REDUCTION OF SOFTWARE DEVELOPMENT COSTS UNDER SOMBRERO, A SINGLE ADDRESS SPACE DISTRIBUTED OPERATING SYSTEM

by

Ron Feigen

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2001

REDUCTION OF SOFTWARE DEVELOPMENT COSTS UNDER SOMBRERO, A SINGLE ADDRESS SPACE DISTRIBUTED OPERATING SYSTEM

by

Ron Feigen

has been approved

December 2001

APPROVED:

_____, Chair

_____

_____

Supervisory Committee

ACCEPTED:

_____

Department Chair

_____

Dean, Graduate College

ABSTRACT

This thesis examines the potential for reduction in software complexity of applications developed under the Single Address Space Operating System Sombrero versus those developed under a conventional process-oriented operating system, Windows 2000.

To test Sombrero‘s impact on development costs, a set of sample database applications with varying functionality was developed under Windows 2000 and Sombrero. Using accepted software engineering metrics, the software complexity of these systems was compared.

The results obtained indicate a substantial reduction of software development costs and defects in a single address space environment such as that provided by Sombrero.

DEDICATION

To my family for all their support.

In memory of my Mother, to whom this day was so important.

ACKNOWLEDGEMENTS

Many thanks for the effort and support of my committee members Dr. James Collofello, and Dr. Susan Urban. Special thanks to my committee chair, Dr. Donald Miller.

In addition I would like to thank the members of the 64-Bit Operating System Group, particularly Alan Skousen, for their support and also Power Software Limited for the evaluation license of Krakatau Professional (C++) Metrics.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER

1 - INTRODUCTION 1

2 - BACKGROUND 4

2.1 Sombrero System Services 4

2.1.1 Single Virtual Address Space 4

2.1.2 Persistent Storage 5

2.1.3 Task Scheduling and Thread Migration 6

2.1.4 Active and Passive Servers 6

2.1.5 Distributed Applications 8

2.1.6 Data Replication 8

2.1.7 Data Migration 9

2.2 Sombrero Compared to Other SASOS 10

2.3 Software Complexity Metrics 13

2.3.1 Halstead 13

2.3.2 McCabe 16

2.3.3 Lines of Code Metrics 17

2.4 Project – Example Code Base to Support Metrics ... 20

2.4.1 Application Characteristics or “Why a DBS for the Project” 21

2.4.2 Description of the Four Stages of Functionality 23

2.4.3 Criteria for Selecting Process Oriented Operating System and Development Environment 28

CHAPTER Page

3 - METHODS USED TO EVALUATE PROGRAM COMPLEXITY 30
3.1 Development Phases 30
3.2 Comparison of Phased Development 30
3.2.1 Measuring McCabe’s Cyclomatic Complexity ... 31
3.2.2 Measuring Halstead’s Software Science 31
3.2.3 Measuring LOC 32
3.2.4 Measuring COCOMO 32
3.2.5 Assumptions 32
4 - IMPLEMENTATION OF EXAMPLE DATABASE APPLICATION 34
4.1 Design Overview of the Windows Application 35
4.1.1 Database Client 36
4.1.2 Windows 2000 Database Server 37
4.2 Services Used in the Windows 2000 Application 39
4.2.1 Windows 2000 WIN32 API Services 40
4.2.2 Services Developed on Top of Windows 2000. . 43
4.3 Windows 2000 Design and Implementation Details ... 48
4.3.1 Memory Mapping and Memory Map Management ... 48
4.3.2 Message Passing 50
4.3.3 Client/Server Communications 51
4.3.4 Load Balancing 51
4.3.5 Data Replication 52
4.3.6 Threads and Synchronization 52
4.3.7 Impacts of Using Distributed Shared Memory . 54
4.4 High Level Design of the Sombrero Application 55
4.5 Services Used in the Sombrero Application 57
4.5.1 Native Sombrero Services 58
4.5.2 Sombrero Services Compared to Windows 62

CHAPTER Page

4.6 Sombrero Design and Implementation Details 66
4.6.1 Virtual Memory Allocation 68
4.6.2 Threads and Synchronization 71
5 - RESULTS 72
5.1 Results of Sombrero and Windows 2000 Metrics 72
5.1.1 LOC Metric 75
5.1.2 Halstead’s Metrics 77
5.1.3 COCOMO 79
5.1.4 McCabe’s Cyclomatic Complexity 79
6 - CONCLUSION AND VALUE OF RESEARCH 82
6.1 Impacts of Sombrero on Software Life Cycle 82
6.1.1 Other Factors 84
6.1.2 Trends in Modern Computing 85
6.1.3 Conclusion 86
REFERENCES 87
ADDITIONAL READING 93
APPENDIX
A DATABASE CLASS USAGE 96
B WINDOWS DATABASE CLIENT DESIGN NOTES 102
C WINDOWS DATABASE SERVER DESIGN NOTES 107
D SOMBRERO DATABASE DESIGN NOTES 112
E DEVELOPMENT AND TEST ENVIRONMENT 115
F WINDOWS 2000 TEST AND MEASUREMENT CONFIGURATIONS 117
G SEPARATELY BOUND APPENDICES 124

LIST OF TABLES

Table		Page
1	Effort For Three Modes of Basic COCOMO	20
2	Metrics Captured from Sample Database	74
3	Factors Used for Basic COCOMO	79

LIST OF FIGURES

Figure Page

1. Windows Functionality Stage 1 24
2. Windows 2000 Functionality Stage 2 25
3. Stage 3 - Windows 2000 Development 26
4. Windows 2000 Load Balancing 27
5. Windows 2000 Replication 28
6. Simple Client/Server Relationship 36
7. Clients Accessing Replicated Databases 37
8. Virtual Memory Moving Window for Large Files 44
9. Data Replication 46
10. Sombrero Database Class Acting on Persistent Data in Virtual Memory..... 57
11. Hardware Used In Testbed 116
12. Metrics Generation Process 118
13. Windows 2000 Phase 1 Database 120
14. Client / Server Relationship 121
15. 1 DBS Server, 3 Datasets, 4 DBS Clients 121
16. Multithreaded Database Server 122
17. 2 DBS Servers, 2 Datasets, 4 DBS Clients 123

1 - Introduction

The premise of this thesis is that the architecture of Sombrero, a 64-bit single address space operating system [17],[18],[19],[21], reduces the complexity of application software and hence the cost of software development.

This thesis compares and discusses the development of a sample application under Sombrero, a single address space operating system and Windows 2000, an industrial grade process-oriented operating system. Comparisons are made using well-known software engineering metrics. These metrics quantify the impacts of developing software under Sombrero and Windows 2000 in terms of software complexity, quality, development effort, and costs.

A Single Address Space Operating System (SASOS) can be thought of as a system where all data and threads of execution, user and/or system, operate in the same virtual address space. A SASOS removes the address space boundaries that make data sharing between processes difficult and expensive, as in traditional process oriented multiple address space operating system (MASOS), like Windows 2000 or UNIX.

The Sombrero operating system being developed at ASU, extends the concept of a single virtual address space to

include all code and data residing on a node's backing store and also encompasses all levels of the storage hierarchy of peer nodes in a distributed system. These characteristics support distributed computing, a less complex programming environment and potentially higher performance [23] than what is available from conventional operating systems.

Sombrero's use of hardware-supported object-grained protection offers the potential of significant performance advantages.

This thesis does not explore the potential performance gains to be realized by Sombrero. We believe that, given the continually decreasing price/performance ratios for computer systems, it will require more than an increase in operating system performance to justify changing to a new operating system and new computing paradigm in order to develop applications under Sombrero.

This thesis explores the impacts of Sombrero on decreasing the complexity of applications, thus decreasing the total cost of developing software. The positive impacts on the software lifecycle include the overall quality of

software applications in terms of potential defects and maintenance costs.

2 - Background

2.1 Sombrero System Services

The underlying Sombrero architecture provides several powerful abstractions that are embodied in transparent or near-transparent system services provided to a software systems engineer, architect, or developer. These services are explained in detail in [3]. A brief summary of Sombrero highlights follows.

2.1.1 Single Virtual Address Space

A Single Address Space Operating system (SASOS) uses a single address space for all data and threads of execution, user and/or system. For the removal of the address space boundary found in traditional MASOS to be practical, a SASOS is generally expected to be deployed on 64-bit or greater processors.

The removal of the process boundary allows the same virtual address to be bound to data and code across multiple protection domains. This allows inter-domain communication typically limited to expensive and/or clumsy message passing or shared memory mechanisms, to have the additional resource of a native and transparent distributed

shared memory abstraction that is integrated into the underlying operating system.

In Sombrero, virtual addresses are permanently and uniquely bound to all code and data objects. Thus, a virtual address can serve as a unique identifier and a virtual address space can serve as the only namespace. This binding persists across multiple replications of an application and/or data set and is consistent across multiple nodes in a distributed system. It is also consistent across all levels of the memory hierarchy. Thus, unlike MASOS distributed shared memory implementations, applications can always depend on a virtual address having the same meaning regardless of where the application is run on the network.

2.1.2 Persistent Storage

In Sombrero, the virtual address space is extended to backing stores, which are viewed as just additional levels in the cache hierarchy. Data becomes persistent without the use of a separate file system. Thus, the traditional concept of a file system, with a separate namespace, becomes optional.

In a distributed system, the same concept of persistent storage holds true as the virtual address namespace spans all levels of the storage hierarchy on every node.

Since virtual memory has native persistence, there is no need to develop pre-processor compiler directives to simulate persistence in a Sombrero system [4].

2.1.3 Task Scheduling and Thread Migration

In Sombrero, thread migration is straightforward. Since objects exist at the same virtual addresses on all nodes, threads may travel throughout the Sombrero network without an address space switch or a change in thread context.

Since thread migration is transparent to the application, the scheduler has greater flexibility in the way it integrates other nodes in its hardware and software resource pool.

2.1.4 Active and Passive Servers

Servers can be active or passive. An active server is analogous to that found in a normal process oriented

operating system (MASOS) where each service must be animated by a local thread.

A passive server is an instantiation of a program class that creates an object that provides a service, but with no local thread of execution. The object is created but not animated by an associated process or thread of execution.

A client accesses a passive server by calling the server object's methods, and in the process supplies the thread of execution. A client accesses a passive server's methods by instantiating a special C++ class object in the client program, designed to represent the server, which then acts as a proxy. Calling the proxy's methods automatically exports the calls to the server by transparently resolving symbolic references added to the client by the compiler into pointers to the server methods that are then stored in the proxy. This allows a server call to appear to the client program as a local class reference. This implementation enables the client to access the server via an ordinary subroutine call.

The server may or may not reside on the same node as the client. Any required migration of service or thread

across nodes is transparent to the developer and the application.

2.1.5 Distributed Applications

By default, all applications are distributed in Sombrero. The Sombrero scheduler will, unless instructed to do otherwise, consider all resources in the network when determining where a thread of execution should run. The concept is analogous to how a process-oriented scheduler manages symmetric multiprocessing (SMP) on a multiprocessor.

2.1.6 Data Replication

As discussed in the previous section, distributing threads of execution in Sombrero is analogous to SMP. The thread of execution will tend to migrate to the node closest to the required resources.

If the required resource is data in virtual memory, the Sombrero scheduler may determine that it is cheaper to migrate (replicate) the data to the node running the thread than it is to migrate the thread to the node with the data.

In addition, Sombrero memory allocation primitives allow an application to specify to which node(s) data is allocated. The server can request that the data be allocated on a specific node or replicated across a set of nodes on the Sombrero network.

2.1.7 Data Migration

Recall that objects exist at the same virtual addresses on all nodes thus, Sombrero can copy data from one node to another as needed, similar to the way cache memory can be used in a symmetric multi-processor system.

Additionally, the Sombrero scheduling and load balancing algorithms may, instead of copying data to a local node, migrate the thread of execution to a remote node where the data resides.

In either case, the migration is done transparently to the application.

2.2 Sombrero Compared to Other SASOS

SASOS can be classified into two categories:

- Hardware supported SASOS (Sombrero and AS/400)
- Software supported SASOS (Opal and Mungi)

Hardware supported SASOS can take fuller advantage of single address space characteristics. In the case of the developmental version being constructed at ASU, Sombrero uses the DEC Alpha's re-programmable PAL code, which allows flexibility in loading the translation look-aside buffer (TLB), to simulate an interim object-grained hardware-based virtual memory protection mechanism. In Sombrero, address translation is no longer needed in a TLB onboard the CPU as all threads of execution share the same virtual address space and therefore all onboard cache can be virtually indexed and tagged, thus CPU protection hardware is used to provide object-level protection services for the operating system, and address translation can be moved to a device on the memory bus.

In general it can be said that a hardware-supported SASOS is potentially much faster than a software-supported

system and, more to the point of this thesis, the software-supported architecture retains much of the architectural legacy of traditional MASOS. A summary of the architecture of hardware and software supported SASOS follow:

- Software SASOS
  - Obtaining system services requires a domain crossing and entry into privileged supervisor mode
  - Layered system structure: operating system kernel, executive services and user-level servers layered on top of each other
  - Direct support for Object Oriented Data (OOD) and Object Oriented Programming (OOP) is difficult: preprocessors and run-time systems are needed
  - Copy set replication and consistency management is usually page based and requires broadcasts reducing scalability
  - Network-wide fault tolerance can require additional mechanisms
  - Page-based object granularity

    - Access authorization, resource accounting and name services typically require kernel services or intervention
    - OS support for protection policy and protection implementation are intertwined
- Hardware SASOS:
    - Obtaining operating system services are orthogonal to entry into privileged supervisor mode
    - System architecture can be flat and modular: OS services, environment servers and user-level programs accessible to each other via ordinary procedure calls.
    - OOD and OOP can be directly supported: An object class can be implemented directly as a protection domain and a server as an instantiation of the class; base classes can be extended via user overrides
    - The common VA space enables simple scaleable per-object copy set replication and consistency management naturally provides a lowest level cache for fault tolerance

- Object granularity can be independent of page granularity.
- Access authorization, resource accounting and name services can be done in user programs.
- Clean separation of OS support for protection implementation and user definition of protection policy is simple.

2.3 Software Complexity Metrics

Software engineering concerns regarding software quality and complexity are magnified by the continual increase in application size and complexity [8]. To achieve the goal of improved software methodologies that can reduce costs incurred throughout the software lifecycle, software engineers and metricians have, over several decades, developed metrics that quantify the complexity of a software design and predict development costs and project quality (in terms of defects).

2.3.1 Halstead

Halstead's Software Science measures a program's vocabulary, volume, length, difficulty, and language level.

From these measurements it estimates a programmer's productivity.

Halstead's Software Science metrics estimate programming effort by measuring the following properties [7]:

- η1 = number of unique or distinct operators appearing in that implementation
- η2 = number of unique or distinct operands appearing in that implementation
- N1 = total usage of all of the operators appearing in that implementation
- N2 = total usage of all of the operands appearing in that implementation

From these metrics Halstead also defines[1]:

- Program Vocabulary: as η as η1 + η2
- Program Volume: V as N log η
- Potential operator count: η*1
- Potential operand count: η*2
- Potential Vocabulary: η* as η*1 + η*2

- Potential Volume: V* as η* log η
- Implementation: N as N = N1 + N2
- Program Effort: E as V2/V*
- Bug Prediction: E**(2/3) / 3000

The following text expands on a few of the more interesting Halstead metrics [5].

- Programming Effort: The programming effort is restricted to the mental activity required to convert an existing algorithm to an actual implementation in a programming language.
- Program Volume: This metric is for the size of any implementation of any algorithm.
- Coding Time: Using the work of psychologist John Stroud [5], Halstead estimates that Coding Time (T) = Program Effort (E) / Stroud number (S). Per Stroud 5 <= S <= 25. In Halstead Software Science, typically S = 18.

---

[1] Halstead defines additional formulas not described in this text

2.3.2 McCabe

McCabe’s Cyclomatic Complexity is a graph-theoretic complexity measurement that is independent of physical size (varying amounts of functional statements leaves complexity unchanged). To McCabe, complexity is a measure of the decision structure of a program and McCabe’s Cyclomatic Complexity is a measurement of the number of paths through a program. Cyclomatic Complexity can be used to measure, locate, and control complexity in software applications.

Cyclomatic Complexity is derived from a flow graph, mathematically computed using graph theory, to determine the number of decision statements in a program.

McCabe’s Cyclomatic Complexity v(G) measures program complexity by counting the program’s decision statements, called predicates. However, a decision statement can contain compound conditions, for example:

```
IF A==B AND C==D THEN
```

If only the predicates are counted, v(G) is equal to 2 (1 IF statement + 1). This does not account for the

compound conditions. The example above could be also be represented as:

IF A==B and IF C==D THEN

In this case, v(G) would be 3. Cyclomatic complexity recognizes that compound conditions increase a program's complexity and so it also counts individual conditions.

2.3.3 Lines of Code Metrics

There are many accepted metrics that are based on a project's source lines of code, as size is a primary cost factor in many models. The lines of code metrics used in this project are discussed in the following sections.

2.3.3.1 Lines of code (LOC).

Lines of code is a shear volume metric. Much study [14] has been done on LOC metrics and variants based on LOC estimates such as SLIM and COCOMO.

There are several factors to a LOC estimate. The numbers of non-commented sourced lines of code (NCLOC) and the commented lines of code (CLOC).

Commented lines of code (CLOC) are also a valid measurement. Total length of a project can be represented as:

LOC = NCLOC + CLOC

2.3.3.2 Constructive cost model (COCOMO).

COCOMO is a public domain metric and is generally acceptable as a software development cost estimation tool, producing cost estimates within 20% of actual requirements, 70% of the time [1].

COCOMO is a relatively straightforward model based on inputs relating to the size of the system and a number of cost drivers that affect productivity.

The original COCOMO [2] is a collection of three models: a Basic model that is applied early in the project, an Intermediate model that is applied after requirements are specified, and an advanced model that is applied after design is complete. All three models take the form:

E = a x S**b x EAF

Where E is effort in person months, S is size measured in thousands of lines of code (KLOC), and EAF is an effort adjustment factor (equal to 1 in the Basic model) [2]. The factors a and b depend on one of 3 development modes:

- Organic mode – relatively simple projects in which small teams work to a set of informal requirements.
- Semi-detached mode – an intermediate project in which mixed teams must work to a set of rigid and less than rigid requirements.
- Embedded mode – a project that must operate within a tight set of constraints.

The Basic COCOMO model computes effort as a function of program size [14]. The Basic COCOMO equation is:

E = a x KLOC**b

The factors a and b for the Basic COCOMO model are shown in on the next page (Table 1).

Table 1

Effort For Three Modes of Basic COCOMO

Mode	a	b
Organic	2.4	1.05
Semi-detached	3.0	1.12
Embedded	3.6	1.20

2.4 Project – Example Code Base to Support Metrics

The purpose of the sample application developed to support this thesis is to provide examples of the programming effort and constructs that would be required when developing real-world applications of the class that benefits from the environment provided by the Sombrero architecture.

The premise of this thesis is that the programming environment resulting from the underlying Sombrero architecture has significant advantages over a traditional environment for developing applications that rely on: data persistence, message passing between processes, manipulating data (especially a large file like a

database), distribution and/or replication across a LAN for load balance, performance or fault tolerance.

2.4.1 Application Characteristics or "Why a DBS for the Project"

The sample application for this thesis should embody significant and complex system services and process interactions. These include: management of large amounts of data, data persistence, message passing between processes, distributed processing and load balancing.

Metrics could have been gathered using a collection of toy and synthetic benchmarks. Instead, a simple database application is selected as it embodies the requisite features mentioned above and can be identified as an application that parallels what is considered to be commercially viable. Database management systems are good examples because, typically, they are distributed applications that rely on many system services including: memory mapping of large files, message passing, data persistence [13] and multithreading. A commercial database also requires process distribution [15], replication and load balancing.

The application developed for this thesis is not a commercial grade application. The lines of code required to produce a commercial quality product could potentially be in the tens to hundreds of thousands of lines of code, or greater. Size (measured in source lines of code) would also be an issue in attempting to port an existing public domain application to Sombrero.

This application, consisting of several thousand lines of C++, can manage very large datasets (limited only by disk size, not available virtual memory) and provides the following primitive functionality:

- Open
  - Create new database. Database is configured by metadata provided to the database class constructor.
  - Open existing database
- Close
  - Preserve database state and metadata
- Search
  - Key
  - OID

- Iterate
- Insert
- Delete
- Replace

Details on design, implementation and usage of the database class are provided in Section 4 - and the appendices.

The sample database is developed in four phases, with each stage of development adding additional features, such as client-server architecture, or multi-threading, to the base set of database primitives outlined above. For Windows 2000, a fifth version of the database is developed that explores how adding distributed shared memory would effect development in a MASOS environment. The distributed shared memory version of the code is simulated, but meaningful metrics were captured.

2.4.2 Description of the Four Stages of Functionality

The four levels of functionality describe architectural characteristics of the database (i.e. data distribution or load balancing) that are implemented in

addition to the basic database primitives described in Section 2.4.1 and detailed in Appendix A.

The development effort under each operating system has a one-to-one mapping to the 4 stages of functionality that are described in the following sections.

2.4.2.1 Stage 1 - base application.

The base application is a database class, whose methods act on a local dataset and are called directly by a driver application, as depicted in Figure 1.

The database class and the driver application run in the same thread of execution. The class does not support multiple instantiations of the database class accessing the same dataset.

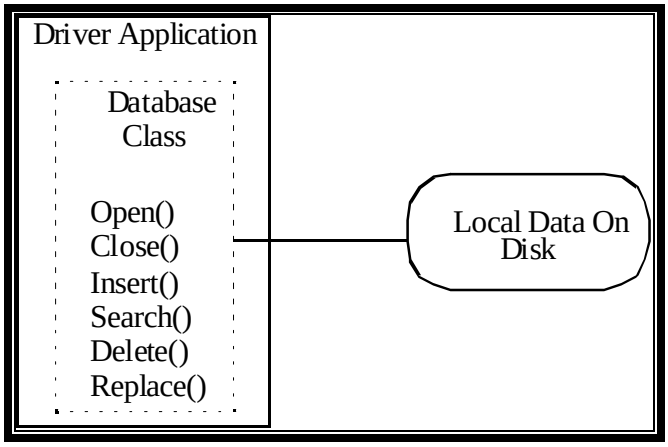


Figure 1 Windows Functionality Stage 1

2.4.2.2 Stage 2 - multiple clients, single threaded server

Stage 2 of the application requirements adds a multiple client-server paradigm to the Phase 1 development as shown in Figure 2.

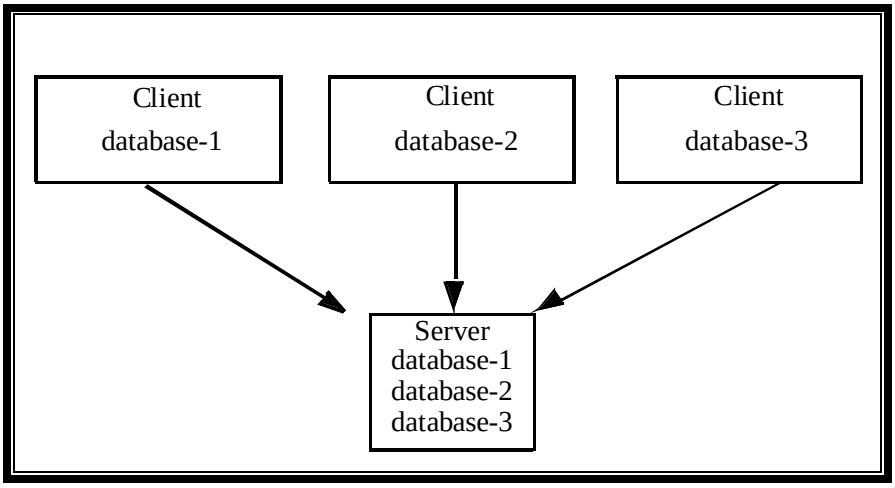


Figure 2 Windows 2000 Functionality Stage 2

The stage-2 database server class has the ability to handle requests from multiple clients directed to the same database or different databases.

This implies that multiple clients can access the same database, requiring that:

- State information is maintained per client.
- Open and close may be effected if multiple clients can open and close the same dataset.

2.4.2.3 Stage 3 – multithreaded server.

This iteration of development, as shown in Figure 3, adds a multithreading capability to the server class, requiring that the server manage critical sections.

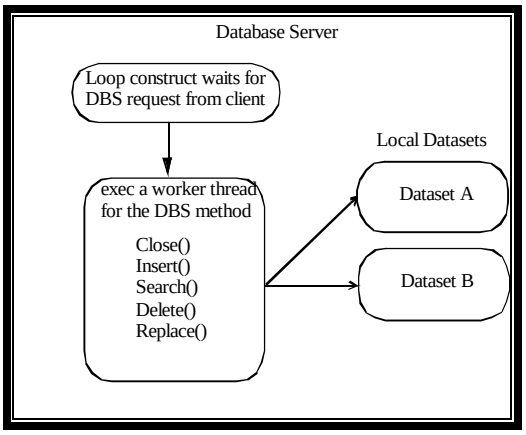


Figure 3 Stage 3 - Windows 2000 Development

The algorithm selected to manage critical sections permits either simultaneous readers or a single writer. A

thread wishing to update (write) the database must wait for all the readers to exit before proceeding. To accomplish this, no readers can enter a critical section lock if a writer is pending.

2.4.2.4 Stage 4 – distribution, replication and load balancing.

Stage 4, seen in Figure 4 and Figure 5 adds the following functionality:

- A load-balancing algorithm for read operations.
- Data distribution
- Operations that modify the dataset must ensure data consistency.

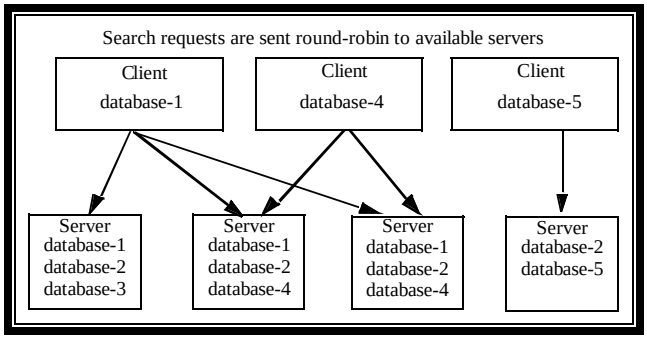


Figure 4 Windows 2000 Load Balancing

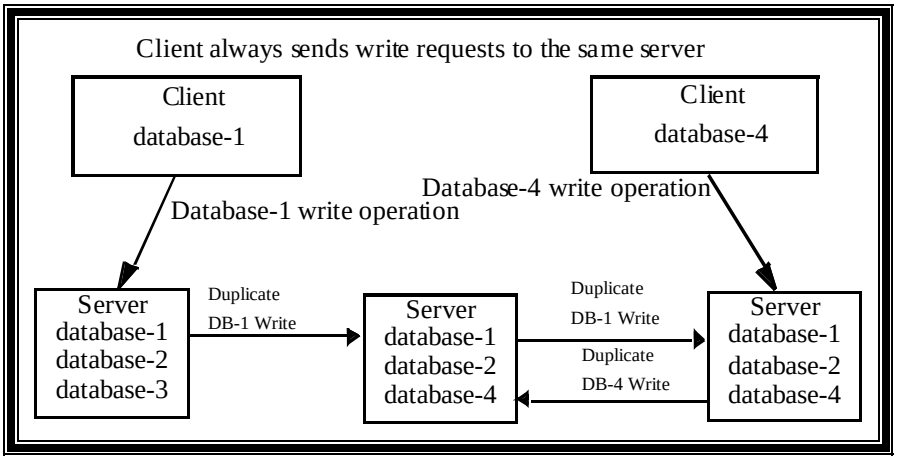


Figure 5 Windows 2000 Replication

2.4.3 Criteria for Selecting Process Oriented Operating System and Development Environment

The criteria for selecting the operating system and development environment for this project is that they be representative of a typical commercially accepted operating system and development environment.

2.4.3.1 MASOS operating system.

The traditional operating system chosen for the development system is Windows 2000. Windows 2000 is a common commercially viable operating system whose basic system services are analogous to those found in other available operating systems such as UNIX.

2.4.3.2 Windows 2000 and Sombrero development environment.

The application is developed using Microsoft’s Visual C++ Version 6.0, a commercially accepted and widely used development environment. Visual C++ is the common development environment for both Windows 2000 and Sombrero.

3 - Methods used to evaluate program complexity

3.1 Development Phases

As discussed in Section 2.4.2, there are four stages of required database functionality, with each stage adding additional functionality to the previous version. These stages of functionality have one-to-one mapping to development phases. In addition, a fifth Windows 2000 development effort simulates the effect of adding distributed shared memory to the MASOS development environment.

3.2 Comparison of Phased Development

Phased development provides the opportunity to compare complexity between the Windows 2000 and Sombrero code base and to compare phases of development to a previous phase (under the same operating system). This provides a view into how the added functionality requirements impact software complexity.

At each development phase, metrics are captured for the code base required to provide its specified functionality.

All of the raw metrics were captured using Power Software Krakatau Professional (C++) Metrics. The raw metrics data captured are summarized as described below.

3.2.1 Measuring McCabe's Cyclomatic Complexity

McCabe's raw metrics are captured using the evaluation version of the metric tool provided by Power Software. The raw data, which captures the McCabe count of decision statements per method, is too extensive for the delta comparisons required by this thesis for each phase of application development. Therefore a summary of the raw McCabe data for each phase is produced and put into table form using Microsoft Excel.

3.2.2 Measuring Halstead's Software Science

The evaluation version of the Power Soft metric tool produces Halstead's Software Science Program Effort and Bug Prediction metrics per source code file. Like McCabe's Cyclomatic Complexity, multiple Halstead metrics for a single phase of development is not conducive to the comparisons on a project basis.

Producing a single metric for each phase of development is accomplished by concatenating all the source files in the project for that phase into a single file. The concatenated file is then used to generate the Halstead metrics for the specific development phase.

3.2.3 Measuring LOC

Total Lines of Code measurements (LOC) are gathered on the concatenated project files used for Halstead.

3.2.4 Measuring COCOMO

COCOMO estimates were based on the LOC metric using the organic model. Only basic COCOMO, as described in Section 2.3.3.2, is calculated.

3.2.5 Assumptions

The following assumptions were made:

- The code base is representative of the types of coding constructs present in commercial applications.
- At this time, there is no running Sombrero operating system [19]. The Sombrero code base has to use a

systems services interface that contains a “best guess” of the semantics and syntax of a running Sombrero; thus the application is not testable.

- The Windows 2000 distributed shared memory version is compiled but not run, as distributed shared memory software was not available.

4 - Implementation of example database application

This chapter provides an overview of the Sombrero and Windows 2000 design and implementation that is required to produce an example database with stage 4 functionality as defined Section 2.4.

Only stage 4 functionality is explored as that version of the sample application contains the full feature set defined in Section 2.4. Discussion of phases 1, 2, and 3 would not provide additional information.

The Windows 2000 and Sombrero design overviews, contained in Sections 4.1 and 4.4, provide insight into the architectural considerations required when developing the same application under each operating system.

The Windows 2000 and Sombrero implementation overviews, contained in Sections 4.2 and 4.5, describe the services needed to support development of each version of the experimental application.

The Windows 2000 implementation overview Section 4.2.2, describes “added value” services. These are more complex services developed on top of the WIN32 Application Programming Interface (API).

The implementation overview Section 4.5.2, describes how Windows 2000 services that are described in Section 4.2.2 are provided in Sombrero.

Lastly Sections 4.3 and 4.6 provide examples of the implementation details for services described in other sections.

In addition, a detailed overview of the database class usage and additional design issues for each method is located in the appendices.

4.1 Design Overview of the Windows Application

This subsection describes the high level design used to develop Phase 4 of the project in Windows 2000.

When developed under Windows 2000, it is necessary to develop a separate client and server, as shown in Figure 6. The client is linked to a database class that provides access to a database server via message passing over TCP/IP.

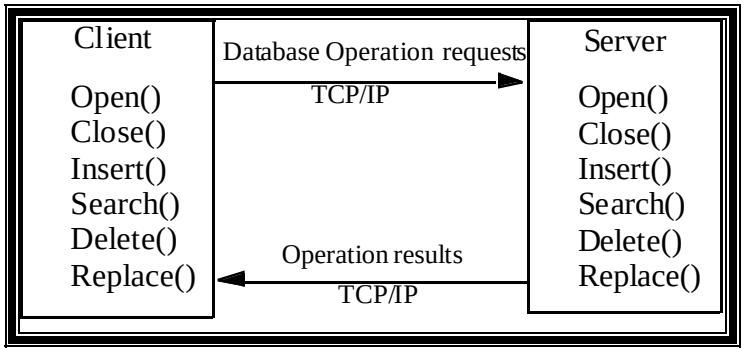


Figure 6 Simple Client/Server Relationship

The database server is a standalone, multithreaded process that listens for requests from a client, then creates a worker thread to perform the required operation.

4.1.1 Database Client

The Windows 2000 database client provides the following basic services:

- Access to database services via message passing (Figure 6).
- Public methods providing messaging services specific to each database primitive.
- Database read requests are load balanced by sending them round robin to a list of available servers (Figure 7).

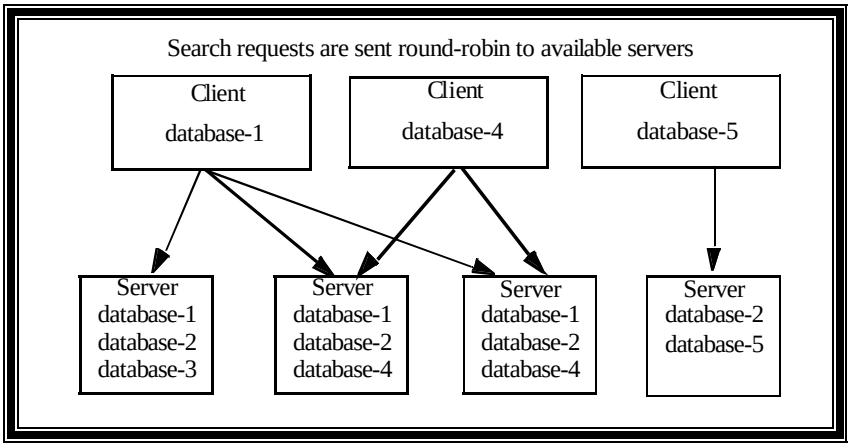


Figure 7 Clients Accessing Replicated Databases

To provide the services described above, the database client must also:

- Maintain client's private state information that includes:
  - Internal database ID. The ID associates an open database with a specific client and is passed to the server with each request.
  - List of servers. A list of all the servers that are hosting the open database.

4.1.2 Windows 2000 Database Server

The database server must implement sorting algorithms plus define and manage certain data constructs such as the

database file, index files, and metadata. These design and code considerations are the same in Sombrero and Windows 2000. This thesis does not focus on database design or implementation. A database design and implementation is included because it is needed to demonstrate application functionality that requires system services whose interface semantics are very different in a MASOS and a SASOS.

The items of interest to this thesis are the operating system services required to support the application. For example, to implement an efficient sorting algorithm the data must be accessible via pointers to virtual memory. Details of the sorting algorithm are not of interest; but how to map data into virtual memory is of great interest.

The Windows 2000 database server provides the following basic services:

- Access to database services via message passing.
- Map and manages databases in virtual memory.
- A database “window” into virtual memory for accessing parts of large datasets that exceed the available virtual address space.

- Constructs for simultaneous access by multiple clients to the same database.
- Worker threads for each database operation.
- Management of critical sections.
- Maintenance of private state information per each database client. This information includes:
  - Unique database ID. The ID is returned to the client when the database is opened.
    - The unique ID associates an open database with a specific client.
  - Most recent search information that includes:
    - Search type.
    - Pointer or OID of object returned.
- Database replication and distribution by forwarding database create, open, write or delete operations to "next-in-line" server[2].

4.2 Services Used in the Windows 2000 Application

The services required by the application include atomic Windows 2000 WIN32 application programming interface

---

[2] This service is not required if distributed shared memory is present.

(API) services and services built by the application on top of the discrete WIN32 services.

4.2.1 Windows 2000 WIN32 API services

This section describes the basic system services required by the sample application. These basic services may be used as building blocks for more complex services that are described in Section 4.2.2 “Services Developed on Top of Windows 2000.”

4.2.1.1 Memory mapped data.

Current databases do not access data by streaming through disk-based file data. Instead large portions of the files themselves are “mapped” into the process’s virtual address space. This allows an application to use memory pointers to quickly and efficiently access data (randomly). Virtual memory access to data based on disk requires that either the entire file, or at least large sections of the file, be explicitly mapped into virtual memory.

An issue for a MASOS is that for very large databases the size of the database and accompanying indexes may exceed limited virtual address space allocated to a

process. Thus, an application will have to develop a service that memory maps only a section of the large file. This provides a “view,” or window, into a subset of the data. This service must be able to move the view around, allowing access to the entire file as required. This “view” mechanism is described in greater detail in Section 4.2.2.1 Memory management for large files.

The mapping of files to virtual memory in a process-oriented operating system is a form of namespace translation. Each time data is mapped, it may be mapped to a different virtual address space; pointers have no meaning across process boundaries. Thus, how would an application create persistent data pointers in a MASOS?

A solution is for a persistent pointer to be stored as an offset to the object’s base address. At run-time, a pointer is formed by adding the offset to the base virtual memory address to which the object is mapped. This method is commonly called “pointer swizzling” [4]. Each time the object is mapped to virtual memory, the base address may change.

4.2.1.2 Native IPC.

Unless the application is constrained to a single process it is usually necessary for cooperating processes to pass messages. For example, a database client would pass a request to the database server for database services. The server would then reply with the results.

TCP/IP is one solution for inter-process communication, as it is reasonably fast, efficient and is location independent. It works the same for processes that reside on the same or differing platforms.

Using TCP/IP or other messaging protocols typically requires the application to deal with issues such as composing and sending messages, and reading and interpreting messages. This implies an agreed-to messaging protocol between the client and server.

4.2.1.3 Multiprocessing/threads.

For a database to be viable, the server must be able to handle multiple requests in parallel. Thus, the application will have to develop methods for creating and managing multiple threads of execution.

4.2.1.4 Synchronization.

The introduction of multiple threads of execution requires that the application have some method of managing critical sections to insure data integrity.

Typical minimal service required would include mutex and semaphores.

4.2.2 Services Developed on Top of Windows 2000.

The sample database application required additional services beyond those provided by the operating system WIN32 API. The following sections describe those services.

4.2.2.1 Memory management for large files.

Mapping a file into memory is a common method for managing large data sets. An issue arises if the amount of data being manipulated is larger than the available virtual address space.

For large datasets this can easily be the case with Intel 80x86 Windows 2000. When the dataset exceeds the available virtual memory space, a view or a subset of the dataset is mapped to virtual memory. Since the view is

smaller than the dataset there must be mechanisms to move the view if the entire dataset is to be accessible.

Figure 8 shows the address implications of multiple name spaces.

To resolve this issue, a Memory Mapping Page Handler is developed to ensure the proper pages from disk are mapped into virtual memory. Not unlike a classic memory management unit, the page handler examines the target address and maps the required pages in from disk.

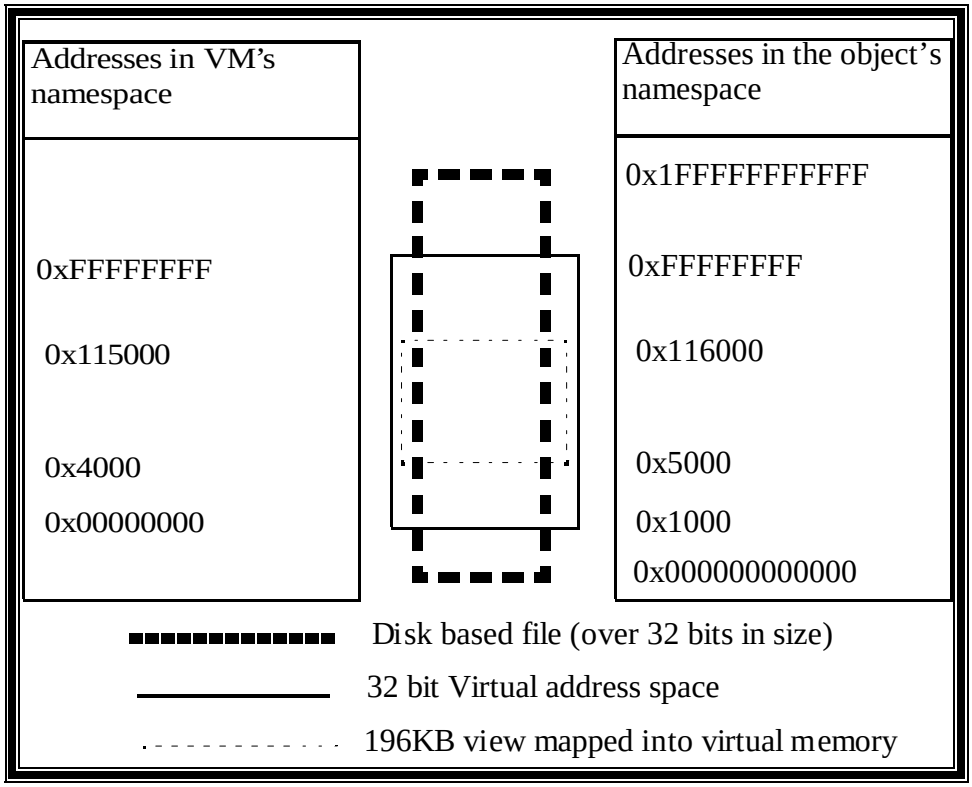


Figure 8 Virtual Memory Moving Window for Large Files

Unfortunately, this page handler is not transparent to the developer like a memory management unit. The application must call the routine prior to using any pointer into mapped memory.

The handler examines the requested address, makes sure the proper page of the data object is mapped, and returns an offset the application uses for swizzling pointers into that mapping of memory [4].

4.2.2.2 Data Replication / Distribution

The sample database has the capability to scale and replicate data across multiple servers. In the implementation in this thesis, this is accomplished by the client passing to a server the list of additional servers to which it wants the data replicated.

The server then replicates any open, close, or write operation requests to one of the peer servers. That server in turn, passes the operation to the next server, etc., as shown in Figure 9. Each server sends the message required for replication prior to executing the database operation, so the operations happen, more or less in parallel, less the latency of sending the messages.

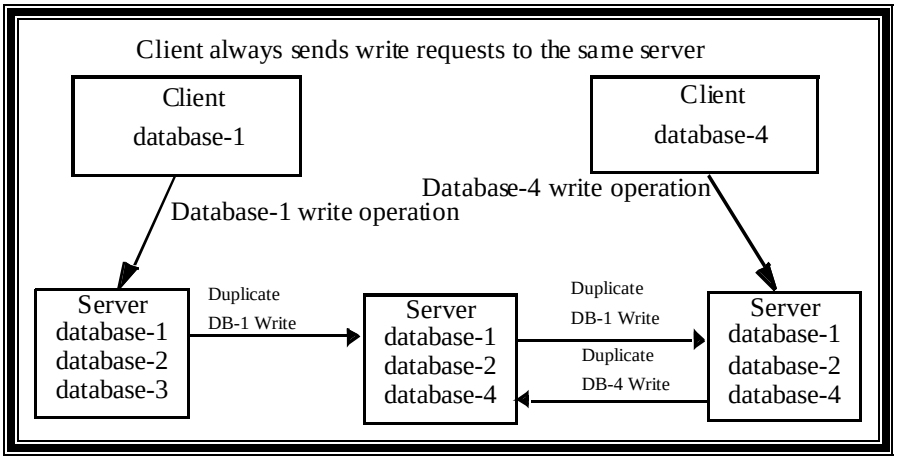


Figure 9 Data Replication

This mechanism requires that the client be aware of the network topology. Certainly in a commercial system a more elegant method would be used. This would invariably require a significant effort to develop.

4.2.2.3 Load Balancing

The sample application accomplishes load balancing by incorporating into the database client a simple round-robin algorithm for read operations.

Write operations are always sent to the database server that was used for the initial database open operation. The server replicates the request to its peers

as described in Section 4.2.2.2. All database servers in the sample application must receive the same write request to maintain data coherency.

Write operations can be load-balanced by clients using a different server for the initial open operation. In the sample application database, the gain would be slight as subsequent write operations are propagated to each server.

Adding distributed shared memory to Windows 2000, as discussed in Section 4.3.7, resolves many of the issues of distributing data. It also improves the load balance algorithm, as it enables write operation to be made to any database server. With DSM, that server does not need to replicate the write operation to its peer servers.

4.2.2.4 Message passing

To support communication between processes, the sample database must pass messages. This required the creation of a communication class to handle creating TCP/IP client and server ports, establishing inter-process communication, message read, message write and close connection.

In addition, the read and write processes require the calling and sending applications to agree on a message

format, including a header that may require an embedded length, for which one side of the connection is responsible for packaging and sending; while the peer must receive and interpret the message.

As TCP/IP is a streams protocol, the read method must make sure it gets all the bytes before it returns to the calling routine. This requires reading incoming data in a looping construct and building a message buffer.

4.3 Windows 2000 Design and Implementation Details

This section describes the design and development process of key features of the sample database developed under Windows 2000.

4.3.1 Memory Mapping and Memory Map Management

Manipulating datasets like an array of objects (database) or sorted arrays of key/indices pairs requires direct, random access to the data. Working with streams primitives (i.e. fopen(), fseek(), fread(), fwrite()) would be very inefficient. Thus, the sample database maps the user database and associated indices into the process's virtual address space.

The logic to accomplish this in Windows 2000 is illustrated below.

```
// Create a file of a specified size (you can't map
// pages that don't exist on disk).  Fileseek() will
// allocate file space even if there is no data
// written.

if (the database does not exist)
      FID = fileopen("database name");
      FileSeek (FID, requested size of database);
      File close (FID)

// Create a file mapping and map pages to VM
// MapFile parameters:

      // base address you would prefer,
      // the first page of the file you want mapped to VM
      // the number of pages to map to VM
HANDLE = create file mapping (database name)
BaseAddress = MapFile(HANDLE, address, page#, # of pages).
```

The virtual address space available to the application is often less than the virtual address space provided by the CPU (typically one-half to three-fourths) due to the design of the MASOS. To accommodate very large data sets, Windows 2000 must provide a routine to page into virtual memory, not unlike a typical memory management unit.

```
// Calculate which page of the file the pointer points
Page = Offset / PageSize
If (that page is already mapped)   //Do we have that view?
then
      return
else
      map the required page into memory
      save the current page for later use
      return (the view offset used for swizzling)
```

4.3.2 Message Passing

There is no IPC style message passing in the base application, as the entire process operates in the same data space. The database class is linked to the driver application; thus, message passing is limited to passing parameters to a subroutine (see Figure 1 Windows Functionality Stage 1).

Subsequent phases of development do require message passing from the database client to database server and, in the case of phase 4, between database servers.

In the sample application, message passing uses TCP/IP primitives. Since TCP is a streams protocol, there is no guarantee that, if the client writes 50 bytes, that all 50 will be returned to the read in a single invocation of read(). Thus, the application must implement read() in a looping construct to ensure all bytes are received. The code to perform this function follows:

```
// read until all expected bytes are processed
// allocate a buffer for the incoming data
char *buffer = new char[buffer size]

total bytes read = 0;
while (total bytes read <= expected bytes)
{
      bytes read = read(&buffer[total bytes read])
      if bytes read < 0
            return -1

      total bytes read += bytes read;
}
```

4.3.3 Client/Server Communications

The client/server relationship in the sample application is typical of what is found in many applications. The server opens a well-known port and waits for a connection from a client

The client opens an ephemeral port, to be used for connection and communications, and initiates a connection to the well-known port of its server. The server accepts connection requests from clients and creates a new port for the data transfer.

This implies that the client must either have some pre-existing knowledge of available resources or a method of learning what resources are available.

4.3.4 Load Balancing

Load balancing in the Windows 2000 sample application is implemented in the client. It distributes its read requests, round-robin, to a predefined pool of available servers.

4.3.5 Data Replication

Data replication is implemented in the DBS server. Each server forwards open, close, and write requests received from a database client to a single peer server, which, in turn, forwards the request to its peer, etc.; thus operations that modify the state of the database are executed on all copies of the database. When the database is created, the client passed the list of peer servers to the database constructor.

The impact of adding distributed shared memory to Windows 2000 is discussed in Section 4.3.7., DSM can simplify data distribution and allow a better load-balancing algorithm.

4.3.6 Threads and Synchronization

The Windows 2000 database implements a multithreaded server (first introduced in phase 3). A master thread waits for messages for a database client request, and then creates a worker thread to perform the requested task.

Since it is acceptable for multiple clients to access the same database, the worker threads need to

implement a locking construct to protect database integrity, yet still allow parallelism.

.A multiple reader, single writer algorithm [16] has been chosen to accomplish this task. The logic is as follows:

```
// Reader locking algorithm
// Allow multiple reads, exclusive writer
// Get the read lock before accessing read counter.
// Prevent writers by getting the write lock.
// Only the first reader needs to get/release the write lock

wait(read lock)
readers += 1;
if (readers == 1)
      wait (write lock)
release(read lock)

   . . .
perform read operation
   . . .

wait(read lock)
readers -= 1;
if (readers == 0)
      release (write lock)
release (read lock)



// writer
// can’t write if there are any readers or writers
wait (write lock)

   . . .
perform write operation
   . . .

release (write lock)
```

4.3.7 Impacts of Using Distributed Shared Memory

The effect of distributed shared memory supplied by application level middleware commonly used in process-oriented operating systems on the sample database is studied in order to compare the effects on software complexity with those from distributed shared memory obtained from the common virtual address space in a SASOS.

A careful analysis will show that the only benefit of distributed shared memory in a MASOS is to the phase-4 database server. Distributed shared memory removes the need for the server to replicate write operations to peer servers as described in Section 4.3.5. With DSM, updates by any server to the database are visible to all servers. This enables the client to use the read operation round robin load-balancing algorithm, described in 4.3.4, for write operations. Again, this is true because with DSM, modifications made by any server to distributed shared memory are available to all peer servers.

DSM provides the stage 4 server the ability to maintain data coherency with no impact to the DBS application developer. This positive impact is evident in the metrics shown in Section 5.1. However, it should be

noted that although we did not capture it in this thesis, a MASOS middleware DSM implementation is quite complex. This is not true in a SASOS – so increased software development costs really occur but in a different level of user application programming in a MASOS.

Phases of development 1 though 3 realize no benefit from distributed shared memory. Distributed shared memory does not solve any other issue, such as multiple virtual address spaces, data persistence or message passing between client and server.

Initially, one might believe, incorrectly, that client-server communication would benefit from distributed shared memory. A client would still be required to compose a message, in a shared memory location, and signal the server of the incoming message. This effort is analogous to passing messages via TCP/IP.

4.4 High Level Design of the Sombrero Application

The Sombrero version of the application provides the same functionality in each phase as the Windows 2000 application.

Due to the native abstractions inherent in Sombrero, the design and development of the Sombrero application with stage 4 functionality, is a simplification of the Windows 2000 application with stage 1 functionality; with the singular exception of having to add mutex to the Sombrero application to protect critical sections. The Sombrero passive server appears as a database class acting on a local dataset (Figure 10). The Sombrero server expends little or no effort dealing with architectural issues such as: persistence, distribution or load balancing. Major design changes in Sombrero include:

- Passive server instead of a traditional client and server. This implies that there is no higher level[3] message passing or IPC.
- Native persistence vs. memory mapping/memory management. In Sombrero, there is no need for the application to:
  - Map or manage virtual memory.
  - Swizzle pointers.

---

[3] Inter-machine messaging at lower layers, via TCP/IP is, still available in Sombrero.

- Load balancing. Sombrero threads of execution can transparently migrate to different nodes to access a resource, or the resource might migrate to the thread.
- Data replication. A Sombrero application can request that the operating system replicate memory objects. By default they are replicated as a side effect of the load-balancing mechanism.

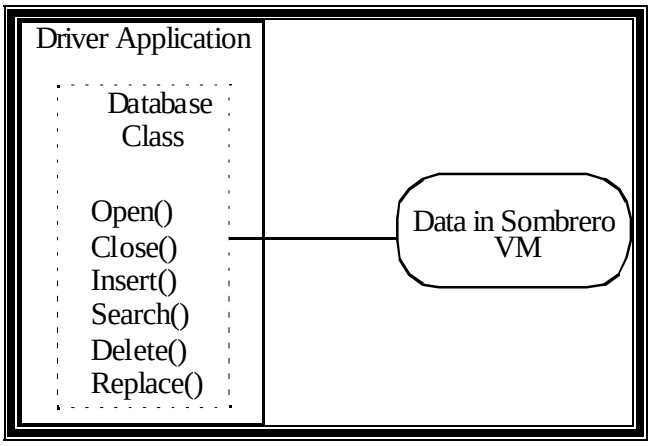


Figure 10 Sombrero Database Class Acting on Persistent Data in Virtual Memory

4.5 Services Used in the Sombrero Application

This section explores new services provided by Sombrero and how Sombrero, by its very architecture, provides many of the Windows 2000 services required for the

database application with little or no impact on the application developer.

4.5.1 Native Sombrero Services

As will be illustrated in later sections, a Sombrero application does not typically need to deal with issues such as message passing, data migration, replication, and distribution. In many cases even multithreading is abstracted to the point where the application must protect critical sections but not deal with the semantics of creating and managing worker threads.

NOTE: The native Sombrero services are currently being developed. The basic set of system services are described in [22], but the syntax and semantics for each command is not finalized. For the purpose of this thesis it is sufficient to approximate the implementation, as this will not significantly effect the metrics.

4.5.1.1 Memory allocation primitives.

Creating persistent storage in Sombrero is similar to allocating virtual memory from the heap in a MASOS. Because the memory allocated is persistent, the allocation

operation must allow for additional semantics that parallel fopen().

These semantics include:

- Name: A text-based alias for identifying the base address of the persistent virtual memory. This is analogous to a filename in a process-oriented system.
- Sharing: Protection and sharing attributes
  - SHARE
  - NOSHARE
  - SHAREWITHGROUP GROUPNAMES
- Mode: Open/Create semantics
  - Attach to previously allocated virtual memory
  - Create new virtual memory
- Repl: Replication semantics
  - No replication
  - Default replication
  - Replicate data on all nodes
  - Replicate on ‘n’ nodes
    - Sombrero picks the nodes
  - Replicate on a specific set of nodes
    - Application provides the list of nodes

- Migr: Migration semantics
  - No migration
  - Migrate data on all nodes
  - Migrate on ‘n’ nodes
    - Sombrero picks the nodes
  - Migrate on a specific set of nodes
    - Application provides the list of nodes.

The syntax of the system service is:

char *allocate(name, size, sharing, mode, repl, migr)

4.5.1.2 Root memory object control block.

The root memory object control block provides an application with a directory of all resources available on the Sombrero network. In addition, the application can access root memory control methods to direct the Sombrero scheduler to run a thread of execution on a specified subset (working set) of the available resources. This is similar to modifying the scheduling policies of a symmetric multiprocessing system.

A calling application can modify the scheduler to force a thread to execute on the local node by passing the #define THIS for the list of servers:

err = rmcb(schedule, <thread id>, THIS);

The list of servers specifies to the scheduler to which server it can migrate the thread. The default is that the scheduler can migrate the thread to any server in the Sombrero network.

4.5.1.3 new().

This familiar C function is also used to create new objects in Sombrero. In Sombrero, a variant of “new” can be called from a command line. For example:

“new (database, student)”

creates an instantiation of an object named “student” from the class “database.” Note: this class is not animated, as it has no thread of execution. This is how Sombrero creates passive servers.

4.5.2 Sombrero Services Compared to Windows

When developed on Sombrero, much of the functionality required by the sample application required little or no effort on the part of the developer. The following describes how the Windows 2000 services described in Section 4.2 are provided in Sombrero.

4.5.2.1 Memory mapped data.

The memory hierarchy of the Sombrero architecture includes the backing store of the local node and that of peer nodes in the local network. Thus, in Sombrero, all data on disk is mapped to the 64-bit address space that can support objects on the order of 18 quintillion bytes.

In Sombrero, data is persistent for the life of the object. Recall that in Sombrero, a passive server is a class instantiation without a process or thread of execution. Thus, all data associated with the passive server will persist until the object is destroyed.

4.5.2.2 Native IPC.

All processes in Sombrero share the same virtual address space. There is no need in Sombrero to pass messages. Either one application signals the other that data is ready, or, in the case of a passive server, the client accesses the server's methods, not unlike a local subroutine call.

4.5.2.3 Multiprocessing and threads.

Sombrero maintains the standard concept of threads. However, in Sombrero, a thread can migrate transparently to a node anywhere on the Sombrero network. There are powerful implications of this feature, such as application transparent load balancing and, data and execution distribution.

Another example involves multiprocessing. Recall the concept of passive servers from Section 2.1.4; now assume there are multiple instances of a client program running on the LAN. When a server method is called, the client thread animates the passive server. If there are multiple clients, the passive server becomes multithreaded by no action of its own. This is why a passive server, although

written as a single-threaded application, must implement code to protect critical sections.

4.5.2.4 Synchronization.

As in a MASOS, multiple threads of execution require that the application have some method to manage critical sections to ensure integrity.

Typical services required include mutex and semaphores.

4.5.2.5 Memory management for large files.

As described in Section 4.5.2.1, all data on backing store is mapped into Sombrero's 64-bit virtual address space. Thus, no additional page routines are required. Neither is pointer swizzling as the virtual addresses are bound to objects across all threads of execution, processes, and nodes in the Sombrero LAN.

4.5.2.6 Data Replication and Distribution.

Sombrero can provide transparent process distribution and data replication or allow the application to exercise control over the algorithm.

Sombrero will replicate data to the node that requires the information or migrate the thread of execution to the node with the data. Scheduling and data consistency are transparent to the application and developer.

If required, an application can force a thread to execute on a specific node; data required by the thread will also migrate. If the application wishes the entire database to be replicated to a node, it can request a thread to execute on a node that touches all the pages of data thus, forcing the transfer.

4.5.2.7 Load balancing.

As discussed in the prior section Sombrero load balances across a LAN in the same transparent manner that the Windows 2000 would load balance across symmetric multi-processors. If the application requires, it can force its own scheduling algorithm as discussed in Section 4.5.1.2.

4.5.2.8 Message passing.

As discussed in Section 4.5.2.2 there is no need for traditional messaging between processes in Sombrero. Sharing data between processes in Sombrero is analogous to

sharing data between threads (of the same process) in a MASOS.

4.6 Sombrero Design and Implementation Details

Using the services described in the Section 4.5, developing the database application is far simpler in Sombrero than it is in the Windows 2000 environment.

The design criteria for the Sombrero database class included the following:

- Passive server: Instead of a traditional client and server. This implies that there is no traditional message passing or IPC.
- Multithreading: This requires protection for critical sections, same as Windows 2000. However, since the thread execution is supplied by the client, neither the client nor the server required multithreading primitives. This is explained in detail in Section 4.6.2.
- Native persistence: No memory mapping or memory management/pointer swizzling required. There is no separate file system namespace to be translated to and from.

- Load balancing: Does not require development effort, as Sombrero threads of execution can transparently migrate to different nodes.
- Data replication: Does not require development effort, as a single Sombrero service request can request that the operating system replicate memory objects.

Ironically with Sombrero, some of the design issues per phase revolved around how to restrict, the native abstractions and functionality, provided to the application by Sombrero. For example, Sombrero supports sharing of its passive servers; but phase 1 functionality (see Section 2.4.2.1) requires that the database not be shared. Thus, the design had to consider how to restrict the capability of Sombrero to meet the phase 1 requirement.

Hence, a simple application can actually take more development effort than a complex application in Sombrero.

The design implications for Sombrero, per phase of functionality (defined in Section 2.4.2), are as follows:

- Phase 1: Single-threaded, single-client database
  - Restrict the database class to serve only one client at a time.
- Phase 2: Single-threaded, Multi-client database
  - Allow multiple database clients, but serialize database access.
- Phase 3: Multi-threaded, Multi-client database
  - Protect critical section.
  - Request that Sombrero does not replicate or migrate the database.
- Phase 4: Multi-threaded, Multi-client, distributed database with load-balancing.
  - Protect critical sections.
  - Request Sombrero distribute the database.

4.6.1 Virtual Memory Allocation

In Sombrero, all data on disk is viewed as persistent memory that natively maps to virtual memory. Disk storage is simply a larger, slower, nonvolatile cache of the

virtual address space. All the application needs to do is allocate the memory via Sombrero memory allocation operators.

As discussed in Section 4.5.1.1, when virtual memory is allocated, or an object is created, the creator can optionally provide attributes that direct Sombrero to replicate the allocated object or restrict access by other threads of execution.

The Sombrero database class will not share any of its data with other applications, other than via its public methods. Thus, when the database is allocated, it is created, NOTSHARED.

Depending on the phase of functionality (Section 2.4.2) the data migration and replication for allocated virtual memory may be restricted per the design considerations detailed in Section 4.6. For example, the Sombrero phase 1 implementation requires a single set of local data. Thus, the VM allocation primitive must direct Sombrero to not replicate or migrate the database on other nodes.

Another example is that the only difference between phase 3 and 4 in Sombrero are the migration and replication

parameters passed to the virtual memory allocation service (described in Section 4.5.1.1).

In phase 3 (no distribution), when virtual memory for the database is allocated, the database class passes allocation parameters that request Sombrero to not migrate or replicate the data to other nodes. This implies that all threads that act on the data will have to migrate to a single instance of the database. Thus, load balancing and distribution are prevented, satisfying the design considerations called out for phase 3 in Section 4.6.

In phase 4, virtual memory for a database is again allocated, but the server class passes allocation parameters that allow data migration and forces data distribution. The distribution of the database, combined with the ability for threads to migrate, allows the operating system to provide load balancing for the application. Again satisfying the design considerations called out for phase 4 at the end of Section 4.6.

4.6.2 Threads and Synchronization

As previously discussed, Sombrero supports the standard concept of threads, but also allows threads to migrate. Mutex is used in the Sombrero application to provide the following functionality:

- Phase 1: a client locks the passive server for exclusive use with a mutex, thus enforcing exclusive access to a dataset by a single client.
- Phase 2: mutex is used to serialize access of passive server methods by multiple client threads.
- Phases 3 and 4: Mutex is used to protect critical sections but allow parallelism.

5 - Results

This section provides the metrics captured at each phase of Windows and Sombrero development (see Table 2 on the next page). The measurements captured are discussed for each metric.

Comparisons are made between the Windows and Sombrero development efforts. This provides an opportunity to compare the projects phase by phase as they supplied increasing functionality from phase 1 to phase 4. Phase 4 of Sombrero is compared to both the phase 4 and the distributed shared memory (DSM) version of the Windows project.

Comparisons are also made between phases of development in the same operating system. The goal is to examine how much effort is required to supply similar functionality in Sombrero, compared to Windows 2000.

5.1 Results of Sombrero and Windows 2000 Metrics

As can be observed in Table 2, the results, across all metrics, for each phase of development, show that applications developed under Sombrero compared to applications developed under Windows 2000 are:

- Less complex (McCabe’s)
- Require less effort to develop (Halstead, LOC, COCOMO)
- Less likely to have defects (Halstead)

The advantage of Sombrero over Windows 2000 increases as the application feature set requirements increase, approaching that which might be required in a commercial application.

This general result was expected; the amount of the advantage of Sombrero over Windows (2:1 to 3:1 depending on metric) was not expected.

A reason for the large delta between the Sombrero and Windows is that the code in the Sombrero application focused almost exclusively on the pure database constructs such as implementing the required sorting algorithms and data structures.

Table 2

Metrics Captured From Sample Database Application

	Halstead Program Effort	Halstead Bug Prediction	McCabe	LOC	Basic COCOMO
Windows					
Phase 1	2,637,658	6.36	85	852	2.03
Phase 2	4,287,457	8.80	121	1348	3.28
Phase 3	4,420,421	8.98	127	1420	3.46
Phase 4	6,204,183	11.26	145	1612	3.96
DSM[4]	4,987,458	9.73	132	1481	3.62
Sombrero					
Phase 1	1,025,712	3.39	67	639	1.50
Phase 2	1,084,757	3.52	65	647	1.52
Phase 3	1,135,836	3.63	68	719	1.70
Phase 4	1,135,836	3.63	68	719	1.70

[4] This is the sample application under Windows 2000 with distributed shared memory.

The Windows 2000 application had to develop the same database constructs as Sombrero, plus add code to include the needed Windows 2000 WIN32 services (Section 4.2) such as mapped memory and reading and writing state information to the file system. The Windows 2000 application also had to include code to develop the added value services such as virtual memory management and data persistence (Section 4.3).

There is little or no effort in the Sombrero development (Section 4.5.2) required to provide similar services.

The rest of this chapter discusses the significance of each metric and explores the results (Table 2) in terms of the measurements made on each phase of development under the same operating system.

5.1.1 LOC Metric

One might expect as a feature set grows so would the lines of code. In Windows 2000, this assumption holds true. As features are added, the size of the application grows, as reflected in the LOC metric for each phase of Windows development.

In Windows 2000 each additional feature has associated lines of code. In other words, one can trace feature requirements to specific code in the application. For example, phase 2 requires (Section 2.4.2.2) a client – server paradigm. In that phase, a Windows 2000 class is created to provide IPC methods for: creating a client, creating a server, reading and writing messages. The design and implementation details for the communication class are located in Sections 4.2.2.4, 4.3.2 and 4.3.3.

Under Sombrero, there is an increase in LOC for phases 1 through 3 but the change is only minor. There is no change in LOC between phases 3 and 4.

The reason for this is because in most cases, the feature requirements that mapped to code developed by a Windows 2000 programmer, under Sombrero are a natural extension of the operating system. Thus, many of the feature requirements trace to Sombrero not the application, so no sample application code is required (Section 4.5.2).

For example the only difference between Sombrero Phases 1, 2, and 3 are how locks are managed.

- Phase 1: a single lock is used to grant exclusive access to the passive server.
- Phase 2: locks are used to serialize access to passive server methods.
- Phase 3: locks are used to protect critical sections.

The Sombrero operating system satisfies the other requirements. For example, in essence, Phase 3 and 4 are the exact same code base. The only difference is how the database virtual memory is allocated. As described in detail in Section 4.6.1, the feature requirements between phases 3 and 4 are implemented without adding additional code, but rather by passing different parameters to the Sombrero memory allocation primitive.

5.1.2 Halstead's Metrics

Halstead's Software Science metrics track the LOC measurements. Halstead metrics are derived from program size and vocabulary [10]. Thus, as additional features are

added in each phase of windows development, the total number of operators, operands and unique operators and operands increases.

The effect of this, per Halstead metrics, is increased development effort and an increase in the software defects.

The formula for Halstead's Bug Prediction is Effort**(2/3)/3000, as defined by Power Software. The author could not locate updated papers or journals to justify this new metric. It is included as an additional interesting data point.

Another interesting data point is Halstead's estimation of the time required to develop an application based on the calculated Program Effort metric (see Table 2). Per Halstead, development time in seconds is defined as:

Time = Effort (E) / Stroud Moments (S) [5], where S = number of psychological moments per second. A constant of 18 is used for S. However, because of the linear relationship to Program Effort this metric adds no additional information to that already contained in Table 2.

5.1.3 COCOMO

As discussed in Section 2.3.3.2, the COCOMO metric is a tuned LOC metric that is a function of KLOC (thousands of lines of code), an Effort Adjustment Factor (EAF) and development mode factors *a* and *b.* As expected the COCOMO results also parallel the basic LOC count.

5.1.3.1 Basic COCOMO Factors

To calculate Basic COCOMO, the following factors were used. The EAF for basic COCOMO is always 1. Organic mode is selected as the mode, as the sample application is a relatively simple, one-man project.

Table 3

Factors Used for Basic COCOMO

a	*b*	EAF
2.4	1.05	1

5.1.4 McCabe's Cyclomatic Complexity

Recall McCabe's Cyclomatic Complexity metric uses graph theory to measure complexity by counting the number

of paths though an application. Thus, LOC count doesn't affect McCabe's.

However, the net effect of adding features in Windows 2000 is an increase in McCabe's complexity, not unlike the LOC metrics.

As functionality increased, so did the McCabe's metrics for Windows 2000. This implies that the code required to implement the functionality per phase is not "straight line," but rather the developer implemented additional paths through the application, thus increasing complexity and the level of effort required for test and validation.

In Sombrero, this is not the case. Additional features did affect McCabe's, but the effect is subtle. For example, restricting the application to run on a single node, distribute data to all nodes, or distribute data to a set of nodes either did not require any additional logical paths through the application or required few additional logical paths.

For example, the Sombrero developer passes different parameters to a Sombrero memory allocation primitive to effect replication and distribution (detailed in Section

4.6.1). This does not affect McCabe's Cyclomatic Complexity; thus Phase 3 and 4 are identical.

As described in Section 5.1.1, the difference between Phases 1, 2, and 3 is locking. The Sombrero database class added GetLock() and ReleaseLock() methods, which affected the McCabe's metrics. However, adding these private methods to the entry and exit points of the public database methods does not affect McCabe's, as it is straight-line code. For instance consider the following pseudo code:

```
//protect crucial section
Get Write Lock()
Update Database
        .
        .
  .
Release Write Lock()
```

The addition or removal of the locking methods above would not affect McCabe's metric.

6 - Conclusion and Value of Research

6.1 Impacts of Sombrero on Software Life Cycle

As is discussed in Section 5 -, in all aspects of development and design, the software engineering metrics computed for the Sombrero project, showed a significant improvement over those computed for Windows 2000. Thus, it is asserted that for certain classes of applications; Sombrero simplifies the software development process. Hence, reducing costs over the software lifecycle [1]. Applications that can benefit from Sombrero require some, or all of the following characteristics:

- Data persistence
- Data distribution
- Fault tolerance
- Data sharing between processes
- Message passing between processes
- Multi-threading
- Load balancing
- Distributed computing
- Location independence

It should be noted, that not all applications benefit from Sombrero in terms of development effort. However, as discussed in Section 2.2, in all probability there will still be a performance improvement from using Sombrero.

An example of an application that would not benefit from Sombrero, would be “computational intensive” applications like a single threaded math package or perhaps a graphic package. Characteristics of applications that are not likely see improvements with Sombrero are:

- Single threads of execution
- No dependence on data persistence
- Datasets that fit into the available virtual address space
- Minimal data sharing or message passing between processes

At first glance, it may appear that many applications fall into this category, but consider simple applications like a local time service or network time protocol (NTP). In a MASOS a time service requires that the user actively access the node’s time function via a system call that

requires a context switch. The context switch generally includes both a thread and memory context switch. The memory context switch can be quite complex and time consuming especially in segmented architectures such as the 80x86 families.

In Sombrero, the time service is in the applications' address space and is directly accessible, thus eliminating the need to make a system call (and avoiding a context switch). The resulting reduction in application complexity from this example is minimal. However, in large applications, with many such system calls the cumulative savings are significant.

6.1.1 Other Factors

One cannot always quantify development effort via metrics, as other factors can influence the life cycle, for example, factors that impact the developer's psychology.

In "The Subjective Nature of Programming Complexity" [12] McNicholl asserts that perception of complexity influences developers and their development. Because of the tendency for developers to relate complexity to the size of

an application, it may be advantageous to control the volume of a given project.

In his paper, “Programming on an Already Full Brain,” Fry [3], while touting the benefits of developing software using Emacs, proposes that, because of the limits to our biological memory, a simplified programming environment produces an improved development environment. We believe these factors add to the benefits of Sombrero programming.

An additional issue, that would have added to the complexity of the Windows 2000 metrics, is language support for persistent data. To provide this service, the Windows 2000 version of the sample database would have had to included a C++ pre-processor [4]. This is not the case with Sombrero. In fact, the current version of the sample database application developed for this thesis included persistence, as it is provided by the Sombrero operating system, not the database application.

6.1.2 Trends in Modern Computing

The trend in application development is toward distribution and fault tolerant computing. Consider the X-Window System where even simple applications are placed in

a client/server setting to run. Leveraging off experiences with the sample applications developed for this thesis, it is easily asserted that the Sombrero environment is better suited to developing these sorts of applications than is a typical process-oriented operating system.

6.1.3 Conclusion

Over the last decade software applications have become much larger and more complex as a result of the availability of higher performance processors, higher capacity RAM and disk drives, faster and more reliable networking hardware, and advances in distributed computing.

In this environment we believe that a Single Address Space Operating system such as Sombrero provides a major advantage over traditional multiple address space process oriented operating systems for software development. This is because in such an environment applications developed are less complex, thus resulting in lower software development costs and a reduction in defects.

REFERENCES

[1] Banker, R., Datar, S., Kemerer, C., & Zweig, D. (1993, November). *Software Complexity and Maintenance Costs.* Communication of the ACM, (36) No. 11, pp. 81-94.

[2] Boehm, B., (1981). *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, N.J.

[3] Christopher, F., (1997, April) *Programming on an Already Full Brain*. Communication of the ACM, (40) No. 4, pp. 55-64.

[4] Fu, M., (1994). A memory-mapped persistent object oriented system with a concurrent programming and execution environment (Doctoral Dissertation, Arizona State University).

[5] Halstead, R., (1997). *Software Engineering, A Practitioners Approach.* Elsevier North-Holland, New York.

[6] Johnson, K., *Software Cost Estimation: Metrics and Models ,* Department of Computer Science University of Calgary, Retrieved from http://pages.cpsc.ucalgary.ca/~johnsonk/SENG/SENG621/cost.htm

[7] Kearney, J., Sedlmeyer, R., Thompson, W., Gray, M., & Adler, M., (1986, November). *Software Complexity Measurement*. Communication of the ACM, (29) No. 11, pp. 1044–1050.

[8] Kemerer, C. (1987, May). *An Empirical Validation of Software Cost Stimulation Models*. Communication of the ACM, (30) No. 5, pp. 416-429.

[9] Kim, W. (April, 1990). *Research Directions in Object-Oriented Databases Systems*. Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, pp. 1–15.

[10] Mauri, R., & Williams, H. (1982). *Extending Halstead's Software Science for a More Precise Measure of APL*. Proceedings of the APL 82 conference, pp. 207–213.

[11] McCabe, Thomas J., & Butler, C. (1989, December). *Design Complexity Measurement and Testing*. Communications of the ACM (32) No. 12 pp. 1415–1425.

[12] McNicholl, D., & Magel, K. (1982). *The Subjective Nature of Programming Complexity*. Proceedings of the first major conference on Human factors in computers systems, pp. 229–234.

[13] Morrison, R., Dearle, A., Bailey, P., Brown, A., & Atkinson, M. (August, 1985). *The persistent store as an enabling technology for integrated project support environments*. Proceedings of the 8th international conference on Software engineering, pp. 166–172.

[14] Pressman, R., (1997). Software Engineering, A Practitioners Approach, Fourth Edition. McGraw-Hill.

[15] Rothnie, J., Bernstein, P., Fox S., Goodman N., Hammer, M., Landers, T., et al., (March, 1980). *Introduction to a system for distributed databases (SDD-1)*. ACM Transactions on Database Systems (TODS) (5) 1.

[16] Silberschatz, A., & Galvin, P., (1998). *Operating System Concepts*, Fifth Edition. Prentice-Hall, Addison Wesley

[17] Skousen, A., & Miller, D. (2000, June). *The Sombrero Single Address Space Operating System Prototype A Testbed for Evaluating Distributed Persistent System Concepts and Implemenation*. The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000).

[18] Skousen, A., & Miller, D. (1999, February). *Using a Single Address Space Operating System for Distributed Computing and High Performance.* 18th IEEE International Performance, Computing, and Communications Conference (IPCCC’99) Proceedings.

[19] Skousen, A., & Miller, D. (1998, August). *The Sombrero Distributed Single Address Space Project.* 2nd USENIX Windows NT Symposium Proceedings.

[20] Skousen, A. & Miller, D. (1998, October). *Operating System Structure and Processor Architecture for a Large Distributed Single Address Space*. Proceedings of PDCS'98: 10th International Conference on Parallel and Distributed Computing Systems.

[21] Skousen, A, & Miller, D. (1999, January). *Using a Distributed Single Address Space Operating System to Support Modern Cluster Computing*. Proceedings of 32nd Hawaii International Conference on System Sciences, HICSS-32.

[22] Skousen, A., (1994) *Sombrero: a very large single address space distributed operating system.* Master's thesis (MS), Arizona State University, 1994.

[23] Tanenbaum, A., Renesse, R., Staveren, H., Sharp, G., & Mullender, S., (1990, December). *Experiences with the Amoeba distributed operating system*. Communication of the ACM (33) No 12.

ADDITIONAL READING

[24] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., & Zdonik, S. (1989). *The Object-Oriented Database System Manifesto*. Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD '89), Japan

[25] Berry, R., & Schoenborn R. (1992, December). *Estimating Requirements for a Large Software Engineering Project*. Communication of the ACM, pp. 375-383.

[26] Boydston, R., (1984). *Programming Cost Estimate: Is it Reasonable*. Proceedings of the 7th International Conference on Software engineering, pp. 153-159.

[27] Heiser, G., Elphinstone K., Vochteloo, J., Russell S., and Liedtke J. (1998, July). Implementation and Performance of the Mungi Single-Address-Space Operating System. *Software: Practice & Experience*, 28(9), pp. 901-928.

[28] Hornick, Mark, F. & Zdonik, Stanley B. (1987, January) A shared, segmented memory system for an object-oriented database. *ACM Transactions on Information Systems*. (5),No. 1 pp. 70–95.

[29] Patil, Milind H. (1999). *Distributed scheduling in Sombrero, a single address space distributed operating system.* Master's thesis, Computer Science and Engineering Department, Arizona State University.

[30] Skousen, A., & Miller, D. (1996). *Resource Access and Protection in the Sombrero Protection Model, Software Protection Data Structures and Hardware Range Protection Look aside Buffer* (Tech. Rep. No. TR-95-013). ASU 64 Bit OS Group White Paper 2, Computer Science and Engineering Department.

[31] Skousen, A., & Miller, D. (1996). *Implementing a Very Large Single Address Space Across Multiple Nodes: Memory Partitioning, Protection Domain Migration, Kernel Replication, Consistency and Fault Tolerance* (Tech. Rep. No. TR-95-021). ASU 64 Bit OS Group White Paper 10, Computer Science and Engineering Department.

[32] Skousen, A., & Miller, D. (1997, December). *Using a single address space operating system to support client/server distributed computing* (Tech. Rep. No. TR-97-019). Computer Science and Engineering Department, Arizona State University.

[33] Zdonik, Stanley B. (1993) *Incremental database systems: databases from the ground up.* Proceedings of the 1993 ACM SIGMOD international conference on management of database systems, pp. 408–412.

APPENDIX A

DATABASE CLASS USAGE

The following sections provide details of a programmatic interface for the sample database class.

A.1 Database Class Constructor

The constructor creates or opens an existing database.

Usage: Database(int CreateType, long NumOfObj, int NumOfKeys, DBSInfo *MetaData, ServerInfo *ListofServers)

- Input parameters
  - Create type
    - Create new database
    - Open existing database
  - Number of database object. Initially allocate storage for ‘n’ number of objects.
  - Number of sort keys in the database object
  - Metadata describing the database. Metadata is an array of class of type KeyData. Class KeyData has the following public member variables.
- Database name
    - Size of object to store in the database

        - Provide the type, size and offset into the database object of each sort key
        - Sort keys must be simple types: integer, long, float, string and character.
    - List of database servers
    - List of servers that will host the database. If the list has more than one server the database is replicated across all the servers.
- Return value
    - None

A.2 Database Class Destructor

The destructor performs the following functions:

- Release resources
- Preserve database state

A.3 Insert

Insert an object into the database.

Usage: int Insert(char *Object)

- Input parameters

- Pointer to a database object
- Return value
  - Success code

A.4 Search

This method searches the database for an object. Search’s can be by OID, search key or iterative.

Usage: int Search(SEARCHTYPE SearchType, int Key, void *Object, void *SearchKey)

- Input parameters
  - Search Type
    - Next: Iterative search
    - Key: Searches by the value of the search key. Matches the key, or if not matches the object identified by the next higher key.
    - OID: Return the object with the specified OID
  - Key
    - The index to the KeyData class in the metadata array.
  - Search Key
    - Object OID for an OID search

        - Data that matches of the type key being searched
    - Return parameters
        - Object from the database
    - Return value
        - Success code

A.5 Delete

Delete deletes the object with the specified OID or if OID is NULL deletes the object that is returned by the last search.

Usage: int Delete(DWORD oid);

- Input parameters
    - OID
- Return value
    - Success code

A.6 Replace

Replace replaces with the specified OID, if OID is NULL replaces the last object searched.

Usage: int Replace(DWORD Oid, char *Object)

- Input parameters
  - OID
  - Object to replace specified object
- Return value
  - Success code

APPENDIX B

WINDOWS DATABASE CLIENT DESIGN NOTES

B.1 Database Class Constructor

The constructor creates or opens an existing database.

- Client Operation
  - Client sends to server the operation type
  - Client sends to server all input parameters to the first server on the list of database servers.
  - Server returns a unique ID used to associate an instantiation of a client class to an open database
- Private data maintained by the client
  - Unique database ID returned by server
  - List of servers that are hosting the database

B.2 Database Class Destructor

This method informs the server the client object has been deleted

- Client Operation
  - Client sends, to the first server on its list of servers:
    - Operation type
    - Unique database ID

B.3 Insert

Insert an object into the database.

- Client Operation
  - Client sends, to the first server on its list of servers:
    - Operation type
    - Unique database ID
    - Object to be inserted
  - Client reads:
    - Success code from server

B.4 Search

This method searches the database for an object.

- Client Operation
  - Client sends, to the next server (round-robin) on its list of servers:
    - Operation type
    - Unique database ID
    - All input parameters
  - Client reads:
    - Success code from server
    - Database object if search is successful

B.5 Delete

Delete deletes the object with the specified OID or if OID is NULL deletes the object that is returned by the last search.

- Client Operation
  - Client sends, to the first server on its list of servers:
    - Operation type
    - Unique database ID
    - OID
  - Client reads:
    - Success code from server

B.6 Replace

Replace replaces with the specified OID or if OID is NULL replaces the last object searched.

- Client Operation
  - Client sends, to the first server on its list of servers:
    - Operation type
    - Unique database ID
    - OID
    - Object to be inserted

- Client reads
  - Success code from server

APPENDIX C

WINDOWS DATABASE SERVER DESIGN NOTES

The Windows 2000 database server is a standalone application that is comprised of a driver application linked to a database server class.

The driver application is a simple message loop that waits for incoming requests from database clients via TCP/IP. When a request is received a worker thread is created.

The data sent by the client is retrieved by reading the fixed length request type (sent by the client). Once the request type is known, the server knows the format of the rest of the client data. The data is read and then passed to the appropriate database method

The driver application is responsible for maintaining the one-to-one relationship between the database client and an instantiation of a database server class.

The methods provided by the stage 4 database class server are described in the following sections

C.1 Database Class Constructor

Open database requires the driver application provide a database name and for a new database the metadata describing the database object and its search keys. For an existing database only the name is required.

The metadata describing the database and current state are preserved on a database close, thus are available when the database is re-opened.

C.2 Database Class Destructor

This method release database resources and preserves metadata and database state such as the pointers into the free list (list of reusable space in the database).

C.3 Insert

Insert takes a single parameter that is a pointer to an object to be stored in the database. Insert inserts the object and its associated search keys.

C.4 Search

Search requires the user pass a pointer to a structure of the type stored in the database, search type and a search key for search types: KEY and OID. Search types include:

- Key
- OID
- Iterate (Next)

Search returns the requested object and sets the “current object pointer”. This pointer may be used in subsequent delete and replace operations.

C.5 Delete

Delete takes a single parameter, OID. If OID = NULL the object pointer to by the “current object pointer” is deleted, else the object with the specified OID is deleted.

C.6 Replace

Replace takes an OID and a pointer to an object to be stored in the database. If OID is NULL the current object

(“current object pointer”) is replaced. Else the object that matches the OID is replaced.

APPENDIX D

SOMBRERO DATABASE DESIGN NOTES

Issues when designing the Sombrero version of the sample application include the use of the passive server, persistent memory and allowing the scheduler to replicate and distribute data. The following paragraphs present some highlights of these considerations.

Basically, with the exception of adding mutex to lock the object, serialize access or protect critical sections Sombrero is straightforward. For example, there is no effort in terms of design or development for the following items:

- All data that is allocated in the passive server is persistent. There is no need to save state to disk. Data that is allocated on the stack does not persist outside of the subroutine call. malloc(), new(), static data or global data persists.
- No pointer swizzling, memory mapping, mapped memory management
- No message passing. The server’s methods are called directly by the client.
- No load balancing or replication issues. The operating system manages both. The server might

want to provide the operating system with information (i.e. list of servers) at initialization via system calls.

- The client supplies threads to the passive server, thus the server is multithreaded unless steps are taken to serialize execution.
  - Either add mutex to serialize execution or add mutex for critical sections
- The instantiation of the passive server object and database are “tied together”. Thus the name of the database and the name of the passive server object are the same.
  - To use a passive server the client gets the pointer to the passive server object. The Sombrero “old” operator with the object’s name returns a pointer to an existing object, similar to how the “new” operator returns a pointer to a newly created object.

APPENDIX E

DEVELOPMENT AND TEST ENVIRONMENT

E.1 Development Environment

As mentioned in Section 2.4.3, all phases of the sample database, for both Windows 2000 and Sombrero, were developed using Microsoft Visual C++ version 6.0.

The development platform was an Intel based, 2-Way SMP, 550Mhz Pentium IIIs, running Windows 2000.

E.2 Test Environment

The sample application was tested on a network consisting of the development platform and another Intel based Pentium III platform running Windows 2000 (Figure 11).

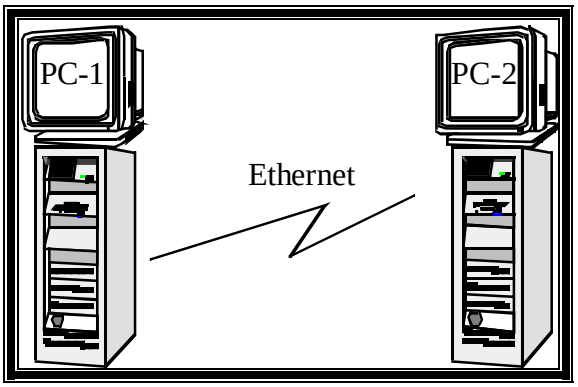


Figure 11 Hardware Used In Testbed

APPENDIX F

WINDOWS 2000 TEST AND MEASUREMENT CONFIGURATIONS

The following appendix describes the run time execution environments used to test each phase of Windows 2000 development and the details on how metrics were generated for each development phase.

For each of the development phases described below, correct execution was obtained in all cases.

F.1 Metrics

Generating the metric data (described in Section 5 -) is a two-step process repeated for each phase of development:

- For each phase of the project, create a single file for all of the code in that phase.
- Use the metrics tool on the single project file to generate metrics for the given phase (Figure 12).

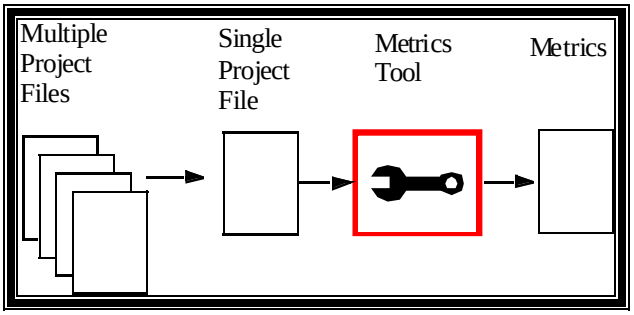


Figure 12: Metrics Generation Process

The test or driver application is not considered part of the database development, thus it is not included in the set of project files used to generate metrics per phase.

The files included for metrics generation are those that provide code to satisfy the specific functional requirements described in Section 2.4.2.

F.2 Phase 1 Test Configuration

Phase 1 of the application, as defined in Section 2.4.2.1, is designed to run on a single platform. The software and hardware configuration is depicted below (Figure 13).

- Database class is linked directly into the single threaded driver application.
- There is a one-to-one relationship between the client/driver application and the dataset.

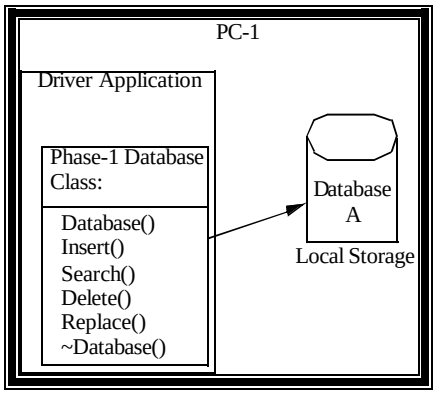


Figure 13: Windows 2000 Phase 1 Database

F.3 Phase 2

Phase 2 of the application as defined in Section 2.4.2.2, provides a single threaded database server that can accommodate multiple database clients.

The software and hardware configuration for phase 2 is described below.

- Database client class is linked to a single threaded driver application (Figure 14)
  - Client class provides access to the database server via its public methods.
- Database server is a stand-alone single threaded application (Figure 15).
  - Listens for request from multiple clients
  - Allows multiple clients to access same data

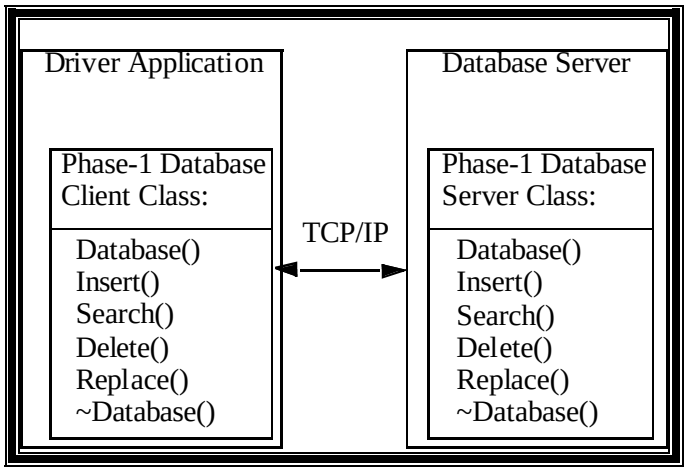


Figure 14: Client / Server Relationship

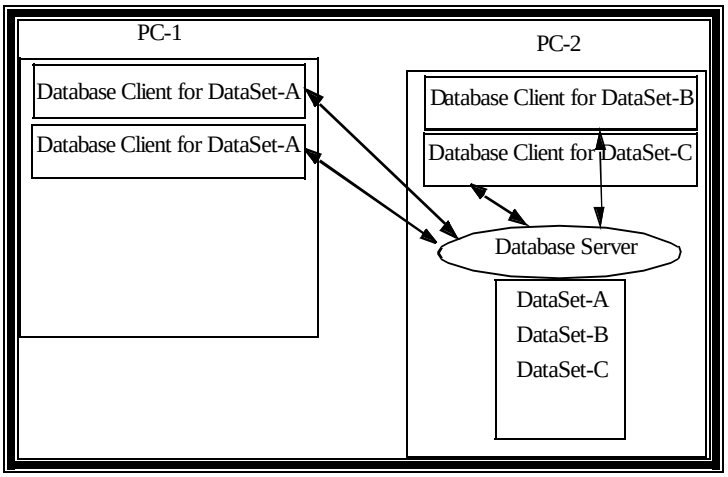


Figure 15: 1 DBS Server, 3 Datasets, 4 DBS Clients

F.4 Phase 3

The configuration for testing phase 3 is identical to phase 2. Phase 3, as defined in Section 2.4.2.3, adds

multithreading to the database server (Figure 16). There are no changes to the database client.

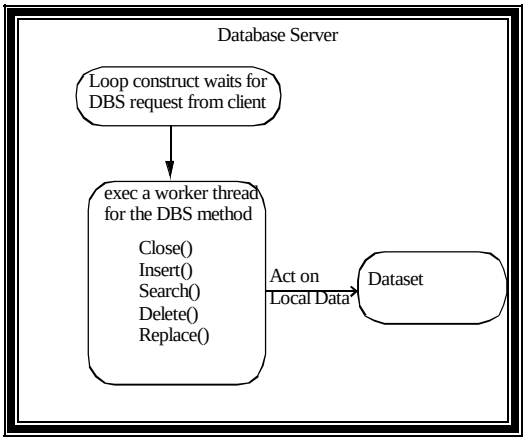


Figure 16 Multithreaded Database Server

F.5 Phase 4

Phase 4, as defined in Section 2.4.2.4, adds data distribution to the phase 3 database server (Figure 5) and load balancing (Figure 4) to the phase 3 database client class. The test configuration for phase 4 is shown in Figure 17.on the next page

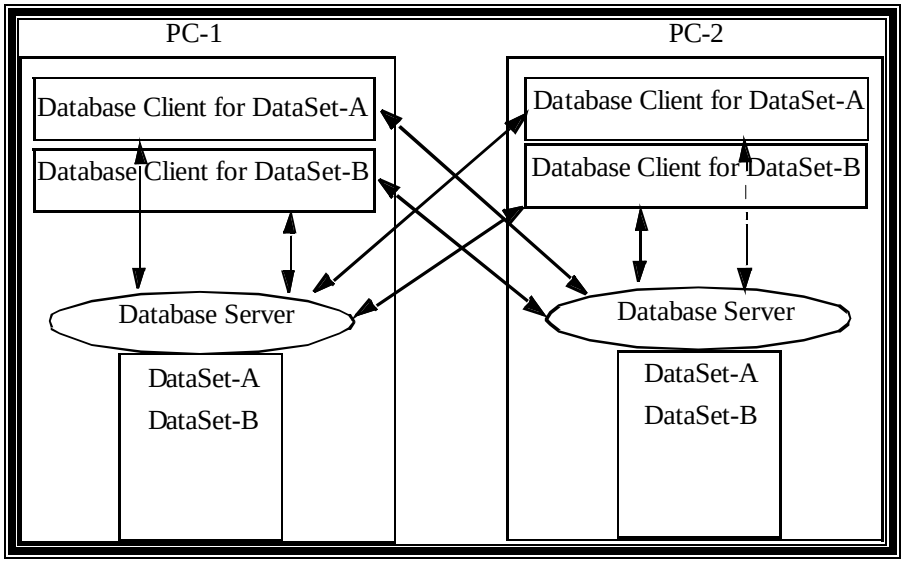


Figure 17: 2 DBS Servers, 2 Datasets, 4 DBS Clients

APPENDIX G

SEPARATELY BOUND APPENDICES

G.1 Code Base

A separately bound appendix contains the source code listings used for each phase of development. The source code is organized as follows:

1. Windows 2000 Phase 1
2. Windows 2000 Phase 2
3. Windows 2000 Phase 3
4. Windows 2000 Phase 4
5. Windows 2000 Phase 5
6. Sombrero 2000 Phase 1
7. Sombrero 2000 Phase 2
8. Sombrero 2000 Phase 3
9. Sombrero 2000 Phase 4

G.2 Raw Metrics

The evaluation version of Karatuku Profession Metrics from Power Software did not include the ability to output the data to a printer. In addition the evaluation version would only calculate McCabe's metrics per class and Halstead metrics were produced per file.

The raw output from the tool, for each phase of development was entered into an Excel spreadsheet. Excel was used to calculate COCOMO and sum the McCabe's metrics for the entire project.

A separately bound appendix contains the printed version of the Excel spreadsheet containing all of the discrete measurements made per each phase of development. These measurements where summarized and organized into Table 2.