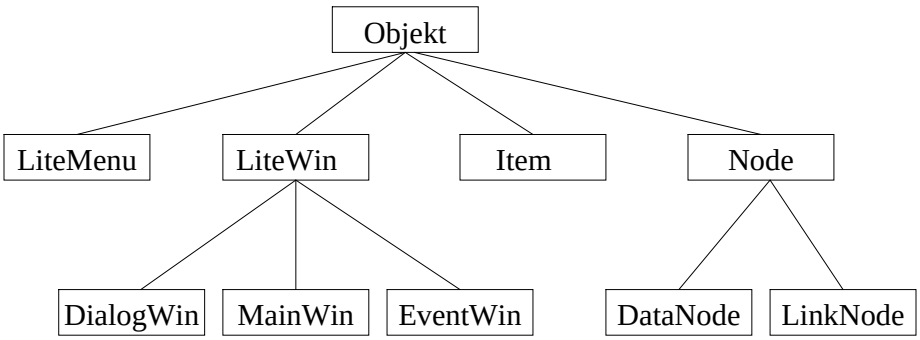


Figure A.1: *The class hierarchy of Ymer in PostScript, note that the class Item is a collection of small classes*

A.2.1 The configuration file

To make the implementation easy, the position of every node in the main window will be decided by the user with the help of the mouse. For this reason we are using two keys in every node. To help the user of Ymer we install a string to small notes in the node, which gives the following user defined configuration program:

```
CONFIG
    set number of direct links in datanode to 5
    set number of direct links in linknode to 1
    set number of direct blockpointers in datanode to 9
    set size of datablocks to 512

OPTIONAL
    include key posdx in datanode
    include key posdy in datanode
    include key noted in datanode

    include key poslx in linknode
    include key posly in linknode
    include key notel in linknode

    set size of posdx to 4 in datanode
    set size of posdy to 4 in datanodes
    set size of poslx to 4 in linknode
    set size of posly to 4 in linknode
    set size of noted to 20 in datanode
    set size of notel to 20 in linknode

    set socket to 10099
```

A.2.2 Implementation

In this section only the HyperBase commands will be given in C++. This is just a suggestion of how to use the HyperBase, all other code will be pseudo code.

The main program is just a small loop to control the events from the HyperBase and the user, and if necessary write the HyperBase event or error event to the user, in the event window:

```
Hb_interface HyperBase;
 //
 //
main();
   HyperBase.Connect(``pink'',''lucifer at white'')
     // wait at events
     // if the end
   HyperBase.DisConnect();
```

The primitives

The primitives described in this section will only be the most interesting, all the other is easy to construct, after the idea that the user makes the first move followed by an action in the HyperBase. If every thing is correct, then the event will be shown on the screen, and so forth.

Create data node:

```
// event (1), main window from user
error=HyperBase.CreateNode(&nodeid);
// wait to user have position the node at (x,y)
error=HyperBase.Write(DATANODE, node id, POSDX, &x);
error=HyperBase.Write(DATANODE, node id, POSDY, &y);
// draw data node with node id at pos (x,y)
```

Link to node:

```
// event (1) at nodeno, in data node
// select another node 'helpno'
if (helpno % 2 == 0) // data node selected
   error=HyperBase.Link(&no, helpno);
   error=HyperBase.Link(nodeno, no);
   // wait for position of new link node (x,y)
   error=HyperBase.Write(LINKNODE, no, POSLX, x);
   error=HyperBase.Write(LINKNODE, no, POSLY, y);
   // draw lines and node
else
   error=HyperBase.Link(nodeno,helpno);
   // draw line.
```

Transport file:

```
// event 7 at nodeno,in data node
// if (OK) button at dialogue window pressed
```

```
    if (direction==HB2UNIX)
       error=HyperBase.Read(DATANODE, nodeno,DATA,
                            &filepointer);
       // write file with file pointer to UNIX file
       // remove \0 from end
     else
       // read file from UNIX and give file \0 as end
       error=HyperBase.Write(DATANODE, nodeno, DATA,
                             filepointer);
// else do nothing
```

A.3 Conclusion

The Ymer was an example where the HyperBase structure was used to make a graphic display, and this feature has to have all information of the position and the position of its neighbor to its disposal. But if the user needs to protect his information in the data node by using the supported locking mechanism, this will make it impossible to draw the graph.

Another feature that the HyperBase is badly missing, is a browser to get all the node numbers. This is a were useful function, when the graph has to be redrawn from scratch.

The last problem is that it is impossible to use the same name for user defined keys in link nodes and data nodes. When storing the position of the nodes, the user defined keys need to be called (*posdx,posdy*) and (*poslx,posly*), but still have the same intention.

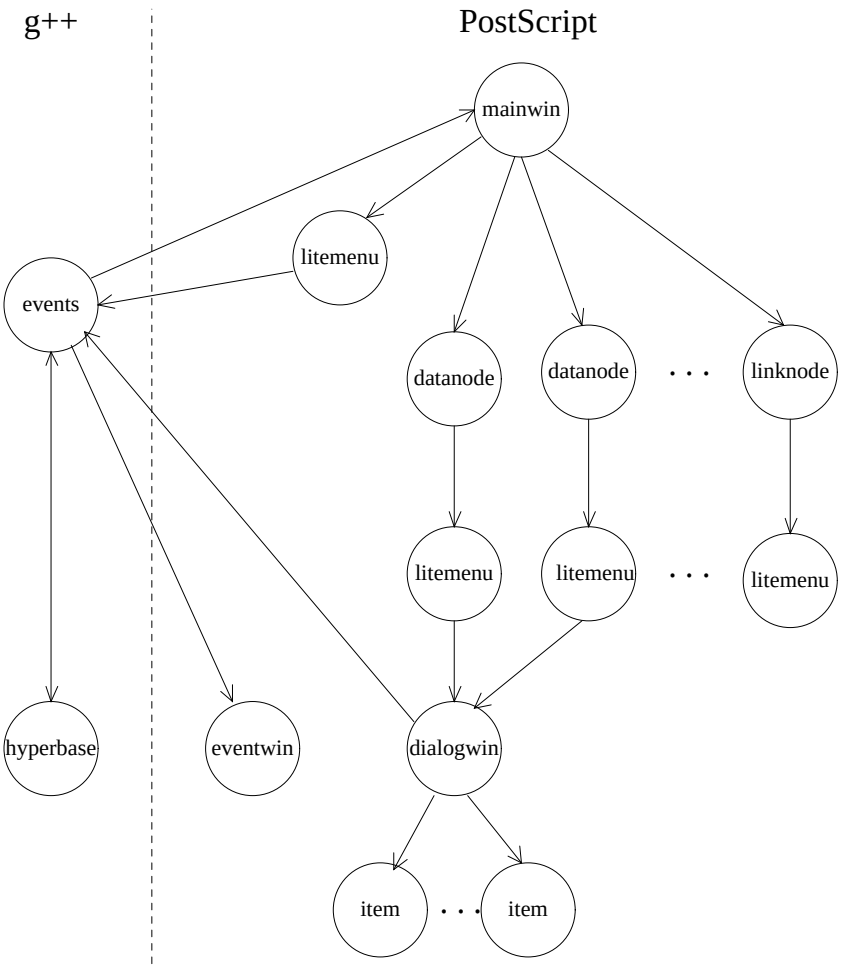


Figure A.2: *The instance hierarchy of Ymer in PostScript and C++*

Appendix B

Manual

The manual is divided into two parts, a "Users Guide" and a "Programmers Guide". The first part describes the HyperBase as it is, while the second part describes how to extend it.

B.1 Users guide

In this section the possibilities in the already implemented HyperBase will be described. This part of the manual is intended to be used by users, who will use the already existing software as it is.

B.1.1 What is HyperText

HyperText is a relative old idea (from 1945). The idea is that text has not to be organized in a linear way, as in a book (from page 1 to X), but can be handled as small pieces of text that have references to each other. When a reader looks at one of the pieces it has references to other pieces, and he chooses himself which of these references, he wants to follow to shift to the next piece of text. The benefit of the system is that the author doesn't have to decide what and in what order, the reader reads the text. This is a great benefit when the text covers a broad area, from which the reader is not interested in all information e.g. a lexicon, or a manual that has to be read by both beginners and experts. Gradually systems has been made, which do not just store information as text, but also as sound and pictures (e.g. HyperMedia programs). These systems appearance are highly different, but they all build on the old idea in HyperText: Information is stored in small chunks, that are connected by references. This fact lead to the idea to build a HyperBase.

B.1.2 What is a HyperBase

Because all these systems builds on the same principle it is not impossible to imagine that these systems was build on top of a common tool. The HyperBase is such a tool. It should not be seen as a HyperText application but rather as a tool to build a HyperMedia. The HyperBase is build to support that many programs, eventually different, uses the same HyperBase at the same time. To make a program use a HyperBase the programmer only have to include a library in his program that gives an interface to the HyperBase. The HyperBase is supplied with both a C++

interface and a C interface. The HyperBase is also supplied with a program that makes it possible for the programmer to configure the HyperBase to fit his needs better. To do that he will need a C++ compiler, more about that in the next section.

B.1.3 The HyperBase operations

The interface to the HyperBase consist of class named Hb_interface which has 19 public operations. They can be separated in tree groups: Those used to build the structure of the HyperBase, those to fill in information and those to make it easy for different programs to use the same HyperBase. Before these operations will be explained, the two basic elements in the HyperBase have to be described.

Link nodes and Data nodes

In a HyperBase a data node corresponds to a chunk of information and a link node corresponds to references between data nodes. Both entity types (an entity is a link node or a data node) consist of some key fields. These are shown in figure B.1.

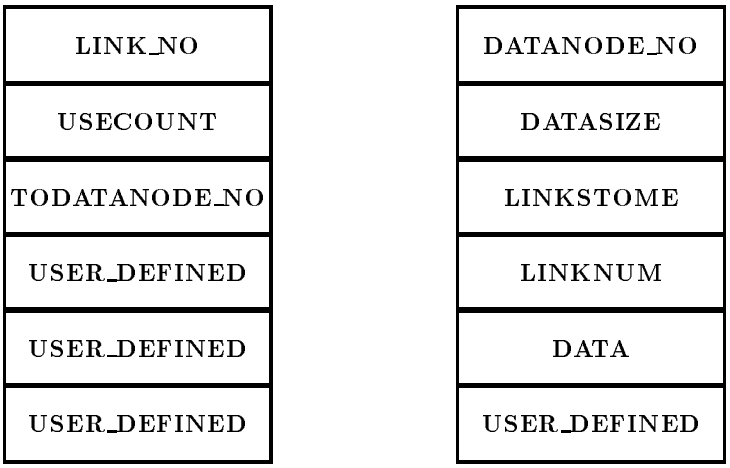


Figure B.1: *The different key fields in the entities. This way to see the entities is a user abstraction, but it includes all the user have to care about.*

The different keys in figure B.1 has the following meaning:

LINK_NO an unique entity number, the number is always odd.

USECOUNT describes how many data nodes that uses the link node as a reference to an other data node.

TODATANODE_NO describes what data node the link node points to.

DATANODE_NO an unique entity number, which is always even.

DATASIZE describes how many bytes data there is in the data field.

LINKSTOME describes how many links thats points to the data node.

LINKNUM shows the links used as references from the data node.

DATA contains the data in the data node.

Structure in the HyperBase

The operations, used to build the structure in the HyperBase are:

CreateNode(long* ent_no) Creates a new data node, and returns the data node number in **ent_no.**

Delete(long ent_no) deletes the entity specified by **ent_no**. It will not be allowed to delete an entity, if there still is references to it.

Link(long* from_ent_no, long to_ent_no) Makes a pointer between two entities (one data node and one link node). The operation have two different functions. If the entity, the pointer will point to, is a data node, there is created a new link node pointing to the data node (this is in fact the only way to create a link node). The link number will be returned in **from_ent_no**. On the other hand if the entity, the pointer will point to, is a link node, the pointer will just be inserted in the data node.

MoveLink(long link_no, long new_node) makes the link node specified by **link_no** point at a new data node **ent_no**.

RemoveLink(long data_no, long link_no) removes the reference to **link_no** in the list of references in the data node **data_no**.

Data in the HyperBase

All the mentioned key fields can be read by **Read** and **ERead**, but only data in data nodes an all user defined key fields can be changed with **Write** and **EWrite**. The reason is that all the read-only key fields describes states in the HyperBase, it is not variables that the user can change.

Read(int type,long ent_no,int key,char value)** Reads a key field, whichs value is returned in value (a char array). **type** describes whether the entity to be read from is a link node or a data node. Use the constants **LINKNODE** for link nodes and **DATANODE** for data nodes, these constants are defined in the header file from the HyperBase. **key** is the key to be read, and there are also defined constants for these.

Write(int type,long ent_no,int key,char* value) Writes down the value pointed to by **value**. Except value are the parameters in **Write** the same as in Read.

ERead(long ent_no, char value)** Reads an entire entity **ent_no** from the HyperBase. What happens is that the struct describing the entity is passed to the caller. The user can then cast **value** to the correct type, making it possible to work directly on all the key fields (except keys for links and data).

EWrite(long ent_no, char* value) Writes down the whole entity **ent_no**, which have been read by **ERead**. To make sure that the system keys will not be changed by accident, **EWrite** will only write the user defined keys down in the entity.

Multiuser operations

Because more users can operate on one base it is necessary to avoid that they change the same things simultaneous, which will lead to confusion. It is also a useful thing to know what other users do. In order to support these things the HyperBase supports events and locks.

The operations to support lock handling are:

Lock(long ent_no) locks the entity specified, meaning that other users can't operate on the entity.

UnLock(long ent_no) unlocks the entity. Please unlock an entity as fast as possible after you have locked it.

ShowLock(char UserId,long ent_no)** Shows the user that has locked the entity, the user's name is returned in UserId.

And the operations to support event handling are:

Event(long ent_no,int operation,int key) subscribe the user to events on the specified operation on the specified key in the specified entity (all these specifications can be all, e.g. 0, ALLOP and ALLDATA), this means that the user gets a message when something happens in the HyperBase, which fulfill the given template. The only operations it will not be possible to subscribe to are **Connect** and **DisConnect**. The messages is stored in a buffer until the user fetches them with the operation **GetEvent** or **ReadEvent**. It will be entirely up to the user to check the event buffer.

UnEvent(long ent_no,int operation,int key) unsubscribes from all events matching the given template.

ShowEvent(char UserId,long ent_no,int operation,int key)** Shows the users that have subscribed to the specified event, the result is returned as a list in UserId.

ReadEvent(long* ent_no,int* operation,int* key) Gets an event,which is returned in the 3 parameters if there is any. Returns at once with or without an event.

GetEvent(int time_ limit,long* ent_no,int* operation,int* key) is also used to get an event, and here the event is returned in the last 3 parameters. The first parameter describes how long time the **GetEvent** must wait, this factor is given in seconds. The difference between **ReadEvent** and **GetEvent** is that the first always returns at once while the second makes the program sleep if there isn't any events.

The operations to connect and disconnect to the HyperBase server are:

Connect(char* server_name),char* user_id Connects the user to the HyperBase that runs on the computer specified by server name, this implies that there only can be one HyperBase server on a computer.

Disconnect() Disconnects the user from the HyperBase.

B.1.4 Configuration of the HyperBase

To support the need for different key fields for different purposes a tool to shape the HyperBase has been written, this is named **hb_config**. **hb_config** is a tool which generates the basic definitions for data nodes and link nodes, and provides a unique naming of all operations and keys defined by the user. When **hb_config** has created these definitions, which is C++ files, it creates a directory with the source code for the HyperBase, the generated C++ files and a Makefile. The HyperBase can then be created by going into this directory and type **make**. The input to **hb_config** is a specification written in a simple language. It was chosen that the specification language should resemble normal written language, in contrast to a sentence written in C++/C syntax. Since there are some parameters which must be set the sentences is divided into 2 parts, a must-part and an optional part. To compile the most simple HyperBase, there must be provided a configuration like the following:

```
CONFIG
   set number of direct links in datanode to 5
   set number of direct links in linknode to 1
   set number of direct blockpointers in datanode to 1
   set size of datablocks to 511
```

The first line defines how many direct links, and the second line specifies how many direct data block pointers there are in a data node. The present implementation supports a unlimited number of direct links. The third line defines the size of the data block which is used to store the information. This is just a block size used to save the data in, how many bytes this gives totally are described at page 33 in the report.

Apart from these definitions which must be defined when you run **hb_config**, there is a number of optional specifications which can be supplied. The keyword **OPTIONAL** indicates that the rest is optional specifications. The following is an example on the possible optional specifications:

```
OPTIONAL
  include key myfancyentryforpinkelephants in data node
  set size of myfancyentryforpinkelephants to 8 in data node

  include key hubert in link node
  set size of hubert to 9 in link node

  set id to elephantbase
  set number of users to 42
  set socketid to 10045
```

The two first optionals shows how to include a key in a data node. The following two lines shows how to include a key in a link node. For each key included, the size of the key must be specified. The last lines shows how some default values can be changed. The default directory to compile the HyperBase in is called HYPERBASE, but now it will be called elephantbase. When the server is started it has a special communication channel on which it handles connection requests. This is default done on the socket with number 10023, but can be altered as shown here. Finally the maximal number of users of the HyperBase can be specified.

To open more than one HyperBase

The user can be connected with more HyperBases because it is possible to make more instances of Hb_interface, but if the user defined keys in one base isn't a superset of the user defined keys in the other base it will lead to troubles because it isn't possible to include two HyperBase header files at the same time. It is only the user defined keys that changes in the interface, when a new kind of HyperBase is defined. This means that the problem can be avoided if the programmer find the constants that represent the user defined keys in one of the header files, and define these in the second header file, without the other definitions, which is identical in the two files. Then it should be possible to use the same Hb_interface. The problems arises because of globally defined constants (all the keys and operations).

C-interface

C++ is not as common a language as C, because of this there is also a C-interface to the HyperBase. For every function in the public part of class hb_interface there is a function in the C-interface. The only difference is that all the C-functions has an extra int as first parameter. It specifies which base the user wants to operate on this time. The programmer decides the numbers himself, the number that he supplies a successful connect is the number that identifies the opened base. It will give an error to try to use the same number twice.

B.1.5 Error codes

The error codes can be divided into 3 parts. A single code **0** indicating that everything went all right, negative numbers indicating fatal errors and positive numbers which is warnings.

The error and warning codes are all 3 digited, where the first digit indicate the block the message came from, e.g. the message 202 is a warning from block 2 (the class **Operations**).

The meaning of the different codes are given in table B.1

B.2 Programmers guide

This second part of the manual is only intended to be used by programmers. The subject here is extensions of the HyperBase. The extensions can be divided into two different categories. Insertion of new fields in the entities and insertion of new operations.

B.2.1 Extra key fields

The HyperBase tool have been designed to make it easy to insert new key fields by use of the configuration file. Further more are the **Read** and **Write** operations specified in a way that makes them capable to work on the user defined keys too.

The configuration file

When you want a user defined key in the HyperBase entities, you will have to specify its name and length in the configuration program. To show precisely how it works there will be given a short example, for a key **My_Fancy** which is 6 bytes long:

Error code	Meaning
0	Nothing suspicious to report
208	Direct write not allowed
207	Link not found
206	Cant use data keys in link node
205	Cant use link keys in data node
204	Wrong arguments in call
203	Unable to open entity
202	Pointed entity not found
201	key not recognized
-201	Could not open/close entity
-202	Entity not found
-203	Unable to write links
-204	There is still pointers to entity
301	No event
302	No send event
350	Locked
351	Locked by other
352	Not locked
353	Not all nodes
-401	The HyberBase server is all ready connected
-402	There is no machine by that name, gethostbyname failed
-403	The socket call failed, could not make a file descriptor
-404	Could not connect to the host with that port; or no more room for clients at server
-405	Can't disconnect from server
-406	fcntl() can't change readfd
-407	The bind call failed
-408	The call getsockname failed
-411	There is no event in event buffer or an error
-502 -503	No such key/ wrong key
-504	No such operation
-505 -507	No such type/wrong type
-506	All ready connected
-508	Not disconnected

Table B.1: *The error codes given by the HyperBase tool.*

```
include key My_Fancy in datanode
set size of My_Fancy to 6 in datanode
```

There will not be given any more details about the configuration program here. If you wants to know more about this part of the HyperBase - please read section B.1.4.

B.2.2 Extra operations

Insertion of extra operations in the HyperBase is a more fancy extension. This kind of extensions can be made in two different ways. One possibility is to make the new operations on top of the already existing HyperBase tool, i.e. to make a new library module with the extra operations defined. This way to extend the operation vocabulary is simple but costs in network communication and main memory, see figure B.2. This is due to the fact that the blocks 4b to 6 will have to bee loaded into main memory by every user. Another way to extend the HyperBase is to go into the source code and code the extra operations. This is a harder way to do things, but it is more efficient both compared to network communication and main memory.

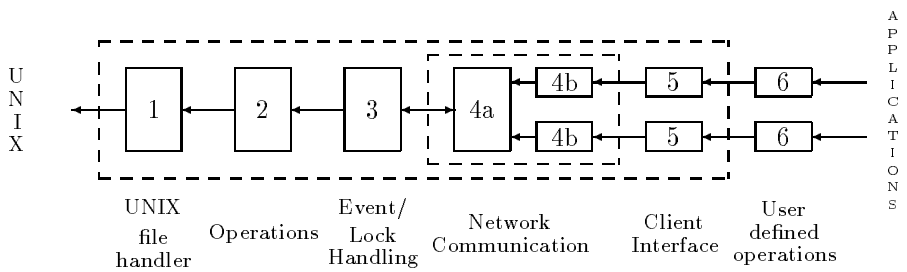


Figure B.2: *Shows how it is possible to extend the HyperBase operation set by introducing an extra class (block 6). All the blocks 1 to 5 is already implemented as classes.*

Superstructure - block 6

To make a new library module to implement new operations has the great advantage that it does not involve any changing of the source code but only an extension. This means that the programmer does not have to understand the already written source code to be able to make the change. Another great advantage obtained by introducing a superstructure is that it gives the possibility to redefine/overshadow the old operations, e.g. to insert an extra check before the user is allowed to write. The disadvantages (you get nothing for free) is that this library module also will have to be included by all the users. Often the new defined operation will use the already defined operations and this will result in a number of network communications, which will slow down the system.

Insertion

To insert the new operations in the HyperBase is a little more complicated than just to make a superstructure, but as already told it certainly also has it advantages.

With the used design of the HyperBase it is actually not very complicated to insert new operations. It just involves inserting some small code segments in different classes in the code.

All the classes where a modification is needed and the kind of modification will be described in the following. The functions/code segments needed to be inserted are typically 10-15 lines long, and most of these code segments are made in a standardized form, so just steal!

Block 5: Insertion of the new function as a public procedure in **block5.hh**. And insertion of the new function in **block5.cc**, which checks the parameters to see if the, are allowed.

Block 4b: Insert a new public function in **block4b.hh**, and code the corresponding function in **block4b.cc** to specify how to send and receive the different parameters.

Block 4a: Insert a new entry in the switch statement in the function **hb_read_message** in the file **block4a.cc**. This entry tells how to read a specific operation from the network, and how to send the result back.

Block 3: Insert a new public function in the class definition in **MainEventLock.hh**. And insert the code for the function in **MainEventLock.cc** specifying how to set/reset Events.

Block 2: Insert the new function as a public function in **Operations.hh**. Then the operation can be coded in **Operations.cc** in the language given by C++ and the classes that implements block 1 and 2. To put it in another way – here you specifies the operations semantic.

Block 1: No changes needed.

hb_config: In the configuration program all that needs to be done is to insert the new operations name in the printf statement in the file **symbol.c**.

After these expansions of the original HyperBase all that needs to be done is to compile and link the server and client processes.

B.2.3 Modularity

The HyperBase is constructed in a number of distinct classes performing each their task. This makes it easier to make system changes. This is due to the fact, that if you e.g. wants to use another network abstraction it will only be needed to rewrite this one class. This also helps to make the program more portable, because it gets easier to localize the programs different functions, if they have to be changed on the new machine.

B.3 Lacks

The operations **ERead** and **EWrite** does not work yet. Maybe the problems will be solved after this report is finished, we will work on it. The reason that **ERead** and **EWrite** does not work is that we can not transfer byte streams containing zeros. This also means that we can not transfer binary files.