DISTRIBUTED CONSISTENCY MANAGEMENT IN SOMBRERO, A SINGLE ADDRESS SPACE DISTRIBUTED OPERATING SYSTEM

by

John Olson

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

ARIZONA STATE UNIVERSITY

December 2002

DISTRIBUTED CONSISTENCY MANAGEMENT IN SOMBRERO,

A SINGLE ADDRESS SPACE DISTRIBUTED OPERATING SYSTEM

by

John Olson

has been approved

August 2002

APPROVED:

_____, Chair

_____

_____

Supervisory Committee

ACCEPTED:

_____

Department Chair

_____

Dean, Graduate College

ABSTRACT

Considerable software complexity and performance improvements in the support for distributed consistency management are possible in a single address space. These improvements stem from the fact that a single address space operating system inherently and transparently can maintain the uniqueness of a single virtual address namespace.

The Sombrero Single Address Space Operating System design includes built-in features to support distributed consistency management that make use of this fact. These features include augmented hardware support for protection; extending the single virtual address space across all levels of the memory hierarchy and across all nodes in a distributed system; the control of memory on an object, rather than a page, basis; additional support for thread migration; and the flat non-hierarchical peer-level design of Sombrero services and user programs.

This thesis presents the basic architecture of consistency management in Sombrero and describes it in terms of the roles of the major participating services and the communications between them. These services include the Protection, Communications, and Paging Services and the Consistency Management Service itself. The purpose and roles of the data structures and algorithms that support distributed consistency management and their relation to other Sombrero data structures are presented. The reasons why consistency management appears to be simpler and promises improved performance in Sombrero are parsed. Sombrero's consistency management is also compared with that provided by conventional middleware Distributed Shared Memory systems running on multiple address space operating systems.

A simulation using these algorithms and data structures to provide sequential consistency and a weaker consistency across a distributed system was developed under Windows 2000 to show implementation feasibility and to observe the dynamic behavior of the consistency algorithms.

To My Family

ACKNOWLEDGEMENTS

Many thanks for the effort and patient support of my committee chair Dr. Donald Miller. Thanks to committee members Dr. Partha Dasgupta and Dr. Bruce Millard for their patience. Thanks to Dr. Miller, and Alan and Don of our research group, for the generous amount of time they spent reviewing this work. Special thanks to Ron Feigen for his tireless encouragement, for without it this day might not have come.

TABLE OF CONTENTS

Page

LIST OF FIGURES……...…. x

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND 3

2.1 Sombrero 3

2.2 Consistency Models 4

2.2.1 Strict Consistency 4

2.2.2 Sequential Consistency 4

2.2.3 Weak Consistency Models 5

2.2.4 Multiple Reader Multiple Writer 6

2.2.5 Object-Based DSM 8

2.3 Comparison to Distributed Shared Memory (DSM) 8

2.3.1 Single Namespace for Data and Communications 9

2.3.2 Middleware Not Required For DSM in Sombrero 9

3 SOMBRERO FEATURES PERTINENT TO CONSISTENCY MANAGEMENT 10

3.1 Basic Abstractions 10

3.2 Data Structures Overview 11

3.3 Protection 14

3.4 Consistency 15

3.5 Pages 16

3.6 Page Invalidates 17

CHAPTER Page

4 CONSISTENCY MANAGEMENT IN SOMBRERO 19

4.1 Data Structures 19

4.1.1 Memory Object Control Block 19

4.1.2 Consistency Management Data 22

4.1.3 The Copy Set Graphs 22

4.1.4 The Last-Known-Writer and Modified-Page-Cache Graphs 29

4.2 "Token Tracking" 41

4.2.1 Design 43

4.2.2 Scenarios 44

4.2.3 Algorithms 51

4.2.4 Message Handlers 53

5 COMPARISON OF DSM AND SOMBRERO 55

5.1 Consistency Policy Supported Per-Object 55

5.2 Copy Set Managed by System 56

5.3 False Sharing Problem Exists in Sombrero and Conventional DSM 57

6 IMPLEMENTATION OF SIMULATION 59

6.1 General Design 59

6.2 Modeling Consistency 63

6.2.1 Sequential Consistency 63

6.2.2 Weaker Consistency 64

6.3 Differences from Real Sombrero 64

CHAPTER Page

7 RESULTS 66

7.1 Overview 66

7.2 Sequential 66

7.3 MRMW 70

8 CONSISTENCY UNDER THREAD MIGRATION 71

8.1 Description of Sombrero and Conventional Systems 71

8.2 Description of Problem 72

8.3 Scenarios 73

8.4 Solutions 76

9 CONCLUSION AND VALUE OF RESEARCH 78

9.1 Direct Observations from Simulation 78

9.1.1 Passive Servers 78

9.1.2 Synchronization 78

9.1.3 Language Support 79

9.2 Related Observations 79

9.2.1 Paradigm Shift 79

9.2.2 Hardware Control 80

9.3 Optimizations to Consider and Areas of Further Research 81

REFERENCES 82

APPENDIX Page

A GLOSSARY AND LIST OF ABBREVIATIONS 86

B SELECTED FIGURES 88

LIST OF FIGURES

Figure Page

1. Request Access 15
2. Page Miss After Token Set 17
3. Simplified Memory Object Control Block 21
4. Copy Set Graph with Arity 2 24
5. Copy Set Parent Graph..... 27
6. Copy Set Parent Graph Depicted As Linear Graphs..... 28
7. Last Known Writer Graph 30
8. Extended Last Known Writer Graph 30
9. Pruned Last Known Writer Graph 31
10. Extended Pruned Last Known Writer Graph 32
11. Modified-Page Cache Graph..... 33
12. Process of Pruning Modified Page Cache Graph..... 34
13. Pruned Modified Page Cache Graph..... 34
14. Page Flushing upon MPC Graph Pruning..... 36
15. Overlay View of Four Graphs..... 39
16. Last Known Writer Graph of Figure 10 After Node Deletion..... 40
17. Delete Node from Last Known Writer Graph Algorithm..... 40
18. Delete Node From Modified-Page Cache Graph Algorithm 41
19. Current Sombrero Architecture on a Single Node..... 44
20. Token Tracking Invocation Scenarios 50
21. Request Write Permission Algorithm 51
22. Send Modified Pages to Peer Algorithm 52
23. Receive Request for Write Token from Peer Algorithm 53
24. Receive Modified Pages from Peer Algorithm..... 54

Figure Page

25. Handle Local MPC Graph Pruning 54
26. Sombrero Simulation Implementation Architecture 60
27. Sombrero to Simulation Mapping 63
28. Single Node Access 66
29. Two Node Access 67
30. Three Node Scenario 68
31. Three Node Multiple Pages 69
32. MRMW Three Node Scenario 70
33. Thread Migration Scenarios on a Memory Object 75

1 INTRODUCTION

This thesis describes research into consistency management features of the Sombrero Operating System [Skousen *Thesis*]. Sombrero is a Distributed Very Large Single Address Space Operating System.

Sombrero uses a single virtual address space to name all objects, data, and code; all services; and all threads in a distributed network of homogeneous computers. This allows persistence, caching, and distribution to be transparent. As with all replicated and distributed data, enforcement of a consistency policy is required for the data to be predictable and usable. This thesis is concerned with the correctness of the mechanism of the Sombrero Consistency Service [Skousen/Miller Using a Single Address Space Operating System for Distributed Computing…]. It also is concerned with the capability of Sombrero to support other policy managers.

This thesis presents Sombrero's basic consistency management architecture, the roles of the major participating services and the communication between them. These services include the Protection Service, Communications Service, Paging Service as well as the Consistency Management Service. This thesis also presents the purpose and roles of the data structures and algorithms that support distributed consistency management and their relationship to other Sombrero data structures. Additionally, this thesis examines the reasons why consistency management appears to be simpler and promises improved performance in Sombrero. It also compares Sombrero's consistency management with that provided by conventional middleware Distributed Shared Memory systems running on multiple-address space operating systems.

A simulation using these algorithms and data structures to provide sequential consistency and a weaker consistency across a distributed system was developed under Windows 2000 to show implementation feasibility and to observe the dynamic behavior

of the consistency algorithms. The simulation takes advantage of the multiple address spaces supported by Windows 2000 to isolate simulated Sombrero Nodes from each other. This allows the complete control of the interaction between simulated nodes which would be the case of Sombrero controlling a real Sombrero network.

While the implementation of Sombrero may differ from the simulation, the degree of isolation of the simulated nodes hopefully should allow some of the algorithm code to be used in Sombrero. In any event, the simulation should aid implementation of Sombrero. The simulation should help identify any implementation pitfalls, and should help better identify the relationship and interaction between services including consistency, paging, communications and protection.

2 BACKGROUND

2.1 Sombrero

In recent years, 64-bit address spaces have become supported by microprocessors. Consequently, the huge number of virtual addresses available makes possible the notion of a single distributed virtual address space. All stock processors, however, are tailored to support Multiple Virtual Address Space Operating Systems (MASOSs). Opal [Chase dissertation] and Mungi [Heiser, Elphinstone, et al], are examples of designs for Single Address Space Operating Systems (SASOSs) that run on stock processors. These designs depend on software support for protected access to the single virtual address space. Protection in Sombrero is orthogonal to addressing and is enforced with hardware assistance [Skousen, Miller, Operating System Structure...]. The advantage to this is the ability to deploy traditionally system- or kernel-level services as user services, and the hardware-assisted Protection Service guards the integrity of the system.

Sombrero provides a common namespace translation in any part of the memory hierarchy. The memory caching hierarchy extends from the on-chip level-one cache down to the hard disk and spans the entire network. Memory, memory addresses (pointers), hardware addresses (devices), function addresses, and threads all have universal, unique names throughout every level of every node, in a Sombrero network. There is no requirement for a classic file system and for explicit methods for making data persist. As data is paged out, it is pushed down the hierarchy to persistent storage. Sombrero, also, obviates address or namespace translation across the network. Other than the binding of a hardware (e.g. MAC) address to a node address, the virtual address in a network communication can always determine – without translation – the network address.

2.2 Consistency Models

Whether data is distributed across computers using a namespace, as does middleware Distributed Shared Memory, or simply replicated in different locations, such as a database might be, definition and enforcement of a consistency policy is required. There are many consistency models of varying popularity. Strict, Sequential, Causal, Weak, Release, and CREW are examples [Tanenbaum DOS] [Don MacLaughlin PhD]. Consistency policy is, in essence, a contract between the application software and the underlying hardware and operating system [Tanenbaum 315]. Typically, the weaker the consistency model of a distributed system, the better the performance and the more difficult the job of the programmer. Any weakening of consistency relaxes, to some degree, the requirement that every read see the result of the most recent write.

2.2.1 Strict Consistency

In Strict Consistency, every read sees the result of the most recent write [Tanenbaum DOS 315]. Strict Consistency is the strongest possible consistency model and is what happens automatically on single processor, single threaded computers, such as a PC running MS-DOS with no DMA or other access to memory besides the CPU. Every memory access must complete in real time before the next one can commence. As soon as any other access to memory is provided, such as DMA or additional CPUs, or when memory is distributed, Strict Consistency becomes difficult if not impossible to implement.

2.2.2 Sequential Consistency

Possibly the most popular consistency policy in distributed systems is Sequential Consistency, which gives the appearance of strict consistency, provides the consistency to which programmers have become accustomed, and is feasible to implement on a

distributed system. Sequential Consistency requires that all processors in a system must see the same sequence of memory accesses [Tanenbaum DOS 317]. Sequential Consistency, slightly simplified, provides that all readers will always see the same sequence of writes – no matter on which node the read or write takes place. A system for managing consistency in Sombrero is presented in [Skousen/Miller Using a Single Address Space Operating System for Distributed Computing…] and [Skousen thesis]. That system was simulated for this thesis and will receive the most analysis.

2.2.3 Weak Consistency Models

Performance in a distributed system can be enhanced by relaxing the consistency model. Weaker consistency models allow concurrent access to a part of memory and guarantee consistency only after some consistency barrier is reached. Weaker models require more involvement by the application programmer. Some of the more general models are discussed in [Tanenbaum DOS p321-333]. There are specialized models, as well, which are suitable for a particular type of computing.

One specialized model, for example, Concurrent Read Exclusive Write (CREW) [McLaughlin Diss], allows pages of the virtual address space to be written, simultaneously, by more than one node or task, without classical mutual exclusion. Mutual Exclusion is achieved by assigning an exclusive range of addresses to which an individual writer can write. The writes by each task are not visible by other tasks until a barrier synchronization at which time the various writes are merged. The application is responsible for maintaining consistency of simultaneous updates to different locations on the same page. It is also responsible for maintaining consistency of simultaneous updates to the same locations on the same page. An example of the latter is defining the contents

of a set of locations on a page as those returned by the first thread to complete processing of a page (eager scheduling).

A more common example is Release Consistency which allows reads at any time but only guarantees the results to be that of the latest write, after a writing process has released its lock. In addition, it's necessary for a reading process to acquire the lock before it reads the variable and after the writer acquired it. Some other models, general and specialized, are Entry Consistency [Tanenbaum DOS p354], Scope Consistency [Iftode, Singh, et al 1996] which require the application programmer to control synchronization to compensate for the weaker relationship between reads and writes.

2.2.4 Multiple Reader Multiple Writer

In Sombrero, the consistency model we call Multiple Reader Multiple Writer (MRMW) is a completely relaxed consistency model that allows complete access to an address on a node without regard to changes at the same Virtual Address on another node. Data never migrates with MRMW, although the addresses covered by MRMW are accessible on every node. Essentially, data covered by MRMW policy is local data even though the Virtual Addresses are valid on all nodes. The data on a particular node is accessible from the local node only and is guaranteed never to migrate. Note that this policy does not preclude the need for the use of the usual synchronization mechanisms between threads running on the same node. To restate the effect of Sombrero's MRMW consistency policy in a single distributed virtual address space, data written to a particular Virtual Address remains locally on the writing node and never replicates to other nodes, but the Virtual Address is accessible on all nodes.

Some data in a distributed system is not distributed. It might be intended for the management of the system and, therefore, has to be unique. In Sombrero, memory

management data blocks, called Memory Object Control Blocks (MOCBs) [Skousen/Miller Using a Single Address Space Operating System for Distributed Computing…], must never be distributed or replicated. A unique MOCB exists on each node that participates in access to an object. Requests to access this data must always be restricted to the local node. In Sombrero, we call this consistency model *Isolated Consistency*. Isolated Consistency is similar to Multiple Reader Multiple Writer (MRMW) Consistency in that it is guaranteed that no data will replicate, and local access to the data will never be revoked because of consistency policy. Legal attempts to access this data always succeed on the node where it is defined and there is never a case when the data has to be synchronized. The difference between Isolated and MRMW consistency is that, with Isolated Consistency, the data space pointed to by the Virtual Address is only valid on one node. In this case, neither the data nor the address space migrates. In Sombrero, Isolated Consistency comes about because the Protection Service limits access to some MOCBs to the local node. This access restriction is not an attribute of the memory object or its consistency policy. The policy of a local Protection Domain that owns the Memory Object prevents it and its resources from migrating across nodes. Summarizing, Isolated Consistency is a policy used by the Protection Service on each node to implement protection. Only the local protection service can access these VAs. It is similar to MRMW except that the VAs cannot be accessed on other nodes. An example of its use is the consistency policy for local MOCB copies used by a local Protection Service. The data at these VAs is maintained by the Protection Service on the given node and nothing else can exist at these addresses on any other node (the VAs in which MOCBs are stored are unique per MOCB and must be in the range allocated to a given node). (Protection Domains are defined in Section 3.1.)

2.2.5 Object-Based DSM

Sombrero has an intrinsic memory model and allocates all memory as memory objects, so it is worth mentioning Object-Based DSM [Tanenbaum DOS p356]. With Object-Based DSM, access to object memory may only be made via the object's methods. This allows enforcement of protection policy, which is what Sombrero does implicitly though its Protection Domains. Access to Sombrero memory implicitly consults the protection and consistency policy managers before allowing the success of any request to read or write.

2.3 Comparison to Distributed Shared Memory (DSM)

A principal advantage of Sombrero's support for distributed consistency management is its scalability. It does not rely on broadcasts, but can route communications calls directly to the node desired, because any virtual address, when used as a network destination address, points to a unique node. The node address allocation range is also its routing table entry. As a network grows to hundreds or thousands of nodes, a broadcast would cause an interrupt in all of the nodes. Additionally, scalability is enhanced since each Node must only keep a limited amount of information including is relative position in the network. Each Node relies on communications calls to the few nodes of which it is aware, the Copy Set Parent and Child and those nodes contained in the other graphs, to convey the necessary information to complete a task. Scalability is inhibited in systems that broadcast as part of the communication required to maintain consistency. Additionally, scalability is inhibited in systems that require each node to keep information about the entire network. As such a network grows, scalability would be limited by the space needed for the data and the

overhead needed to maintain consistency. DSM systems typically require one of these methods. Sombrero requires neither.

2.3.1 Single Namespace for Data and Communications

Sombrero uses the same namespace for communications and memory. For example, when communicating with other nodes, the Virtual Address (VA) of a Memory Object Control Block (MOCB) can be used as a destination address to communicate with its Protection Service located on the network. (MOCBs are described in Section 3.1. The Sombrero Communications Service uses the VA directly to route message. Some research into the feasibility of using VAs for routing was done in [Patil thesis].

2.3.2 Middleware Not Required For DSM in Sombrero

There is no additional software that must be licensed, installed, or maintained on each node to support memory distribution in Sombrero. Every Sombrero node intrinsically includes this support.

3 SOMBRERO FEATURES PERTINENT TO CONSISTENCY MANAGEMENT

A memory access occurs when any thread tries to access a virtual address (VA)for reading, writing or execution. A memory access can come from various sources when programs include library calls like *read()*, *write()*, *exec()*, *getc()*, *puts()*, language operators like assignment (=) and pointer dereference (->, *) and simple subroutine calls. The Write access is the most complex type of access. Therefore, its operation is the example outlined in this section. The final Sombrero data structures, and protection, consistency management, paging and communications algorithms that are implemented could vary from what is outlined below. The pertinent parts of Sombrero are introduced here in Section 3 , and covered in greater detail in Section 4.1.

3.1 Basic Abstractions

A *Memory Object* (MO)is characterized by *Range* - a distinct, variable sized, contiguous allocation of the virtual address space. The name of the MO is the beginning Virtual Address of the MO's Range. A Range contains *Memory Regions*, sets of Virtual Addresses within a Range with Non-Overlapping Protection Attributes (Read/Write/Execute). A *Protection Domain* is a set of Memory Objects and Protection Domains which are reachable by a Thread executing within it, and their Associated Access Permissions. There are General and Carrier Protection Domains (*GPD*s and *CPD*s). A GPD is code, data and other GPDs accessible to all threads within it. A CPD is data and GPDs privately accessible to an individual Thread. A *Thread* is the state of a Computation represented by the current or most relevant CPU Register State (PC, SP, General Registers, etc.), its reachable CPD Memory, and Status Information. A *Principal* is a unique protected identity that represents a user or a system service which binds resources and activities to an owning user.

This thesis makes frequent use of the term "*Copy Set*." The copy set is the set of nodes which are participating in the access to an object.

3.2 Data Structures Overview

The nomenclature of Sombrero Data Structures is introduced here along with brief descriptions of each. They are covered in greater detail in sections 4.1, 4.1.3, and 4.1.4. All of them are defined as well in [Skousen/Miller Using a Single Address Space Operating System for Distributed Computing…].

A *Memory Object Control Block* (MOCB) is a small data structure that contains information about one of the basic abstractions, the Memory Object. The MOCB contains, among other things, the Virtual Addresses defining the Range of the Memory Object and pointers to access control information. The access control information includes, by formal definition, the Access Control List and one or more *Region Control Blocks*, discussed below.

The *Root Node* of a MO is the node on which the MO was created. The Virtual Address Range of an MO and the Virtual Address of the MO's MOCB on the Root Node are both within the Virtual Address Range of the Node. The MOCB on an MO's Root Node is called the *Root Memory Object Control Block* (RMOCB[1]) for an MO. An RMOCB is an MOCB in every sense and is special only by its location in the network. The Root Node is also the head of the *Copy Set Graph*, described below.

Since memory objects are distributed in Sombrero, each node which participates in access to an MO must have an MOCB for that object. The MOCBs that exist for an MO, beyond the first, are called *Surrogates.* The name of the MO, which is the

[1] Note that the term RMOCB was used to describe the control block managing an entire node in early Sombrero papers and presentations. That data structure is now called a *Node Control Block* (NCB). What is now called the RMOCB was, earlier, referred to as "the MOCB on the inception node" or something similar.

beginning Virtual Address of the MO's range, is contained in every MOCB for an object. The group of Nodes holding MOCBs for an object is called the *Copy Set*. The members of a MOs Copy Set all hold MOCBs for an object. The VAs of the MO's Range of Memory is also in every MOCB for an object as is access control information. However, unlike the Name and Range, the access control information can vary on each node. Sometimes, nodes participating in access to a MO need to communicate between each other in an orderly way. To facilitate this, each MOCB has several node-specific fields which hold the addresses of some of the MO's MOCBs on other nodes. The MOCBs serve as vertices and the pointers as edges of inter-nodal, distributed, graphs which are used by Sombrero and can be used by other programs.

All the data in an MOCB could be likened to private class data because they are only accessed by method calls to the Protection Service.

The *Region Control Blocks* (RCB) are pointed to by the MOCB and are used in conjunction with the MOCB to manage the object on an access-type basis (Read, Write, Execute).

The *Range Protection Lookaside Buffer* (RPLB) is a hardware device, which takes pairs of Virtual Addresses and Access Requests and compares them to cached pairs of Regions and Access Privileges. Like a TLB, it provides protected access during the CPU instruction pipeline, but it does not provide address translation.

Protection and Resource Access Lists (PRALs) are lists that link Memory Objects, Protection Domains and Threads for the purpose of defining access permissions. PRALs are used as information sources for loading the RPLB. Information in the PRALs serve as Resource Access Control Lists or Protection Domain Access Lists depending on the need [Skousen, Miller, Operating System Structure...].

Each MOCB contains some small number, e.g. 2 or 3, of VAs called the *Copy Set Graph Pointers*. This small number is the Copy Set Graph arity and is the same for every MOCB of a given MO. Each of these pointers is the VA of another MOCB for the same object. Collectively, these pointers and the MOCBs they point to, form the *Copy Set Graph*, a distributed, inter-nodal tree starting at the Copy Set Root Node and extending to all of the Copy Set members through its children.

The *Copy Set Parent Graph* is the reverse of the Copy Set Graph. It starts at each leaf of the Copy Set graph and extends to the Copy Set Root Node and is comprised of the Copy Set Parent Pointers in the MOCBs of an object. Each MOCB contains exactly one Copy Set Parent Pointer.

The *Last Known Writer* (LKW) *Graph*[2] is a distributed graph comprised of the LKW pointers in an object's MOCBs. These pointers are, similarly, the VAs of other MOCBs for the same object and collectively form an inter-nodal, distributed, graph. Following an LKW graph being used for a single writer consistency policy to its end will locate the node that most recently wrote to an object. When a node sends the write enabling token to another node, the LKW Graph Pointer of the local MOCB is set to the Virtual Address of the node to which the write enabling token is being sent.

The *Modified Page Cache* (MPC) *Graph*[3] is a distributed graph formed from the MPC pointers in an MO's MOCBs. As is the case for the other graphs described previously, these pointers are also the VAs of other MOCBs for the same object. These VAs collectively form an inter-nodal, distributed, linear graph. This linear graph links nodes which, when taken together, contain the latest copies of all the pages which have

---

[2] Note that the LKW Graph was named the "Write Last Graph" in early Sombrero papers and presentations.
[3] Note that the MPC Graph was named the "Write Cache Graph" in early Sombrero papers and presentations.

been written for an object. The MPC Graph starts at the current LKW node and provides a scalable, distributed method for locating the latest copy of a page.

3.3 Protection

Sombrero protection is applied to the previously described basic abstractions called Protection Domains. Again, a Protection Domain is a set of Memory Objects and Protection Domains reachable by a thread executing within it along with their associated access permissions. There are two types of Protection Domains, General Protection Domains (GPDs) and Carrier Protection Domains (CPDs).

On every memory access on a given node, the node's RPLB is referenced to determine if the accessing thread has permission to access the Memory Object containing the requested address. If there is an entry in the RPLB, an enabling token is present in the MOCB by definition. In that case, access is granted immediately. If there is no entry in the RPLB, the Protection Service is invoked to search for access permission in the PRAL pointed to in the MOCB. A PRAL search will discover whether the accessing thread's GPD or CPD has rights for the requested access type on the given Memory Object (MO).

If access is allowed, the Protection Service checks the MOCB to see whether an enabling token is present. If the token is present, the RPLB is loaded with the needed protection entry and the memory access can be restarted. However, if a token is not present, the Consistency Service registered for this MO must be consulted. See Figure 1 for a typical case of fetching the token for a single-writer-policy MO.

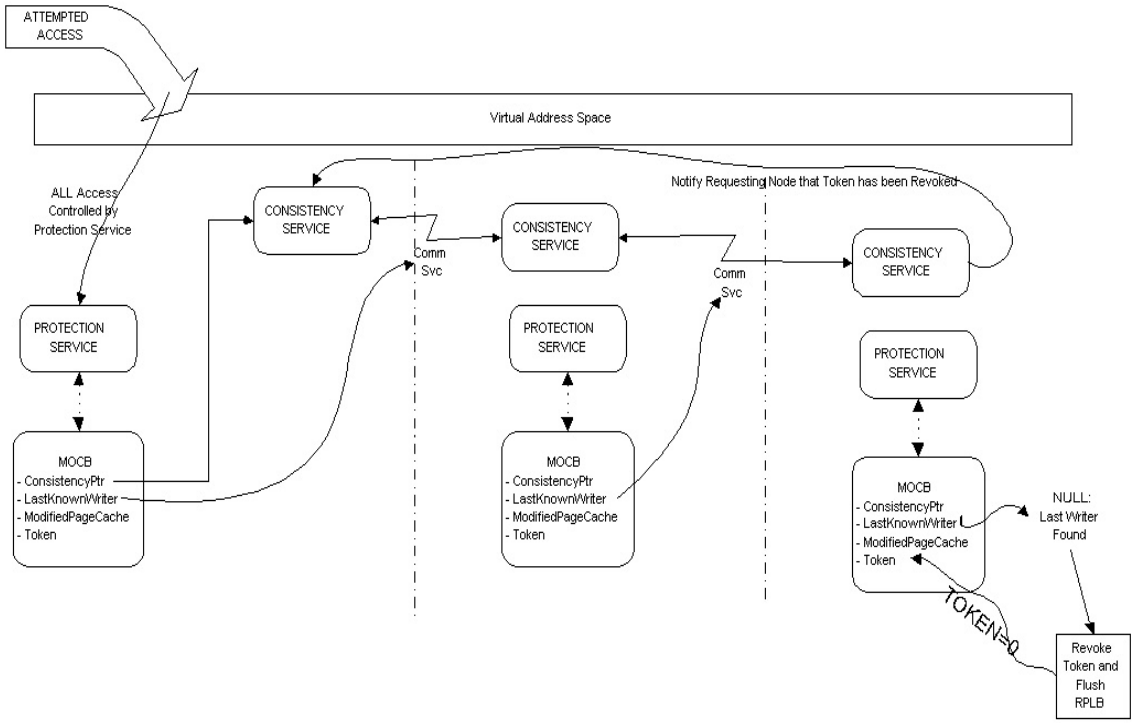


Figure 1. Request Access

3.4 Consistency

Sombrero includes a default consistency mechanism that supports Sequential Consistency called *Token Tracking*. Token Tracking utilizes the MPC Graph and the LKW Graph when processing calls to find pages and grant and revoke tokens. Sombrero also inherently supports MRMW Consistency. Once the service for a particular consistency policy is registered in the MOCB, the basic calls are the same regardless of policy. A custom policy manager program is not prohibited from supplying additional methods, but the basic interface methods must be supported by each.

The MOCB contains a pointer (VA) to the entry point for the Consistency Service (Figure 3. Simplified Memory Object Control Block). All Consistency Services must have the same interface, and the entry point is similar to the top of a *Virtual Function Table* (vTable) [Troelsen pg 31] enabling polymorphic behavior through an abstract

interface with a single set of defined method calls. By virtue of this, calls to the Consistency Service can be compared calls to an abstract interface. The interface is well-known, and the implementation is hidden. Note that the Consistency Service is distributed and its address is the same throughout the network and is the same in every MOCB for an MO.

Whenever the token is missing after permission to access has been established, the Protection Service invokes *request_access(Write)* using a well-known offset from the Consistency Service pointer in the MOCB. The calling node then blocks or stalls the *write()* until the Consistency Service grants the token on the requesting node. If the Consistency Service is MRMW, it may immediately set the token. Or the Consistency Service may have to contact other nodes to do other work prior to setting the token. This is transparent to the caller. To the caller, every Consistency Service behaves the same way.

3.5 Pages

When the access is restarted, if the needed page is not in the processor cache, it will have to be fetched by the Paging Service. The Paging Service searches the local cache hierarchy which includes all levels of storage including the processor cache levels, RAM, and the hard drive. If the page is available locally, it is paged in and, if necessary, loaded into the processor cache. Then the access is restarted.

If the page is not available locally, the Paging Service contacts the Protection Service which contacts the Consistency Service using the Consistency Service pointer referred to earlier. The Consistency Service then traverses the MPC graph by sending messages to the node pointed to by the MPC graph pointer (See Figure 2). Each node contacts its local Paging Service to see if the page is available locally. If the page is

available, it is guaranteed to be the latest. When the page is found, the search ends and the page is forwarded by the Consistency Service on the node containing the page to the peer Consistency Service on the requesting node. The receiving Consistency Service then removes the pages from the network buffers and sends them to the local Paging Service to handle. The access is then restarted.

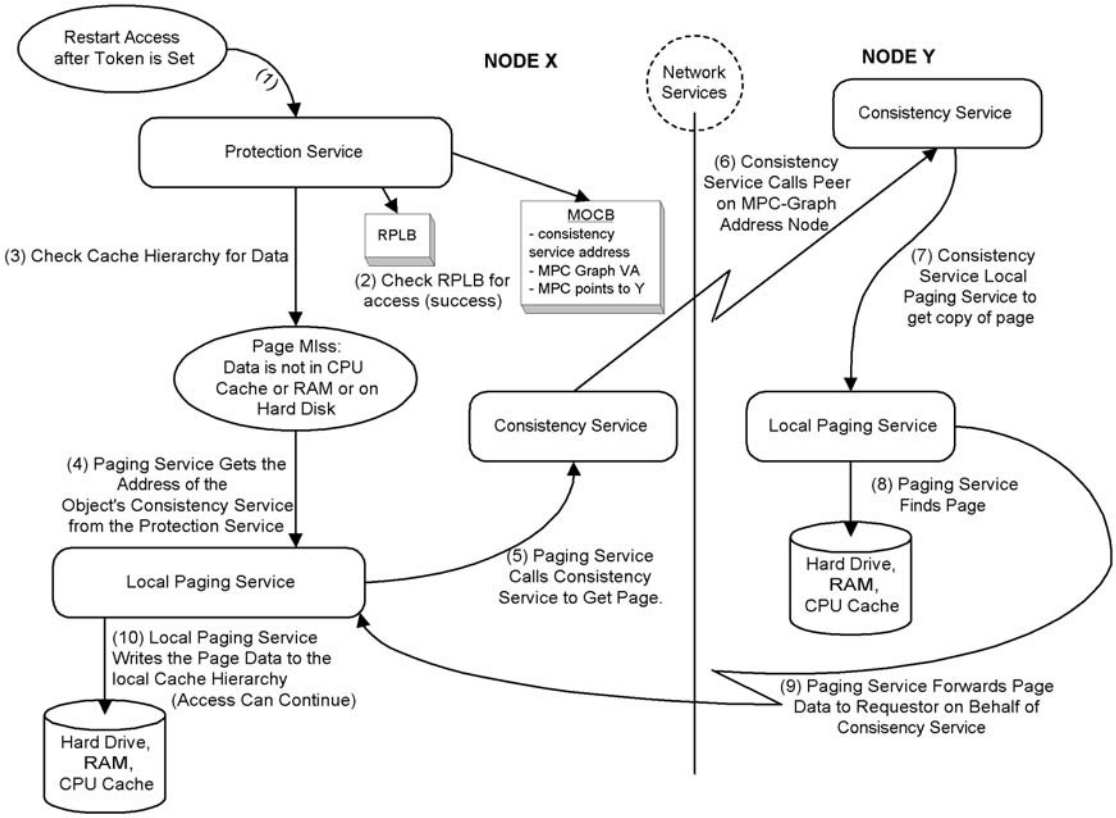


Figure 2. Page Miss After Token Set

3.6 Page Invalidates

Aside from limitations caused by the difficulties in agreeing on global time and the finite speed of light, strict consistency can be maintained if, prior to a write, all read tokens for the MO are revoked, or if all copies of the page containing the target address of the write are invalidated. All **read()** operations will see the *result* of the most recent

completed write, not the most recent *request* to write. The request to write is stalled until the enabling token exists on the writing node. This is easy to enforce with Token Tracking because there is only one write enabling token for an address. Additionally, the graphs give sufficient information for control and communications between members of the copy set. Token transfer and page invalidates can always take place without the need for a central manager. Prior to the granting of a write token, the Consistency Service sends a message to the root node requesting that the page be invalidated by the entire copy set. Every node in the copy set invalidates the page, and sends a message to its children on the copy set. When all children have acknowledged the invalidate operation, each parent node reports the same to its parent. Finally, when the root node receives notification, a message is sent to the writer that it is ok to set the write token. This enforces strict consistency on the entire object. In the case of a **write()** operation, a message has to be sent to the Copy Set to invalidate the written page on all participating nodes. For sequential consistency, the page-invalidates don't have to occur prior to the **write()** operation taking place, but they must occur prior to any subsequent accesses to that page by the writing program on any other node, and prior to any propagation of the page by the writing Node. This is because any read access by the accessing thread subsequent to the write, must return the results of that write regardless of the node on which the read takes place. Any threads on other nodes accessing that memory location will get the old value until the invalidate operation is completed. This is a small glimpse into a larger potential problem covered in section 1 .

4 CONSISTENCY MANAGEMENT IN SOMBRERO

4.1 Data Structures

A complete depiction of Sombrero Data Structures is shown in [Skousen Thesis]. A discussion of the parts that pertain to Consistency Management follows. Additionally, those pertinent structures are shown in Appendix B.

4.1.1 Memory Object Control Block

To review, a Sombrero *Memory Object* (MO) is the basic Sombrero abstraction characterized by its *Range*, a distinct variable sized contiguous allocation of the virtual address space. An MO is named by the base virtual address of its Range. The *Memory Object Control Block* (MOCB) is a small data structure containing non-replicated, node-specific data which is needed to manage the memory object, locally, and across the network. The MOCBs for all MOs on a node are contained in a system MO which uses Isolated Consistency, enforced by MRMW Consistency and the Protection Domain (see section 2.2.4). This system MO is allocated on every node. An MOCB for a particular MO exists in this system MO on each node that is a member of the Copy Set. When an MOCB on a node is a member of the Copy Set, it means that there has been, at least, some access to the object on that node. The MOCB contains the Addresses of the Range, the Access Control List (PRAL) pointers, the address of the particular Consistency Service implementation used by this Memory Object, and other data needed by the system for management of the Memory Object.

Each Memory Object (MO) also contains one or more Regions for managing the MO by type of access. The Region has associated with it an access type set, which is a subset of {Read, Write, Execute}. An address within a region will not appear in another Region with the same access type. Regions are managed by using Region Control Blocks (RCBs). Each MOCB contains the number of RCBs corresponding to the number of

Regions. Each RCB contains reference to the addresses controlled by the Region. The list of addresses must fall within the Range of the Memory object but need not be contiguous. Each RCB also holds a Token. While Sombrero allows more than one Region per MO, the simulation in this thesis makes the assumption that, without loss of generality with respect to consistency, an entire memory object contains a single Region whose set of Virtual Addresses includes the entire Range. The purpose of RCBs is to enhance concurrency of access to a MO. Investigation of the use of RCBs for concurrency is beyond the scope of this thesis.

The Token is data that enables or disables access to the Region. The actual Token Data Structure is not finalized. The Tokens are set via method call by the Consistency Service, which implements a particular consistency policy. The Tokens, however, are owned by the Protection Service which implements the methods to alter the value of the token. These methods are *settoken(Access_Type), unsettoken(Access_Type)*, and *hasToken(Access_Type)*, where *Access_Type* is in {*Read, Write, Execute*}. These methods act only on data residing locally on the MO's associated MOCB, or "surrogate," as it is called in Sombrero. The Consistency Service is the "authorized agent" for calls to these methods.

Each Memory Object also contains a Token Graph Structure. The Token Graph Structure contains per-object management data including pointers for the Copy Set, Copy Set Parent, Modified Page Cache, and Last Known Writer Graphs.

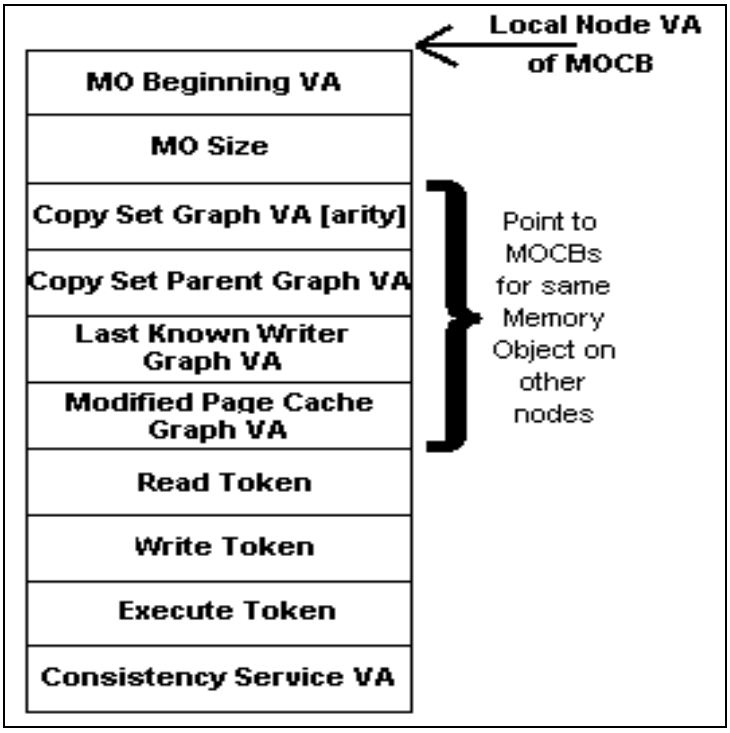


Figure 3. Simplified Memory Object Control Block

In the simulation contained in this thesis, all the data items needed from the single RCB and the Token Structure are flattened into a simplified, simulated, MOCB.

As discussed above, this assumes that no loss in generality with respect to consistency management occurs by using a Memory Object that consists of a single Region. This assumption is valid because access always will occur using one access-type at a time. And since a particular access type for an address never exists in more than one Region, resolution of access to a particular address is never ambiguous. A simplified MOCB, similar to the one used in this thesis' simulation, is shown in Figure 3. Refer to Appendix B for diagrams of the proposed real Sombrero structures.

4.1.2 Consistency Management Data

The Token Graph Structure within the MOCB contains 4 pointers to other MOCBs which reside on other nodes but represent the same Memory Object. These pointers are called the Last Known Writer, the Modified-Page Cache, Copy Set and Copy Set Parent Graph pointers. Sets of these pointers – when linked between an MO's MOCBs, one on each participating Node – comprise the series of graphs referred to in Section 3 and discussed further in Sections 4.1.3 and 4.1.4.

4.1.3 The Copy Set Graphs

The Copy Set is the set of nodes, each of which has a MOCB for a particular object. The Copy Set Graph is a distributed graph formed from the Copy Set Graph pointers of all MOCBs for a Memory Object (see Figure 4). The Copy Set Graph is needed by the Protection Service to keep track of all nodes participating in access to an object. The Copy Set Graph is also important to the Consistency Service when it needs to contact all participating nodes about the consistency status of an object. Thus, while the Copy Set Graph is maintained and protected by the Protection Service, it can be accessed by all, and may be needed by most Consistency Service implementations. Note that in Figure 4 and other figures depicting Sombrero inter-nodal graphs, "Name=VA(MO)" refers to the beginning Virtual Address of the Memory Object. That VA is the MO's name as noted in Section 3.2.

The Copy Set Parent Graph is a similarly-distributed graph, and the reverse of the Copy Set Graph. It allows reverse traversal of the copy set and is useful in sending acknowledgement to messages received from the Copy Set Graph. There is a single path from any node to the root of the Copy Set graph (See Figure 5).

4.1.3.1 The Copy Set Graph

The Copy Set Graph, as well as the other 3 graphs, is first created as a null graph, i.e. it is created as a single vertex representing the node on which the object was created. This node is called the Inception Node or Root Node. A pointer edge is added to the graph when a second node requests access to the object. The Protection Service on the requesting node contacts the Root Node of the Memory Object which sends information back to the requesting node to create an additional MOCB for this object, called a surrogate, locally on the requesting node. The Root Node then adds the address of this new requesting node to one of its Copy Set Graph fields in the MOCB (See Figure 3). The new surrogate MOCB Copy Set Parent Graph field is populated with the address of its parent, in this case the root node. New requesting nodes are added in the same way until all 'arity' Copy Set Graph fields are filled on the root node. After that, new nodes are added to the Copy Set Graph by passing the request to add the address to one of the Copy Set child nodes pointed to by the Copy Set Graph addresses. The request is passed along the existing graph until a free (value==NULL) Copy Set Graph pointer field is found. See [Figure 4]. The Copy Set Graph arity is a Sombrero initialization parameter. The default arity is 2.

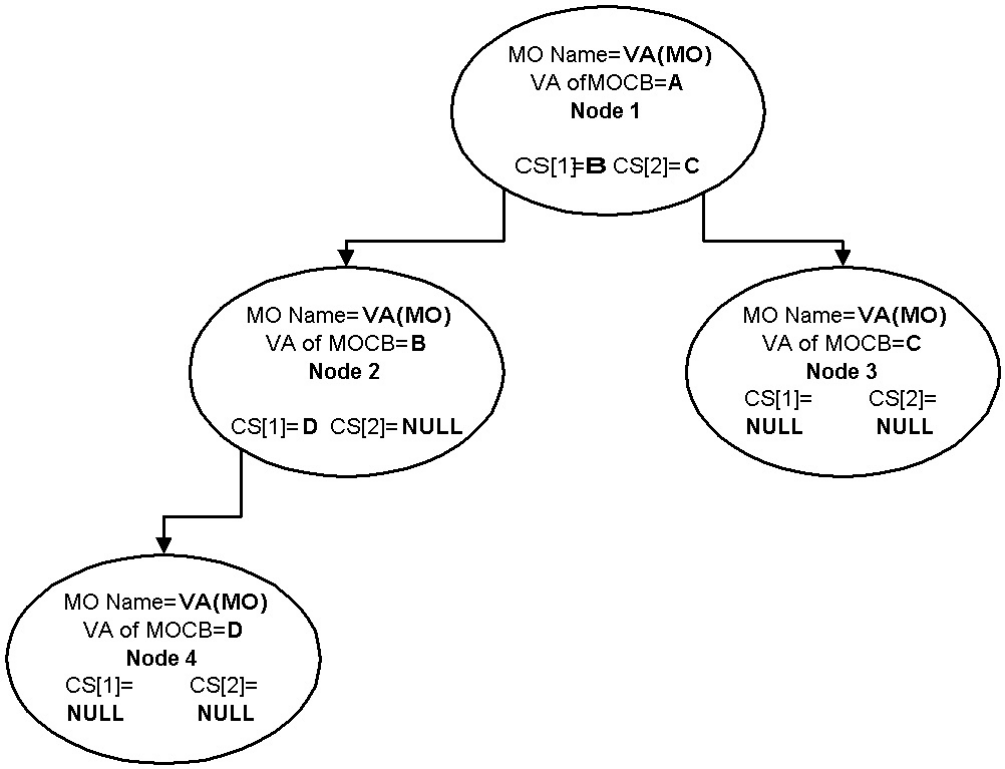


Figure 4. Copy Set Graph with Arity 2

In order for the Copy Set to be notified, the Root MOCB is contacted. By default, the name of the memory object is its Virtual Address which is within the Range of the MO's Root Node. This allows the Communications Service to route the message to the Root Node using the MO's VA. The Root MOCB, then, contacts all of its children on the Copy Set Graph. Each child node, in turn, contacts its children, until the ends of the graph are reached. When a leaf node, a node that has no Copy Set children, is reached, it immediately sends an acknowledgement to its Copy Set parent to notify it that the message was received. When a node has received acknowledgement from all of its Copy Set Graph children, it in turn sends an acknowledgement to its parent. Finally, when the Root Node receives acknowledgement from all of its children, a message is sent

to the requesting node that the Copy Set has been traversed. See Figure 4 and the corresponding Copy Set Parent Graph in Figure 5.

One advantage to the Copy Set Graph is that it avoids broadcasts, requiring communications only with nodes participating in the access of an object. Another advantage is that there is no central manager for the Copy Sets, which could become a bottleneck. Additionally, the Single Address Space precludes the need for address translation. The graph pointers can be passed directly to the Communications Service, and on to the proper peer service on another node. This is because the SombreroVA space is the same on every node and can be used by all programs, services, threads and data. It is also possible for the Communications Service to use the VAs for routing if the network drivers, hardware, and firmware are set up for it.

4.1.3.2 Copy Set Parent Graph

The Copy Set Parent Graph is a collection of pointers to MOCB copies, allowing reverse traversal of the Copy Set Graph. It is related to the Copy Set Graph in that there is a reciprocal relationship between parent and child nodes. It functions to optimize Copy Set Graph use, for example, for replies from nodes to messages sent from their parent or when nodes need to be removed from the Copy Set. The Copy Set Parent Graph has many starting points (nodes) and one terminal node (See Figure 5), which is the root of the Copy Set Graph. It could be considered as several individual linear graphs sharing some nodes (See Figure 6).

The Copy Set Parent Graph is useful in acknowledging requests for page invalidates sent to the copy set. The page invalidate algorithm sends a message to the Copy Set root node. The root node then sends a message to all of its children. This is repeated by each child node until the leaves of the graph are reached. Then, after page

invalidation, the leaves send acknowledgement messages to their parent. When the parent has accumulated all of the acknowledgements of its children, it sends its own acknowledgement to its parent. Eventually, the root node gets acknowledgements from all of its children and the original requesting node of a page invalidate is notified. One message is sent for each node. On a network of n nodes, 2n messages will be sent. Assuming parallelism in the network, the time to traverse the entire copy set for page invalidates is dramatically affected by the arity of the copy set graph. Parallelism here means that there is sufficient network bandwidth so that marshalling and other message preparation time is long compared to the transmission time. (If this is not the case, then parallel message preparation on multiple nodes won't have a benefit because the messages will be backlogged waiting for access to the network.)

The Copy Set Parent Graph is also useful when deleting an MOCB copy. The parent of the deleted node must be informed of the deletion, and the parent pointer precludes the need to query the whole Copy Set Graph to find the parent. If the deleted node has no children, very little additional work is needed. If the deleted MOCB copy has children, the Copy Set Graph must be modified so that the deleted MOCB's children remain included in it. The task of creating and deleting MOCB copies is orthogonal to consistency management and is the job of the Protection Service. It is mentioned here to give additional background to the reader. The simulation in this thesis only executes fixed Copy Set environments, configurable when the test run is started, with no run time Copy Set changes taking place.

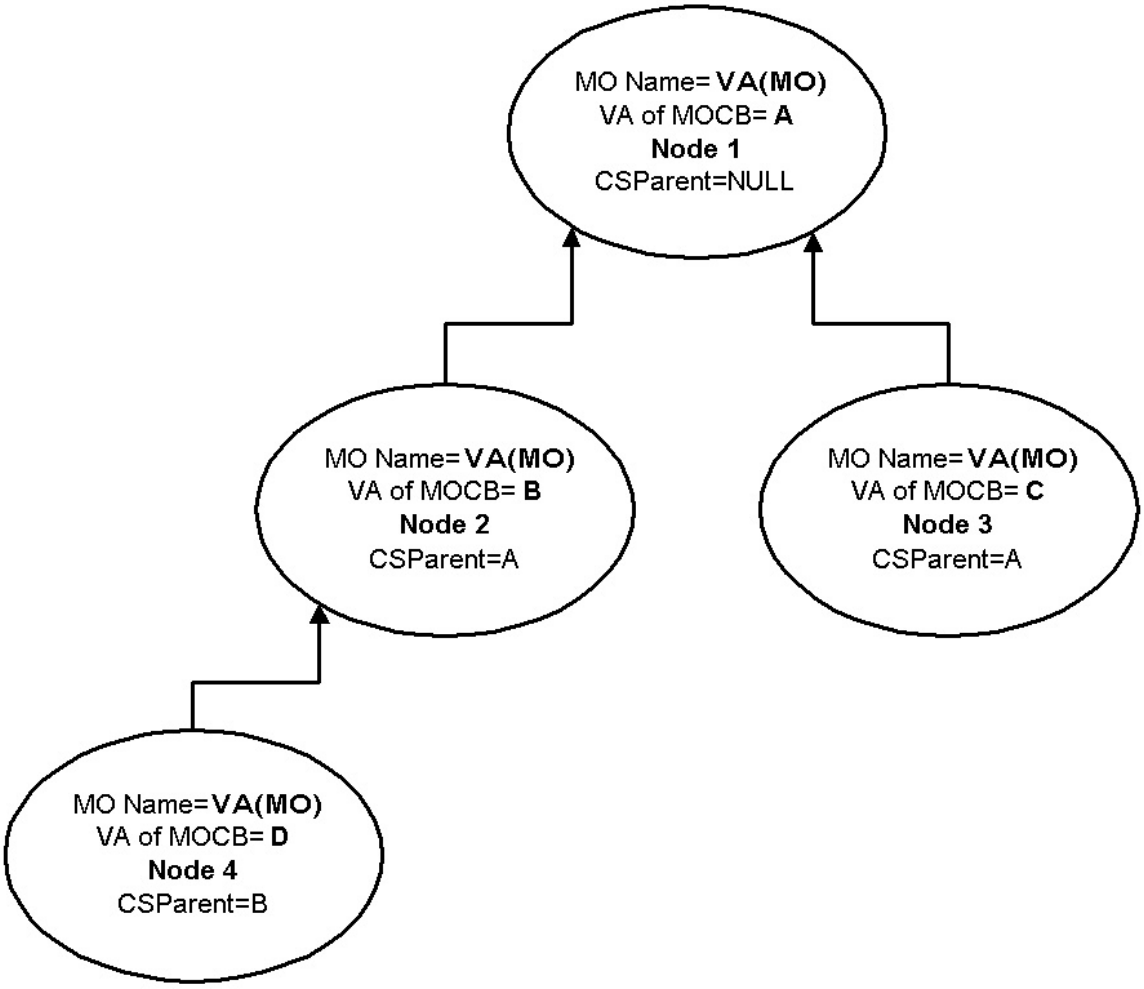


Figure 5. Copy Set Parent Graph

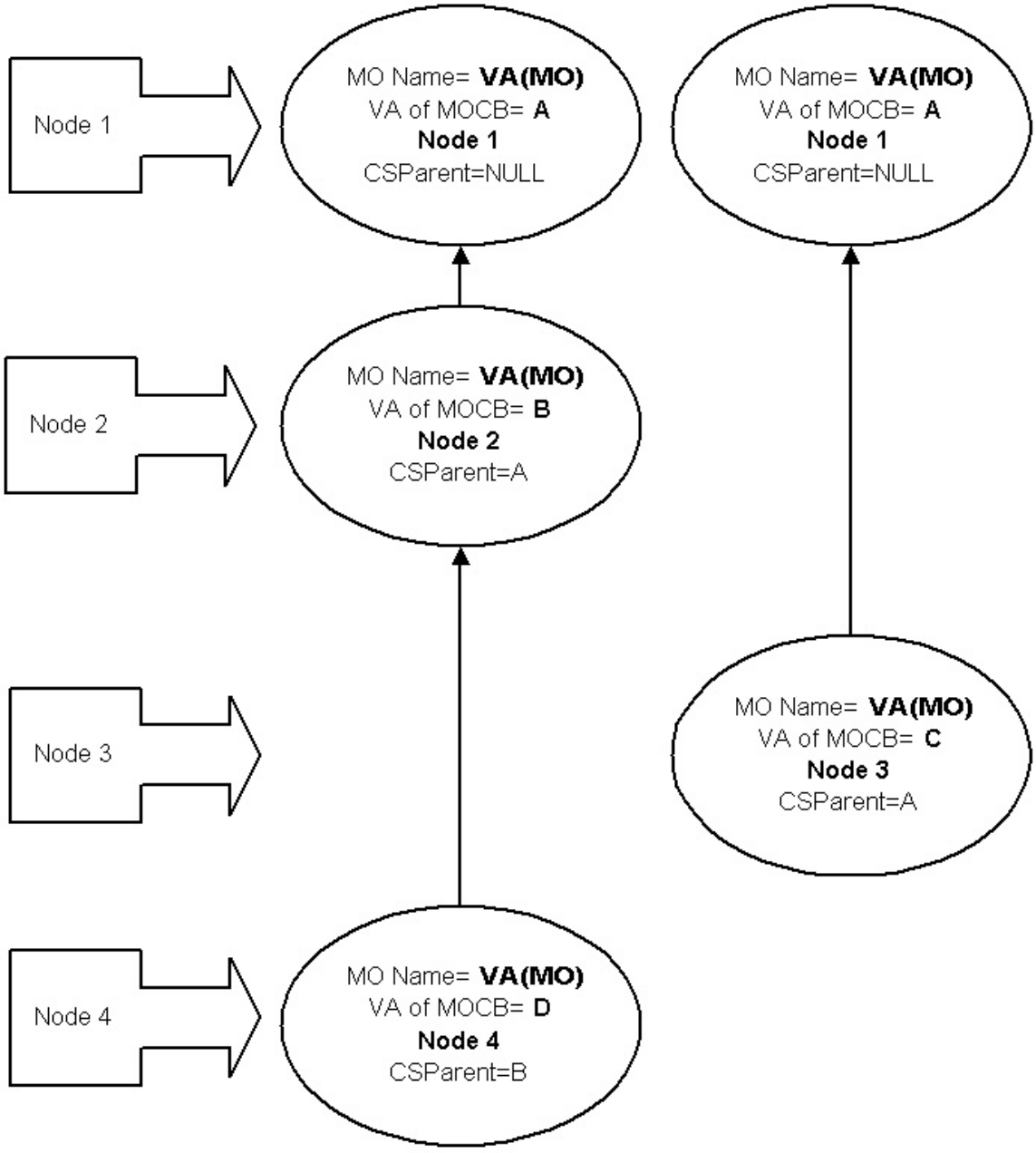


Figure 6. Copy Set Parent Graph Depicted As Linear Graphs

4.1.3.3 Graph Maintenance

The Copy Set Graph is entirely controlled and maintained by the Protection Service and is not within the scope of this thesis.

4.1.4 The Last-Known-Writer and Modified-Page-Cache Graphs

The Last Known Writer and Modified Page Cache Graphs are designed to be used with Sombrero's default Consistency Service implementation, "Token Tracking." They are described in the following sections in that context although they could be used by any Consistency Service implementation. Some examples of their use are included in the following sections and a more extensive set of examples is outlined in Section 4.2.

4.1.4.1 Last Known Writer Graph

The Last Known Writer (LKW) pointer in a node's MOCB for some object is the address of the MOCB copy to which it sent the token. The current writer has a NULL LKW Graph pointer. All the Last Known Writer pointers, linked together, form a distributed, inter-nodal graph called the Last Known Writer Graph [Figure 7].

When a memory object is first created, its MOCB contains a NULL LKW graph. When that Object is first written by a second node, the Last Known Writer Graph is extended. When the inception node relinquishes its write token and passes the authorization to set the Token on to the new writer, the address of the new writer is placed in the "Last Known Writer" field of the inception node's MOCB. If more nodes get write access, the graph is similarly extended. By following the graph, the most recent writer can be found by any past writer. All MOCBs for a MO, except the MOCB on the LKW node, must have an entry in their LKW field. Upon creation of a surrogate, the address of the Copy Set root node is entered if a member of the LKW Graph is not known. In this way, nodes which have never written are directed to the root node which always can find the LKW because the Root Node is always a member of the LKW Graph.

Consider a simple example system with 4 participating nodes. Each oval in Figure 7 represents an MOCB copy for one object, on one node. Node 1 is the node on which the memory object was created. Then, Node 2 wanted to write and obtained the write token. Subsequently, node 3 and node 4 made requests to write and received the write token, in that order. At this point, the Last Known Writer Graph is shown in Figure 7.

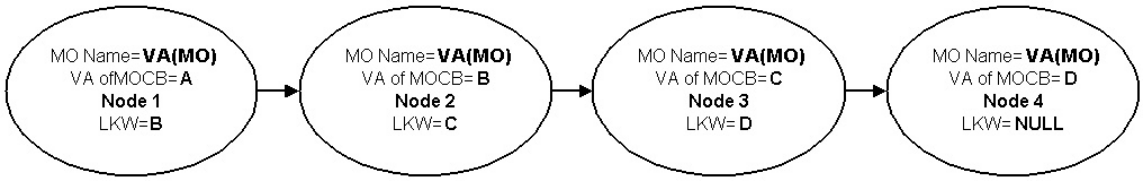


Figure 7. Last Known Writer Graph

Now, suppose that Node 2 again requests the write token. As in the previous example, the Last Known Writer pointer in the last writer's MOCB is set to the VA of the MOCB on the requesting node. In Figure 8 this is shown on Node 4, the last writer, with LKW equal to the MOCB address on Node 2, the requesting node.

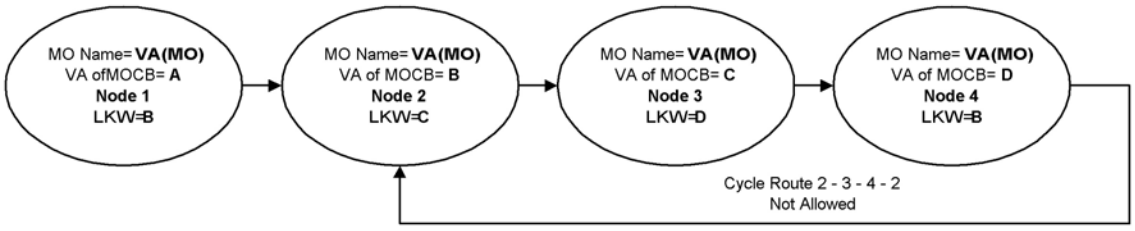


Figure 8. Extended Last Known Writer Graph

A cycle has been created. Node 2, upon receiving the write token, sets its Last Known Writer pointer to NULL. The resulting graph is shown in Figure 9. Note that this graph is directed and acyclic, but does have multiple paths. This automatic elimination of the cycle is called "pruning." Receivers of the write token will always set the Last Known Writer pointer to NULL, since there is no subsequent writer node when a MOCB copy holds the write token. The token holder is, by definition, the end of the Last Known Writer Graph. A NULL Last Known Writer pointer implies that this MOCB copy has authority to set the Write Token. Note that it is possible for a write token to be revoked without the token being acquired by another MOCB copy. Setting an object to be read-only is an example of such a situation. When the Consistency Service revokes the Token, it calls the Protection Service method *unsetToken(Write)*. Besides disabling the token, this method flushes the entry for the access type on this MO from the given node's RPLB. If the node doesn't relinquish the right to set the Token, no graph modifications occur.

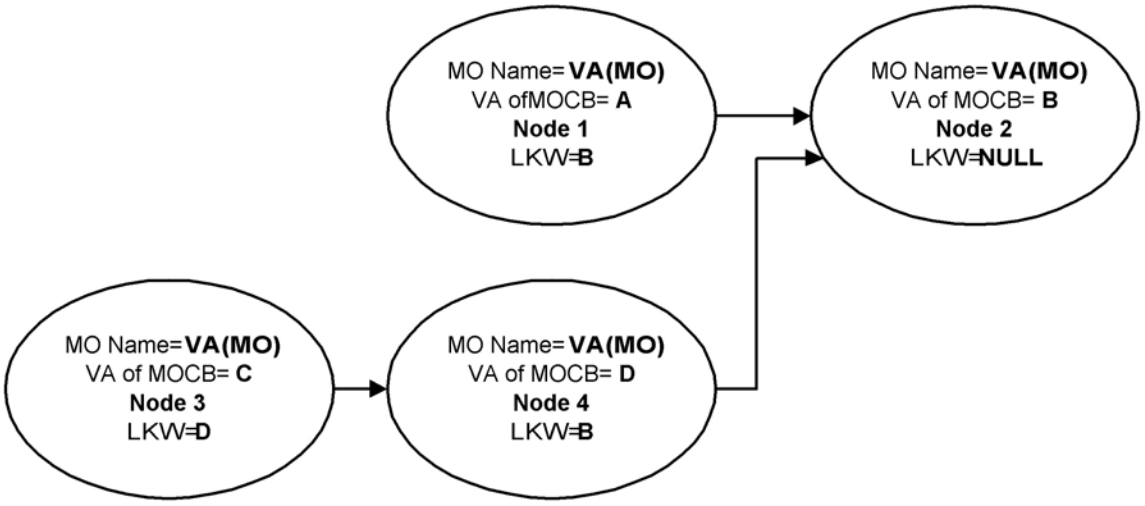


Figure 9. Pruned Last Known Writer Graph

As an additional illustration, Figure 10 shows that the graph can be extended by another node if Node 5 receives the write token. Node 2 becomes an interim stop for 2 different paths on a traversal to find the last writer.

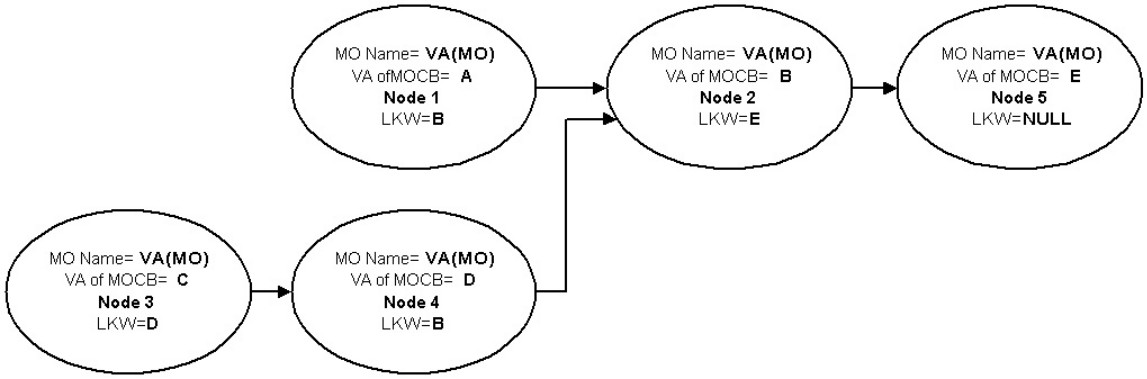


Figure 10. Extended Pruned Last Known Writer Graph

The Last Known Writer Graph is used when a write request occurs. The most recent writer must be found because that is the starting point for updating the LKW and MPC Graphs which may occur when the write enabling token is transferred.

4.1.4.2 Modified-Page Cache Graph

The Modified Page Cache Graph is comprised of the MPC Graph pointers of all the MOCBs of an MO that hold valid pages of the MO that may not be deleted. They may not be deleted because they may be the only copy of those particular pages. The nodes represented by the MPC Graph is the set of Nodes that, when taken together, contains all the valid pages of a MO. No other nodes are assumed to have any valid pages although they likely will. An MPC Graph is shown in Figure 11.

The LKW node is always the root of the Modified Page Cache (MPC) Graph and the MO's Root Node is always the tail of the MPC Graph. The Modified-Page Cache (MPC) Graph is a graph which will locate previously-written, up-to-date pages of a memory object. The root of the Modified-Page Cache Graph is always the LKW Node,

which is at the end of the Last Known Writer Graph, and the tail of the MPC Graph is always the MOs Root Node, which is the root of the Copy Set Graph.

When the write enabling token is transferred, the MPC Graph may be lengthened or shortened depending on whether the new writing node is currently on the MPC graph or not. If the new writing node is not on the MPC graph, the current LKW node sets its MPC Graph pointer to the address of the new writer. And the new LKW node sets its MPC graph pointer to the old LKW node address. The MPC Graph is lengthened by one node at the head.

If the new LKW is already on the MPC Graph, which is the case when the MPC Graph Pointer is not NULL or the new writer is the Root Node, pruning occurs and the graph is shortened. Modified pages along the deleted portion of the Modified-Page Cache Graph are sent directly to the new writer. See Figure 12. The new writer initiates pruning by notifying the old writer to begin the pruning operation. The MPC Graph is followed node by node until the new writer node is reached. Each node along the way initiates the page transfer and then notifies the next node on the graph to continue the operation.

Using the same example as for the Last Known Writer Graph, Figure 11 shows the Modified-Page Cache Graph at the same point in time as Figure 7. This graph snapshot happens to be an inverse of the LKW graph in Figure 7, but this will often not be the case.

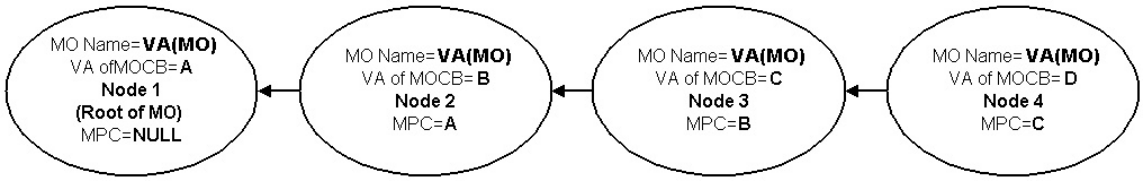


Figure 11. Modified-Page Cache Graph

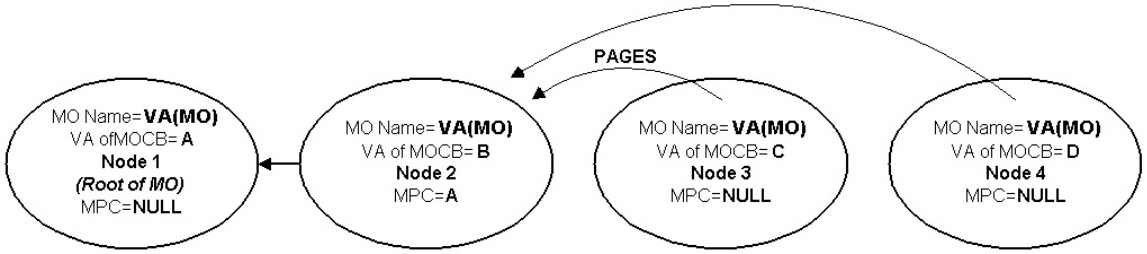


Figure 12. Process of Pruning Modified Page Cache Graph

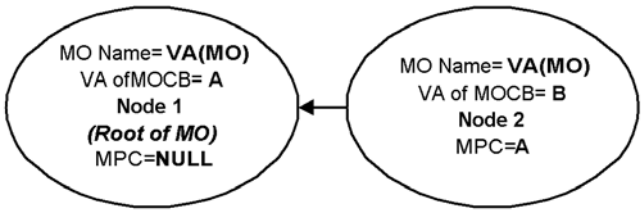


Figure 13. Pruned Modified Page Cache Graph

When Node 2 in is to receive the write token, all modified parts of the object on nodes 3 and 4 are sent to Node 2, and Nodes 3 and 4 are no longer on the graph. It is always the case that if both Nodes 3 and 4 wrote to a particular page of an MO, Node 2 would receive the results of the most recent write to that page from the node that did the most recent writing. The resulting graph is shown in Figure 13, which is a snapshot of the Modified Page Cache Graph at the same point in time as Figure 9. The flushed pages are not necessarily invalidated on the nodes which are pruned out of the graph. The graph's purpose is to be a sure way to find an up-to-date copy of a page, without relying on broadcasts or a central page manager. If the flushed pages remain valid, there is no reason to prohibit read access on the flushed nodes, since a page's existence on the modified page cache graph does not mean that particular copy is the only valid copy. (Note that the Modified Page Cache Graph always has a single path.) Page invalidates take place when a page is written, not when the write-enabling token is transferred.

The page flushing operation is accomplished with calls by the Consistency Service to the Paging Service to get page data for all pages which were written locally or were copied in as part of a previous graph pruning operation. Ideally, the Paging Service will have sufficient page state to control which pages are referenced by a particular instance of such a call. Insufficient page information can result in considerable network traffic from duplicate data transmission, and it would be the receiving instance of the Paging Service that would handle the duplicates. The Consistency Service is blind to the contents of the flushed pages, and is only responsible for making the correct calls at the correct time. When obtaining valid pages from the Pager via a call such as *"GetModifiedPages(MOName)"* it would be reasonable that the Pager didn't clear the "modified" flag at this point. The Consistency Service could be expected to make an additional call to the Pager such as *"ClearAllModifiedFlags(MOName)."*

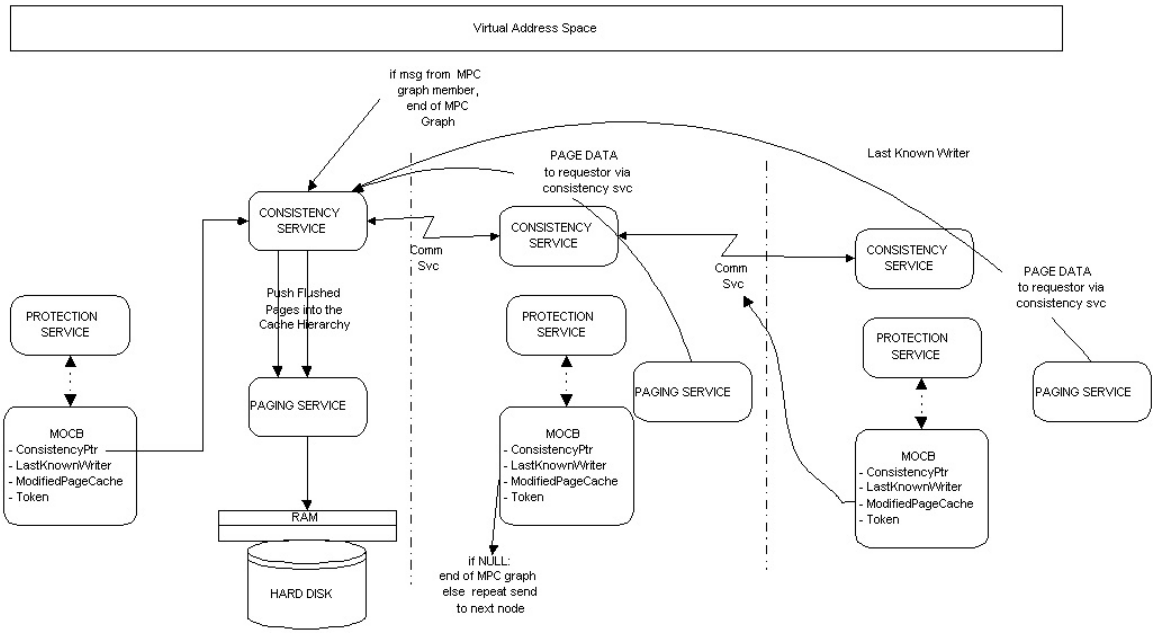


Figure 14: Page Flushing upon MPC Graph Pruning

In the case of the new LKW already existing on the MPC Graph, the Consistency Service on the current LKW node initiates a pruning of the MPC graph when it is about to relinquish writer status, an operation which flushes copies of all valid pages from nodes being pruned out of the graph to the original requesting node. Member Nodes of the MPC graph between the original requesting node and the tail of the MPC graph, the Copy Set Root Node, are not affected. The Consistency Service on the current LKW node contacts the Paging Service to fetch copies of all local valid pages associated with the MO. These pages are not invalidated as part of this operation. They are sent to the original requesting node where that node's Consistency Service takes them out of the communications buffer and passes them to the local Paging Service for placement in the cache hierarchy (See Figure 12 and Figure 14). The current Node sets its MPC Graph

pointer to NULL, removing itself from the MPC Graph, and the message to prune the MPC graph is forwarded to the next node on the MPC Graph.

The receiving node then repeats the process of fetching local pages and sending them to the original accessing node. It is possible that more than one node will have valid copies of the same page. Recall that the LKW and MPC Graphs are per-object and not per-page. In this case, in the absence of optimization, duplicates will be sent but they will contain the same data because all valid copies of a page are consistent by definition for the default case of sequential consistency considered here. Prior to sending the pages, the current node contacts the local Protection Service for the MPC Graph pointer.

When the MO's Consistency Service on the original requesting node gets notification that pruning is completed from the participating nodes, the Consistency Service calls back to the local Protection Service to do two things: 1) to set the LKW pointer in the local MOCB to NULL. 2) to set the enabling token in the local MOCB by calling *SetToken(Write)*. Note, that in the case of the original requesting node already existing on the MPC Graph, the value of the MPC graph pointer on the requesting node is not changed, and is not relevant to, this sequence of events. The remainder of the MPC graph remains intact. During the call to *SetToken()* the local Protection Service on the requesting node atomically loads the RPLB with an entry permitting the access which may then restart.

It is possible for an MOCB to have a token and still not have the needed page. The token gives permission to use any page of the object (VA range), but does not guarantee that a page is physically present on the node. As the token is passed from node to node, unmodified pages are not copied along with the token move. Only the pruning

of the MPC Graph as outlined above, or a page miss, will cause a page to be physically moved between nodes.

A likely scenario would be for a node to request the token, write to a few pages, and have the previous node re-request the token. Only those few pages that were written to will physically reside on that node. Those pages will reside on the current token-holding node, but not necessarily anywhere else. All this page-shifting occurs transparently, underneath Token Tracking, which is managing consistency at the object level.

When an MOCB copy is pruned from the Modified Page Cache Graph, its modified pages are copied to a different node on the graph. Therefore, a valid copy of a page can always be found by following the Last Known Writer Graph to the last writer and then the Modified-Page Cache Graph to the first instance of the page. There is no need to update pages unless a token is present or requested. Figure 15 shows the graphs from Figure 4, Figure 5, Figure 9, and Figure 13 in a single overlay view.

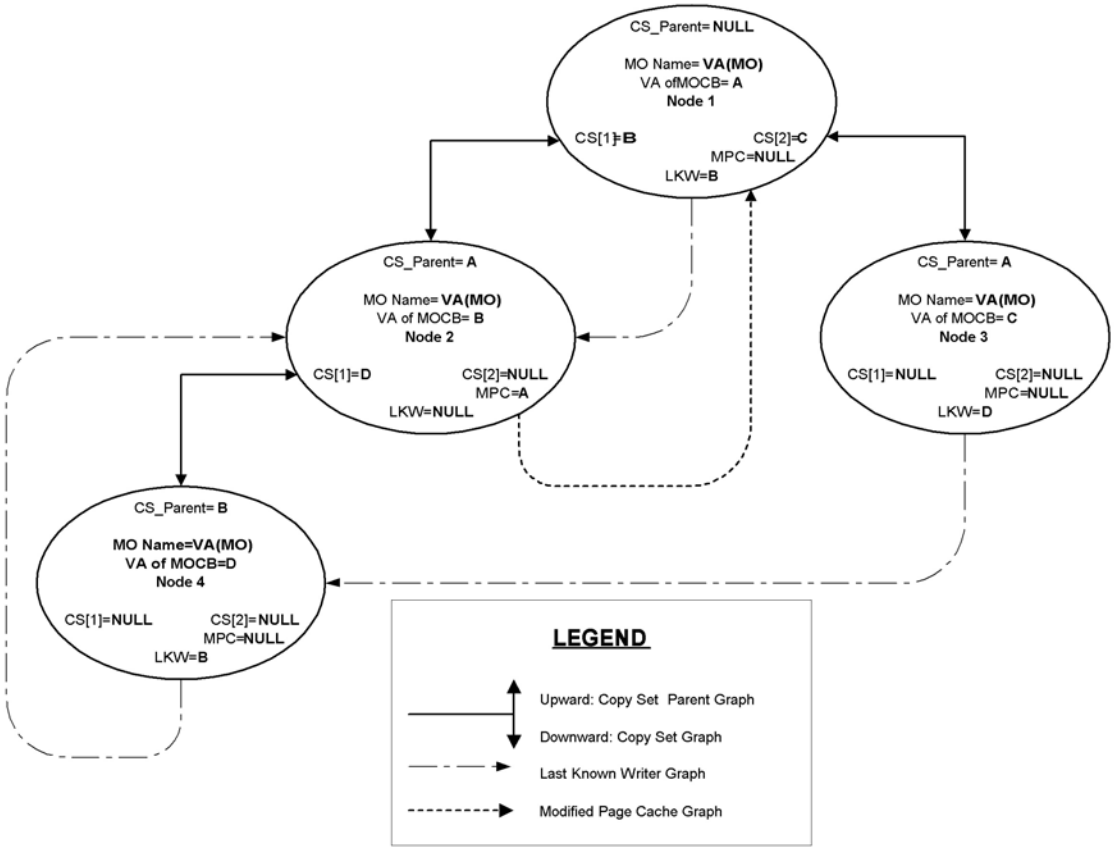


Figure 15. Overlay View of Four Graphs

4.1.4.3 Graph Maintenance

An MOCB can be added to the Modified-Page Cache or Last Known Writer Graph only when a page is modified. If pages are copied between nodes for read or execute purposes, or if a surrogate MOCB is created, any page of that object on that node is a replicate and exists, at least once, somewhere on the MPC Graph. However, if an MOCB is deleted, and happens to be on the Last Known Writer or Modified-Page Cache Graphs, special action must be taken. Note that the nodes which contain MOCB surrogates for an object make up the Copy Set in addition to the Root Node MOCB, and are not necessarily on either the Modified-Page Cache or Last Known Writer Graphs.

When deleting from the Last Known Writer Graph, any MOCB copies pointing to the deleted MOCB copy are sent a message to point to the deleted MOCB copy's Last Known Writer Graph successor. Figure 16, shows the Last Known Writer Graph from Figure 10, after deleting the MOCB copy from Node 4.

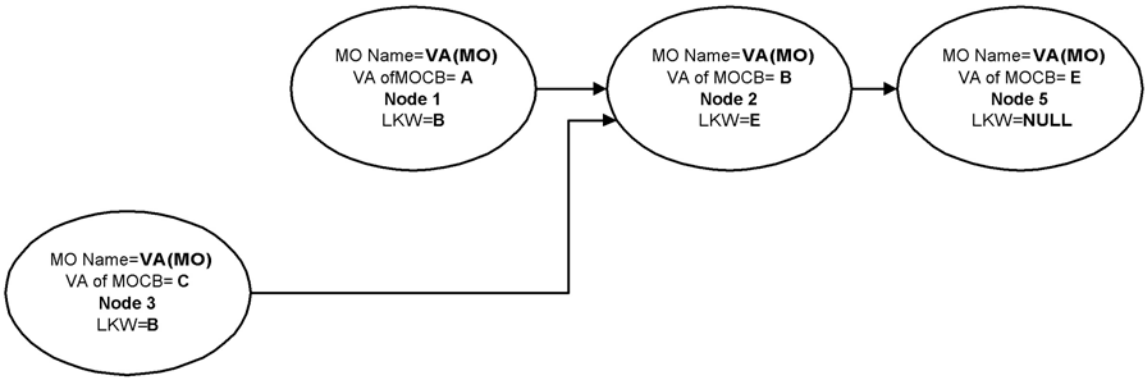


Figure 16. Last Known Writer Graph of Figure 10 After Node Deletion

Figure 17 gives the algorithm for deleting a node from the LKW Graph. Note: The simulation described in Section 6 does not include node deletion.

```
//=====================================================================
// Only occurs as part of a node deletion, shown here for illustration
//=====================================================================
delete_from_last_known_writer_graph()
{
if (LKW_pointer is not Null)
{
  send_message_to_copy_set(self_pointer, LKW_Pointer)
}

//=====================================================================
// If this node points to the deleting node, change to point to
// node pointed to by deleting node.
//=====================================================================
receive_delete_from_last_known_writer_graph(sender_pointer, LKW_Pointer)
{
  if (Local_LKW_Addr==sender_Pointer)
    Local_LKW_Addr=LKW_Pointer;
}
```

Figure 17. Delete Node from Last Known Writer Graph Algorithm

Figure 18 shows the algorithm for removing a node from the Modified-Page Cache Graph. The pruning of the graph itself seems trivial, but any modified pages must be copied to a remaining node on the MPC Graph, and the pointers to the deleted MOCB must be changed to the Modified-Page Cache Graph entry existing in the deleted node prior to deletion. The "after" graphic of the Modified-Page Cache Graph is almost trivial, being similar to removing a node from a linked list. It is the algorithm which is interesting.

```
//=====================================================================
// Only occurs as part of a node deletion, shown here for illustration
//=====================================================================
delete_from_modified_page_cache_graph()
{
  if (MPC_Pointer != NULL)
  {
    if (pagingSvc.modified_pages_exist_on_node(MO_Range))
    {
      pagingSvc.GetPageData(Page_Addrs,&Page_Data);
      send_pages_and_addresses_over_net(RootNodeMOCBAddr,
                                        Page_Addrs, &Page_Data);
    }
  }
  else if (Root Node)
    {take special action}
  send_message_to_copy_set(self_pointer, MPC_pointer); // always
 }
```

Figure 18. Delete Node From Modified-Page Cache Graph Algorithm

4.2 "Token Tracking"

Token Tracking in Sombrero is the design of the default Consistency Service which can be used to enforce sequential consistency semantics between copies of Memory Objects on different nodes. Token Tracking uses the four built-in graphs described earlier. The Consistency Service is one of the few programs that require the ability to be aware of the nodes on which the VA pages of a distributed object are present. Normally, replicated copies of data residing at the same Virtual Addresses are viewed as the same data by Sombrero Programs.

Exactly when a token is granted or revoked is a policy decision controlled by the Consistency Service. The default Token Tracking Service provided by Sombrero is a policy mechanism enforcing sequential consistency with Single Writer Multiple Reader semantics. Token Tracking is part of the Sombrero package, but does not require access to privileged instructions or Sombrero protected data sets.

The tokens are owned by the Protection Service, reside in the MOCB, and are accessible by method call to the Protection Service. The number of MOCB copies which may, simultaneously, hold a particular token is a matter of policy controlled by the Consistency Service for a particular Memory Object. Each MO has exactly one Consistency Service which, itself, is distributed and accessed by the same VA on every node. Token Tracking is the default Sombrero Consistency Service for applications.

Each node which participates in access to a Memory Object has an MOCB for the object stored locally in system-protected heap space. This MOCB may be on the inception node, or may be on another node, in which case that MOCB is called a surrogate. Note that each node has a small Range of Virtual Addresses within the Range of the node, where data is stored which must be protected to insure the integrity of the system. The MOCBs are stored here. Usually, it is not necessary to make a distinction between original and surrogate MOCBs; hence, both will be referred to as "MOCBs." The entire object does not have to be physically present on a particular node, but if any portion of the object physically resides on a node, an MOCB for the object must be placed on that node.

Calls to the Consistency Service can be made by the Protection Service or be made by the application needing the service by using the Consistency Service address stored in the MOCB. For example, a call to a system service to write an object will – in

the event of a missing token – call the entry point to the Consistency Service, which is registered in the MOCB. The Consistency Service, then, takes the necessary action to supply the needed page(s) to the requesting node and requests the Protection Service to set the token.

4.2.1 Design

Token-Tracking uses three system calls to manage the token. They are: [*SetToken(Access_Type)*], [*UnSetToken(Access_Type*)], and [*Boolean HasToken(Access_Type)],* where *Access_Type* is in the set {read, write, execute}. It uses the four directed acyclic graphs previously described, the Last Known Writer Graph, the Modified Page Cache Graph, the Copy Set Graph and the Copy Set Parent Graph. The graphs consist of pointers (virtual addresses) to MOCB copies. The Last Known Writer pointer in an MOCB points to the MOCB on the node which received the write token from this node. The Modified Page pointer points to the MOCB on the node which held the write token prior to this one. Each MOCB is unique in the system. It contains information relevant to the local node and to its relationship with other nodes (See Figure 3) and, therefore, must not be replicated.

Consistency semantics other than sequential can be implemented. In addition, Sequential Semantics can be implemented, in several ways. As a result, Token-Tracking is implemented as a peer service. It is not a protected system service and can be replaced by other implementations, as previously discussed. See Figure 19 to see a representation of some of the services inside the Sombrero Executive. The only interface requirement for the Consistency Service is that the well-known interface and resulting callbacks, as seen by the Protection Service, be the same for all implementations. Every Consistency Service must supply a set of methods, e.g. *get_access(Access_Type)*. After processing,

the Consistency Service will call back to the Protection Service using *unsetToken(Access_Type)* and *setToken(Access_Type)*. What the consistency policy manager does between the call it receives and its callback is a matter of consistency policy. In the simplest case, a MRMW consistency manager is a matter of unconditional call back to the Protection Service immediately with *setToken(Access_Type)*.

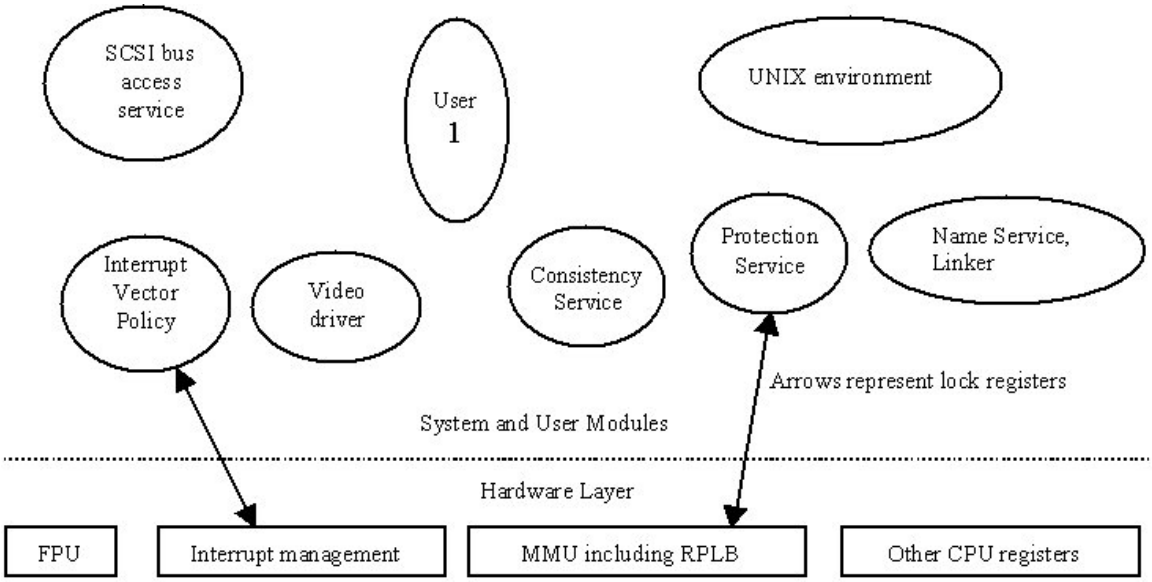


Figure 19. Current Sombrero Architecture on a Single Node

4.2.2 Scenarios

The "Token Tracking" Consistency Service is invoked in different ways depending on the conditions of the object. This section outlines the various scenarios that, in general, will invoke the registered Consistency Service, and cases relating to the behavior of "Token Tracking" specifically.

There are two general categories of invocations of "Token Tracking." One category is invocation when trying to obtain the enabling token. The second is invocation when an enabled node tries to access a specific address within an MO's range. If the page containing a virtual address exists locally, the Consistency Service is not

invoked. If a page miss can not be serviced locally, the Paging Service invokes the Consistency Service. Figure 20 shows the various ways that the "Token Tracking" Consistency Service implementation can be invoked. The italicized algorithm names are pseudo-function names that are used as "keys" in the database sense. Start in the "Execution or Event/Algorithm" field and then follow the italicized names from the "Action /Consistency Service Algorithm" column to the same italicized name in the "Execution or Event/Algorithm" column to follow an execution.

Figure 20. Token Tracking Invocation Scenarios

	Execution or Event/Algorithm	Invoking Service or Agent	Conditions	Action / Consistency Service Algorithm	Token Tracking Specific
1	**(A)** Attempt to Read, Write or Execute (this node is the "requesting node") / RequestAccess (e.g =, *read( ) write( )*, *run( )*)	Protection Service	Token Present, protection data not in RPLB (RPLB Miss)	Load Protection Info into RPLB / no action by Consistency Service (shown here for clarity)	N
			Token Not Present	Request Token / **[2]** *RequestToken* (local call)	
2	Obtain Token/ *RequestToken*	Protection Service	Requesting Node is already on LKW Graph	Contact Next Node On LKW Graph/ **[3]** *RequestTokenFromPeer* (network call)	Y
			Requesting Node is not already on LKW Graph (note: an MO's Root Node is always on the LKW Graph)	Contact Root Node / **[3]** *RequestTokenFromPeer* (network call)	

Figure 20. Token Tracking Invocation Scenarios (continued)

3	Consistency Service Receives Request for Token from Peer / *RequestTokenFromPeer*	Consistency Service	Node is the Last Known Writer (LKW) for the MO.	1) revoke MO's enabling token locally (Call Protection Service) 2) Set local LKW pointer to VA of requesting node MOCB (Call Protection Service) 3) Send Message to requesting Node / **[4]** *NotifyGrantTokenOK* (network call)	Y
			Node is not the LKW	Contact Next Node On LKW Graph/ **[3]** *RequestTokenFromPeer* (network call)	
			Requesting Node is already on MPC Graph. An MO's Root Node is always on the MPC Graph as the tail node. (MPC Graph Pointer not NULL) or (node is MO's Root Node)	1) Set local LKW Graph Pointer to NULL (prunes if not previously NULL) 2) Initiate Prune of MPC Graph by calling last LKW (sender of *NotifyGrantTokenOK*) / **[5]** *PruneMPCGraph* (network call)	
			Requesting Node is not on MPC Graph	1) Set LKW Graph Pointer to NULL 2) Set MO's enabling token locally (Call Protection Service)	

Figure 20. Token Tracking Invocation Scenarios (continued)

5	MPC Graph Member Receives Request to Prune MPC Graph / *PruneMPCGraph*	Consistency Service	Receiver is not Requesting Node	1) Get all pages stored locally from local Paging Service 2) Send Pages to Requesting Node / **[6]***SendPageData* 3) Send Message to Prune to Next Node on MPC Graph / **[5]** *PruneMPCGraph* 4) Set local MPC Graph Pointer to NULL (removes node from MPC graph)	Y
			Receiver is Requesting Node	no action (pruning is complete) See *SendPageData* case 1 for setting of the enabling token.	
6	Requesting Node Receives Pages from MPC Graph members that are pruning themselves out of the MPC Graph / *SendPageData*	Consistency Service	Pages Received from all sending Nodes	1) Retrieve Page Data from Network Buffers and send to Paging Service 2) Set MO's enabling token locally	Y
			Waiting for page data from more nodes	Copy Page Data from Network Buffers and send to Paging Service	
			Page is available locally	Paging Service obtains page locally and pages into memory or cache. Note: Consistency Service is not invoked.	
			Page is not available locally	Request Page from Consistency Service / **[8]***RequestPage* (local call)	

Figure 20. Token Tracking Invocation Scenarios (continued)

8	Find-a-Page-on-network Request from Paging Service / *RequestPage*	Paging Service	Requesting Node is already on LKW Graph		Contact Next Node On LKW Graph/ **[9]***FindLKWForPageRequest* (local call)	Y
			Requesting Node is not already on LKW Graph (note: an MO's Root Node is always on the LKW Graph)		Contact Root Node / **[9]***FindLKWForPageRequest* (network call)	
9	Consistency Service Receives Request for LKW search for page retrieval from Peer / *FindLKWForPageRequest*	Consistency Service	Receiving Node is Last Known Writer (LKW)	Page does not exist locally	Try to locate page on MPC Graph / **[10]***FindPageOnMPCGraph* (network call)	Y
				Page exists locally	Fetch page from paging service and send to requesting node	
			Receiving Node is not LKW		Call next node on LKW Graph/ **[9]***FindLKWForPageRequest* (network call)	

Figure 20. Token Tracking Invocation Scenarios (continued)

10	MPC graph member receives request for a page / *FindPageOnMPCGraph*	Consistency Service	Page exists locally		Send Pages to requesting node / **[11]***SendPageData2* (network call)	Y
			Page not available locally	Receiving Node is MO's Root Node	Page has never been written. Send zero-page to Requesting Node / **[11]***SendPageData2*	
				Receiving Node is not MO's Root Node	Try to locate page on MPC Graph / **[10]***FindPageOnMPCGraph* (network call)	
11	Requesting Node Receives Pages from MPC Graph member that has requested page / *SendPageData2*	Consistency Service	N/A		Copy Page Data from Network Buffers and send to Paging Service	Y

Figure 20. Token Tracking Invocation Scenarios

4.2.3 Algorithms

This section includes details of the graph implementations. The graphs are interdependent and an algorithm may need to access more than one graph. The consistency management graphs are accessed only if there is a valid attempt to access memory and the required token is absent, or if the required page is not available locally. In the case of a failed access attempt, a protection miss, the Protection Service, first, checks that the requesting thread has permission to access the object. Once authorization is established, the token is checked. If the access attempt fails authorization, the Consistency Service is never invoked. If the request is to write, and the token is

```
//=====================================================================
// Initiate call to find LKW and to flush modified pages to this
// requesting node
//=====================================================================
request_write_permission(MO_Name) /* call into Consistency Service */
{
  if (LKW_pointer != NULL)
  {
    send_request_for_write_token_to_peer(LKW_Addr,Self_Addr);
    /* blocks here until Last Known Writer graph is pruned */

    do
    {
      if (wait_for_reply(timeout))  /* wait for reply */
        recv_modified_pages_from_peer( MPC_Addr, Requestor_Addr,
                                          Page_Addrs, Page_Data);
      else
        return (error);
    }
    while (MPCGraphNotPruned); // not trivial, 2 cases
  }
  /* now at end of Last Known Writer Graph */
  set_token(Write); /* MOCB primitive */
  set_LKW_pointer(NULL);  /* Prunes the LKW graph */
  return "Write Token Has Been Set";
}
```

Figure 21. Request Write Permission Algorithm

not present, a call is made to the Consistency Service to obtain the write token. The algorithm for that request is as shown in Figure 21.

```
//=================================================================
// Get any valid pages from the paging service and send them to the
// requesting node.  Note:  no invalidating takes place
// send_modified_pages_to_requesting_node() is received by
// recv_modified_pages_from_peer()
//=================================================================
send_modified_pages_to_requesting_node(MPC_Addr, Requestor_Addr,
                                       Page_Addrs, &Page_Data)
{
  if ( pagingSvc.modified_pages_exist_on_node(Page_Addrs) )
  {
    pagingSvc.GetPageData(Page_Addrs, &Page_Data);
    send_pages_and_addresses_over_net(Requestor_Addr, Page_Addrs,
                                      &Page_Data);
  }
  MPC_Addr = NULL; /* always prune */
}
```

Figure 22. Send Modified Pages to Peer Algorithm

4.2.4 Message Handlers

In order to implement requests, the instances of the Consistency Service must communicate with other instances of themselves on other nodes. The communications cause invocation of message handlers following an interrupt from network hardware. Since Sombrero supports passive servers, there is no need for a thread to wait continuously for a message to arrive. The thread can migrate to the necessary node and perform the receive operation. Figure 23 shows the message handling operations performed by a node when it receives a request for the write token from a peer and Figure 24 shows the message handling when the requesting node receives pages from a node on the MPC Graph.

```
//=====================================================================
// search for the LKW, and if found, begin flushing pages back
//   to the new writer.
// forward the request to the next node on the graph
//=====================================================================
recv_request_for_write_token_from_peer(Requestor_Addr)
{
  if (LKW_pointer == NULL)
  {
    /* at end of Last Known Writer Graph (at last writer) */
    LKW_pointer = Requestor_Addr;
    /* begin pruning the MPC graph */
    send_modified_pages_to_requesting_node(MPC_Addr, Requestor_Addr,
                                           Page_Addrs, Page_Data);
    forward_request_for_page_flushing_to_peer(MPC_Addr,Self_Addr);
    MPC_pointer = NULL;
    unset_write_token();
  }
  else
  {
    /* continue searching for the LKW */
    send_request_for_write_token_to_peer(LKW_Addr,Requestor_Addr)
  }
}
```

Figure 23. Receive Request for Write Token from Peer Algorithm

```
//=====================================================================
// write pages back to local cache hierarchy
//=====================================================================
recv_modified_pages_from_peer(Requestor_Addr, Page_Addrs, Page_Data)
{
  {
    if (Page_Addrs != NULL)
      pagingSvc.update_cache_hierarchy(Page_Addrs, Page_Data);  }
  }
}
```

Figure 24. Receive Modified Pages from Peer Algorithm

Figure 16 shows the message handling when a member of the MPC graph receives a request to prune itself from the MPC Graph.

```
//=====================================================================
//  Handle Local Pruning and forward the request
//=====================================================================
receive_request_for_page_flushing_from_peer(MPC_Addr,Sender_Addr)
{
  if (MPC_Addr==Self_Addr)
  {
    /* pruning is done */
    setToken(Write);
  }
  else
  {
    send_modified_pages_to_requesting_node(MPC_Addr, Requestor_Addr,
                                           Page_Addrs, &Page_Data);
    forward_request_for_page_flushing_to_peer(MPC_Addr,Self_Addr);
    MPC_Addr=NULL;
  }
}
```

Figure 25. Handle Local MPC Graph Pruning

5 COMPARISON OF DSM AND SOMBRERO

Sombrero's support for distributed Consistency Management is scalable. Only the information that is needed locally is stored locally and the Consistency Service makes communications calls directly to the affected nodes not relying on broadcasts. Virtual Addresses can be used by the Communications Service for routing.

5.1 Consistency Policy Supported Per-Object

Sombrero supports Consistency Policy on a per-object basis. An object has a registered Consistency Server which itself replicates as the managed MO replicates. When the MO replicates, MOCBs are created on each node where access is requested and the address of the distributed Consistency Service is recorded in that MOCB. The Consistency Service is implemented as a Protection Domain with access to at least one MO. The Range of the MO contains the VA of the entry point to the Consistency Service. Like all MOs in Sombrero, this entry point is the same Virtual Address throughout the network. The Virtual Address of the entry point to a Consistency Service is in the consistency service field of every MOCB for an MO that used that particular Consistency Service and is the same because the VA is valid network-wide. When the Consistency Service is accessed, if its MOs don't already exist locally, an MOCB is created for it and the needed pages brought to the local node. This replication is handled automatically by the system. No special handling is needed by the System to distribute the Consistency Manager during distribution of an MO.

Since all execution shares the one address space in Sombrero, and, since that address space is distributed, all memory in Sombrero is intrinsically Distributed Shared Memory (DSM). Traditional DSM typically uses a distributed namespace to translate between distinct underlying VA namespaces on the various nodes. The usual VA-to-Physical address translations occur between these distinct VA namespaces and their

underlying physical memory and swap files on the various nodes of a DSM system. A translation between the local VA and the DSM distributed namespace is needed for every inter-nodal message passing, RPC, and page shuttling occurrence. The DSM VA must be again translated to the VA on the receiving node. Software, often dubbed "middleware" loaded on each node performs this additional pair of translations. In Sombrero, there is no such "middleware" needed to do page shuttling, namespace translation and message passing. The Sombrero system inherently supports the uniqueness of the single distributed shared VA namespace. Additionally, the application program developer need not have knowledge of either individual pages or the network. Sombrero, like all systems containing physically distributed memory, must pass low-level messages across the buses that connect these memories.

Each memory object in Sombrero may be mapped to any desired consistency manager. The consistency semantics policy maintained by a consistency manager can be defined by the user and tailored to the application. This is because system services in Sombrero are peer services and do not, in general, need to have access to privileged instructions (See Figure 19). The Protection Service inherent in Sombrero, which does have access to privileged instructions, protects other memory objects from poorly-written or rogue services.

5.2 Copy Set Managed by System

The Copy Set is automatically managed by Sombrero. There is no separate centralized "clearing house" for namespace management. Any address names the same location throughout a Sombrero network. This means that the Virtual Address Space is distributed along with all objects in a Copy Set, and the organization, management, and protection of each Copy Set are managed jointly and transparently by the local Protection

Services. Similarly, the Paging Service is built into Sombrero and no additional paging or communications services, such as those commonly found in current software-DSM middleware, are needed to implement distributed shared memory.

5.3 False Sharing Problem Exists in Sombrero and Conventional DSM

False sharing is a situation that arises when a memory location, uniquely used by one node, is in the same coherency unit as another memory location, uniquely used by another node [Tanenbaum DOS p336]. Although there is no conflict when both nodes write their respective addresses, the coherency unit becomes invalidated upon each write; and it generates unneeded Memory Management System (MMS) and network processing, to keep distributed memory consistent. A coherency unit is the chunk of memory used by the MMS for consistency. The larger the coherency unit is, the higher the probability of a false sharing problem. Multiprocessors use memory blocks or cache lines. DSM systems use pages or groups of pages. Sombrero's consistency is maintained at the object level, which will at times aggravate the False Sharing problem since there is only one Write Token. Mitigating this is the fact that the data of entire Sombrero Memory Objects is not invalidated upon every write, only the permission to write, and the particular pages written are invalidated. Subject to consistency policy, readers of unmodified pages may continue to have access. Nonetheless, when there is only one Write Token, False Sharing among writers of a Memory Object may be severe. With properly designed hardware, Sombrero can allocate very small Memory Objects to alleviate the False Sharing problem but many Memory Objects will likely be large. Large Memory Objects with many concurrent Write-Access attempts from different nodes could result in thrashing as the write token is shuttled back and forth even if the access is to different parts of the MO's range. Dynamically sized coherency granularity to reduce False Sharing is supported in

Sombrero by Regions and Region Control Blocks. Separate Write Tokens are provided for each Region Control Block. Recall that Regions in Sombrero is non-overlapping parts of a Memory Object. Performance degradation due to false sharing is beyond the scope of this thesis.

6 IMPLEMENTATION OF SIMULATION

The purpose of this simulation is to increase the understanding of Sombrero by modeling parts of the Sombrero Operating System design. The hope is that the simulation can cast some insight into correctness, performance and scalability that can't be done by static analysis alone.

6.1 General Design

The simulation consists of 1) a set of processes each of which is an instance of a Win32 Console Application with each instance representing a simulated node, 2) a Visual Basic Windows Application, called the UI Application, for generating user input to the various simulated nodes, and 3) a Win32 Console Application called the "Host" application which routes messages between simulated nodes and logs activity. The number of simulated nodes will vary so the console is suppressed for that application. Each simulated node has a Windows Socket which accepts input from the Visual Basic Application and a Windows Socket that accepts input form the Host application. See Figure 26.

Possible input received by each simulated node from the UI Application include requests to read or write a Virtual Address. Each simulated node assumes all access attempts are legal from a Protection Service standpoint. The simulated Consistency Service is consulted and if inter-nodal communications are necessary, the simulated node sends a message on its other socket to the Host Application. The Host Application then logs the request, and routes the message to the appropriate node based on the contents of the message. The Host Application maintains a table of Node name (VA) to Socket Handle pairs. There is a message protocol defined between the UI Application and the Simulated Node Applications as well as another protocol defined between the Host Application and the Simulated Node Applications.

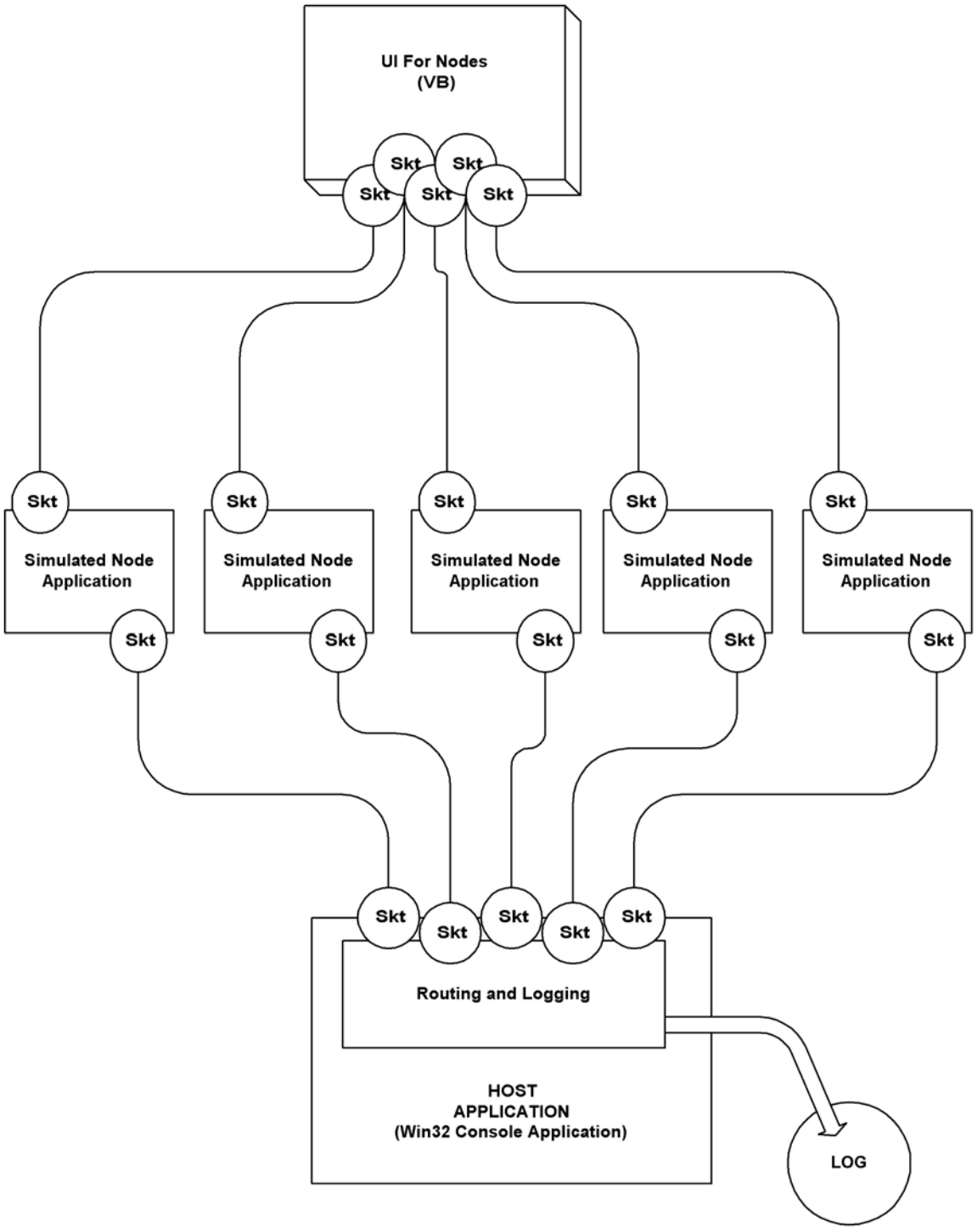


Figure 26. Sombrero Simulation Implementation Architecture

Sombrero consists of a single virtual address space distributed over many nodes. Each virtual address, in any system, translates to a physical page at runtime. In a distributed system, the virtual address being addressed is translated at runtime to a physical address on the local node that is carrying out the access. The simulated node application above is a single Windows 2000 process is used to simulate a single node (see Figure 27). Windows 2000 isolates each process's virtual address space from each other, translating the VA of each into a physical address at runtime. When processes are switched between during a context switch, VA translations are changed, which, slightly simplifying, prohibits any process from accessing the actual data that is used by another process. A parallel can be drawn between this behavior and the isolation between physical nodes on a distributed system. Any node does not have access to the physical data residing on another node. The view of the virtual address space in each Windows 2000 process is used to simulate a Sombrero node's view of the single virtual address space. Each simulated node has its own translation of VAs to the local physical store, the same as it would be for a real Sombrero node. The Windows 2000 paging system and context switch functionality simulate, per Windows 2000 process, the per-node VA management facilities on a Sombrero node.

The entire memory range of a memory object on each simulated node is created at the same virtual address, using Windows C++ library calls. The entire range of VAs for the simulated Memory Object is, in Windows 2000 terminology, reserved and committed, in each process, each representing a simulated node. The Windows pager and VM system is left to manage the whole range on each simulated node normally as single process with no consideration for the distributed model and distributed paging. Page valid state on each simulated node is maintained in a separate data structure, locally, in

each Windows 2000 process. Old data will be kept by Windows 2000 as it has no knowledge of the simulated Sombrero state of the page. The separate data structure determines whether the simulation considers the page valid or invalid.

The Communications Service is simulated using Windows sockets. Each simulated node establishes communications with a well-known host application. The host application maintains a table of socket handle-to-virtual address relationships and functions as a router, when a simulated node sends a message to another node. See Figure 27 for a side-by-side comparison of Sombrero functionality and the Windows 2000 simulation of it. The Windows 2000 host application is substituting for other parts of the Sombrero System not shown in the Sombrero part of the figure. In addition, it performs a logging function to demonstrate correctness.

While normal operation of a real Sombrero system will find nodes dynamically added and deleted from the Copy Set, the Consistency Service is applied, in real Sombrero, to a snapshot of the Copy Set at any given point in time. For this reason, the simulation operates on a fixed set of nodes, set at simulation program startup, for a given test case, which represents such a snapshot. The management of the Copy Set is performed by the protection service. It is not a function of the Consistency Service and is out of the scope of this research.

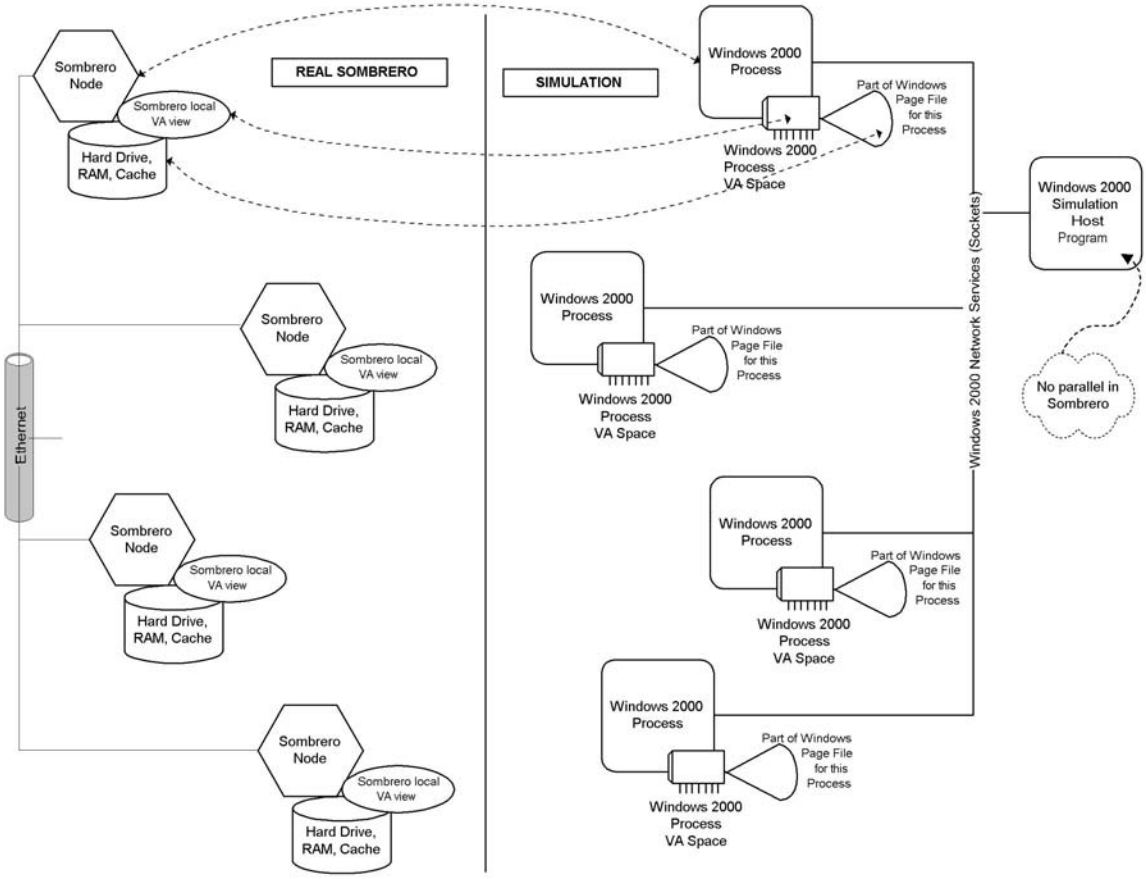


Figure 27. Sombrero to Simulation Mapping

6.2 Modeling Consistency

The host application (see Figure 27) is used to record all activities in the order they are processed. The host application is single-threaded.

6.2.1 Sequential Consistency

The simulation contains an abstract class, CConsistency, which defines the interface for consistency management in Sombrero and can be instantiated with a child class of various semantics. CSequential inherits its interface from CConsistency and implements Sequential Consistency.

6.2.2 Weaker Consistency

Implementing a weaker consistency model is as simple as writing a subclass of CConsistency and instantiating that as the CConsistency * that enforces consistency for a particular simulated Memory Object.

6.3 Differences from Real Sombrero

Refer to Figures B1, B2 and B3 in Appendix B and Figure 3. Simplified Memory Object Control Block. This simulation uses a simplified control block structure. The fields not used by the Consistency Service are absent in the simplified, simulated MOCB. The region control block is not used. An assumption is made that there is no loss of generality with respect to consistency by mapping the entire memory object into a single Region. The RCB in real Sombrero enables non-overlapping separate regions for a particular access type. Executable code is an example. This allows a set of Protection Domains to have *execute* rights to a code portion without any consistency management overhead, while allowing reads and writes at different addresses within the object, or on different nodes, simultaneously as long as the access was legal. Additionally, Regions also help mitigate the False Sharing problem. As mentioned, performance optimizations based on RCB attribute mapping are beyond the scope of this research. The simplified MOCB for this simulation contains all the fields needed by a Token-Tracking consistency manager that works on a Memory Object that has a single Read-Write-Execute Region covering the entire Range of the Memory Object.

In Sombrero, the consistency policy managers are passive servers. When the policy manager is invoked, the thread of execution moves to the policy manager and migrates throughout the network as needed during consistency policy management and related support. When consistency policy management completes and succeeds, the

enabling token is set which enables the access. The thread of execution then returns to the original memory-access attempt in the calling program on the originating node. In Windows 2000, there is no concept of a passive server. In the simulation, the thread accessing the memory in question is blocked using Win32 calls, while the policy is enforced.

7 RESULTS

7.1 Overview

This research investigates whether different consistency models can be managed correctly in Sombrero. The simulation models the functional operation of Sombrero's consistency management design. It shows that it is correct and feasible, that it works for more than one consistency policy implementation, and investigates some of its dynamic behavior. The simulation developed as part of this research correctly ran test cases for Sequential Consistency and MRMW consistency semantics. Threads may migrate in Sombrero which affects consistency. While out of the scope of the simulation contained in this thesis, Chapter 1 discusses consistency under thread migration.

7.2 Sequential

Tests were run using the simulation under various scenarios of Sombrero's Token Tracking consistency policy. The experimentation shows that in all test cases, all nodes saw the expected results, seeing the writes in the same order.

The test cases ranged from a single node to intermingled reads and writes from several nodes. The simulation programs did not mature to the level that enabled the volume of testing where network latency would be significant to the test. All tests accessed one byte.

MO Range:0x40102000 – 40104000 Page Size: 0x1000			Page Size: 0x2000 Node Size: 0x100000		
Node Addr	Node	accessed address	operation	value	notes
0x40100000	1	0x40102000	write	'a'	
0x40100000	1	0x40102000	read	'a'	
0x40100000	1	0x40102000	write	'b'	
0x40100000	1	0x40102000	read	'b'	

Figure 28. Single Node Access

The Single Node Access depicted in figure Figure 28 is almost trivial, but necessary to demonstrate that the most basic read and write functionality works.

MO Range:0x40102000 – 40104000 Page Size: 0x1000			Page Size: 0x2000 Node Size: 0x100000		
Node Addr	Node	accessed address	operation	value	notes
0x40100000	1	0x40102000	write	‘a’	
0x40100000	1	0x40102000	read	‘a’	
0x40200000	2	0x40102000	read	‘a’	page migrated
0x40100000	1	0x40102000	read	‘a’	page still valid on node 1
0x40200000	2	0x40102000	write	‘b’	page invalidated on node 1
0x40200000	2	0x40102000	read	‘b’	
0x40100000	1	0x40102000	read	‘b’	page updated on node 1
0x40100000	1	0x40102000	write	‘c’	page invalidated on node2
0x40200000	2	0x40102000	read	‘c’	page updated on node 2

Figure 29. Two Node Access

Figure 29 shows a simple case of two nodes accessing the same page of the same memory object. When node 2 read the page, the data migrated to the new node, 2. When node 1 read the memory location, no network traffic was generated because the read token was still present. The test showed that token tracking correctly invalidated and shuttled pages when access was requested. In Sombrero, the Copy Set Graph, managed by the local Protection Services, is used to notify nodes of page invalidates. In the simulation, the Host Application sends the messages to all nodes to notify of page invalidates. In Sombrero, 2n messages, where n is the number of nodes in the Copy Set, are always needed to notify and acknowledge page invalidates to the entire Copy Set. Every node receives one notify message and sends one acknowledgement message with its parent, except the root node who communicates with the requesting node. All write

operations that trigger a page invalidate will generate 2n network messages in addition to any others needed to complete the operation. Page invalidates triggered in Figure 29, Figure 30, and Figure 31 do generate theses 2n messages. These three figures and Figure 28 make the assumption that node 1 already has the write token and no page invalidates are triggered from the first write operation listed.

MO Range:0x40102000 – 40104000 Page Size: 0x2000 Page Size: 0x1000 Node Size: 0x100000					
Node Addr	Node	accessed address	operation	value	notes
0x40100000	1	0x40102000	write	‘a’	
0x40100000	1	0x40102000	read	‘a’	
0x40200000	2	0x40102000	read	‘a’	page migrated
0x40300000	3	0x40102000	read	‘a’	page migrated
0x40200000	2	0x40102000	write	‘b’	page invalidated on node 1 & 3
0x40200000	2	0x40102000	read	‘b’	
0x40100000	1	0x40102000	read	‘b’	page updated on node 1
0x40300000	3	0x40102000	read	‘b’	page updated on node 3
0x40100000	1	0x40102000	write	‘c’	page invalidated on node2 & 3
0x40200000	2	0x40102000	read	‘c’	page updated on node 2

Figure 30. Three Node Scenario

MO Range:0x40102000 – 40104000 Page Size: 0x1000				Page Size: 0x2000 Node Size: 0x100000			
Node Addr	Node	accessed address	operation	value	page	# msgs	notes
0x40100000	1	0x40102000	write	‘a’	1	0	root node
0x40200000	2	0x40103000	write	‘x’	2	4	MO is distributed, pages not replicated
0x40200000	2	0x40102000	read	‘a’	1	2	page 1 migrated
0x40200000	2	0x40103000	read	‘x’	2	0	existing local data
0x40300000	3	0x40102000	write	‘b’	1	10 (2n incl)	page 1 migrated to node 3, invalidated on nodes 1 & 2
0x40200000	2	0x40102000	read	‘b’	1	2	page 1 migrated
0x40100000	1	0x40103000	write	‘y’	2	11 (2n incl)	page 2 migrated to node 1, invalidated on node2, non-existent on node 3
0x40200000	2	0x40103000	read	‘y’	2	2	page 2 updated on node 2
0x40300000	3	0x40103000	read	‘y’	2	2	page 2 updated on node 3
0x40100000	1	0x40102000	write	‘c’	1	6 (2n)	page invalidated on node2 & 3 (already has token)
0x40200000	2	0x40102000	read	‘c’	1	2	page updated on node 2

Figure 31. Three Node Multiple Pages

The “# msgs” column in Figure 31 shows the number of network messages generated by each operation including the 2n messages generated in completing the page invalidate operation. This helps illustrate that the size of the copy set will affect scalability. But in a large network with sparsely distributed objects, less network interruption will occur when compared to a broadcast. The number of messages invoked vary significantly with the state of the graphs. There is clear advantage being near the

root node which is the root of the Modified Page Cache Graph. Further development of this simulation would allow tests that could expose more dynamic behavior data from scenarios with more nodes and more operations on larger Memory Objects.

7.3 MRMW

The simulation always allowed reads and writes to the affected memory and never attempted to distribute or retrieve the data across the network. The subclass for the CConsistency class has no communications code so it was a virtual impossibility that this would fail in the simulation.

MO Range:0x40102000 – 40104000 Page Size: 0x1000				Page Size: 0x2000 Node Size: 0x100000			
Node Addr	Node	accessed address	operation	value	page	# msgs	notes
0x40100000	1	0x40102000	write	‘a’	1	0	
0x40200000	2	0x40103000	write	‘x’	2	0	
0x40200000	2	0x40102000	read	‘\0’	1	0	all pages are “zero page” by default
0x40200000	2	0x40103000	read	‘x’	2	0	
0x40300000	3	0x40103000	write	‘c’	2	0	
0x40200000	2	0x40103000	read	‘x’	2	0	
0x40100000	1	0x40103000	write	‘y’	2	0	
0x40200000	2	0x40103000	read	‘x’	2	0	
0x40300000	3	0x40103000	read	‘c’	2	0	
0x40300000	3	0x40102000	write	‘b’	1	0	
0x40100000	1	0x40102000	write	‘c’	1	0	
0x40200000	2	0x40102000	read	‘\0’	1	0	
0x40200000	2	0x40102000	write	‘z’	1	0	
0x40300000	3	0x40102000	read	‘b’	1	0	
0x40200000	2	0x40102000	read	‘z’	1	0	
0x40100000	1	0x40102000	read	‘c’	1	0	

Figure 32. MRMW Three Node Scenario

8 CONSISTENCY UNDER THREAD MIGRATION

Although the scope of the simulation contained in this thesis does not include effects on consistency caused by the behavior of migrating threads, analysis of the simulation exposed a number of issues relating to thread migration. These issues are discussed here in hopes that they will be of some use in the Sombrero prototype. Sequential Consistency requires that each node in a distributed shared memory system see a sequence of memory accesses in the same order[4]. For example, a program writes a "zero" to a memory location, and then a "one." Once some reading processor reads "one" from that location, it must not read "zero" on any subsequent reads to that location unless and until it is written to again. Also, wihin an individual processor, strict consistency applies. In a node-centric, process-centric DSM system, this is the expected result. In Sombrero, the properties of a Distributed Single Address Space require some changes to the conventional node-centric and local-location-centric definitions and understandings of things like execution, process, processor and thread. In particular, the question of what constitutes the equivalent of an "individual processor" needs to be defined.

8.1 Description of Sombrero and Conventional Systems

Token Tracking, Sombrero's implementation of Sequential Consistency, only allows one writer at a time on an entire Region of a Memory Object, and it is guaranteed that all writers to an MO will be on the same node. Writers from different nodes cannot write to a given Virtual Address simultaneously. Page invalidates occur prior to a write token transfer, so on any node, a series of read accesses to a page will see values in the

[4] The text of the definition of Sequential Consistency from Lamport in 1979 as shown in [Tanenbaum DOS 318] is "The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

same order, first the old value, then the new value. Once the page is invalidated, subsequent access will fetch the updated page from the node that most recently wrote to it. Threads executing on a writing node will never see an old value because the page exists locally and will never be fetched from a non-updated node. Sequential Consistency holds in that different processes may see different values on a read operation as long as they see the effect of writes in the same order.

In a conventional system, each node has its own processes and threads operating on distributed data. Execution takes place in a process' Virtual Address Space. In that sense, access is "node aware." Transfers of executions to other nodes for load balancing, RPC, availability enhancement or other purposes are usually explicit and/or the data is marshaled along with the call. In Sombrero, however, threads of execution may, transparent to the programmer, migrate between nodes. Since all items in Sombrero, executable code, threads and data, share the same Virtual Address Space, there is no inherent node-aware or node-centric property associated with an execution. When a thread migrates in Sombrero, it continues to work in the same context on a different node. It is not like RPC where the thread on the calling client gets suspended and a thread on the called server takes over working on the marshaled data that is transmitted between nodes.

8.2 Description of Problem

If an MO or a Region has both Write and Read attributes, there is both a Write and a Read Token associated with it. If we make the assumption that a Read Token and a Write Token can be enabled on the same MO or Region simultaneously on different nodes, then enforcing consistency has some challenges. It is entirely possible that a thread could migrate from the writing node after having read the new value, then read the

same Virtual Address on the destination node and get the old value unless some mechanism to prevent it is implemented. Assuming that a Sombrero Protection Domain or a Sombrero Thread meets Lamport's definition of "processor" that he wrote in 1979, this would violate Sequential Consistency rules (see footnote 4 on page 71). The Virtual Address Space in a Sombrero Network is ubiquitous. An address accessed on one node means the same when accessed on another node. Threads are named by their Virtual Address and there is no inherent binding of a thread to a node, CPU, or other physical entity and more importantly to an instantiation of a General Protection Domain or the instantiated Memory Objects within it. Threads and data in Sombrero are Location Independent[5].

Here are some questions that arise with the discussion of Lamport's definition of Sequential Consistency. Does a Sombrero Protection Domain constitute a "processor?" Do all the threads executing in a General Protection Domain constitute a "program?" Or is a program the executable code in an MO? If a General Protection Domain is equivalent to Lamport's "processor," then the problem may be aggravated because threads can exist anywhere in a Sombrero network executing in the same General Protection Domain.

8.3 Scenarios

There are several scenarios that could lead to consistency problems when threads migrate. Additionally, a Memory Object can be included in more than one General Protection Domain (GPD), and can be accessed by more than one thread whether those

[5] [Tanenbaum DOS 251] Tanenbaum's description of Location Independence, "a system in which files can be moved without their names changing," is directed at files but can be applied to Sombrero entities. Sombrero goes a step further in that Sombrero entities not only can be moved to a different location keeping the same name, but also can be simultaneously present in multiple places in the nodes of the network with the same names (VAs).

threads are executing in the same or different GPDs or are on the same or different nodes. Figure 33 shows various scenarios of access to the same address in a Memory Object in which the returned data is not sequentially consistent. The figure assumes that all of the scenarios follow program order. The initial value of the memory location is 0.

Scenario Number	Thread Name Access Type {R,W}(value)			Node Name			PD Name PD Type: ({G,C}) where: G=GPD, C=CPD			Comments
	First Thread	Second Thread	Third Thread	First Node	Second Node	Third Node	First Protection Domain	Second Protection Domain	Third Protection Domain	
1	A W(1)	A R(0)	--	J	K	-	X (G)	X (G)	--	Thread A writes on Node J, migrates to Node K, and reads the old value
2	A W(1)	A R(0)	A R(1)	J	K	L	X (G)	X (G)	X (G)	Thread A writes on Node J, migrates to Node K, reads the old value, then migrates to node L where the invalidate has happened and reads the new value.
3	A W(1)	B R(1)	B R(0)	J	J	K	X (G)	X (G)	X (G)	Thread A writes on Node J. Thread B reads the new value, migrates to node K and reads the old value
4	A W(1)	B R(0)	--	J	K	--	X (G)	X (G)	--	Threads A and B are in the same Protection Domain. A writes on Node J. B reads the old value on node K.
5	A W(1)	B R(1)	C R(0)	J	J	K	X (G)	Y (G)	Y (G)	B and C are in the same Protection Domain. A is in another. B reads A's write and C subsequently reads the old value.
6	A W(1)	A R(1)	A R(0)	J	J	K	X (C)	X (C)	X (C)	Thread A writes to a private MO within its Carrier Protection Domain. It reads the result of the write, then migrates to node K and reads the old value.

Figure 33. Thread Migration Scenarios on a Memory Object

8.4 Solutions

Stepping back, the key point is that what Lamport was referreing to as an 'individual processor' are the entities participating in a single computation. In Sombrero, this is all of the threads of execution and Memory Objects they access to perform the computation, whatever nodes on which this occurs, or Protection Domains in which this occurs. The existence of passive servers, servers that can be animated by a migrating thread, means that the separation of 'individual processors' provided in the client-server paradigm in process-oriented systems no longer applies in Sombrero. Both the client and server must be considered part of an 'individual processor' and access to any common Memory Object must be strictly consistent. Page invalidates, read and write token revocation, page transfers, and read and write token enabling must be managed to effect this.

One likely correct solution to this problem is to use synchronization mechanisms to make sure the migration of threads after a write does not occur until the completion of the page-invalidate operation. This approach has the potential of performance degradation and deadlock and needs to be investigated further. Some variations on this idea would be to define a consistency policy on a Protection Domain scope so only those threads executing in the General Protection Domain of the writer would be affected. This scenario would guarantee Sequential Consistency on all MOs in a particular General Protection Domain (GPD). There would be no guarantee of Sequential Consistency of the value read from memory by the same thread executing in other GPDs.

Note that the issues of multiple threads attempting to access a memory location on the same node could constitute a race condition, which is no different than contention for a memory location on any system. Since all VAs are shared by all legal accesses, it is

possible that the need for synchronization mechanisms will increase under Sombrero. The use of conventional synchronization mechanisms may be a solution for assuring consistent access across the network because the VA of a synchronization variable would be valid systemwide and its strict consistency is assured by the system.

There are some instances that require sequential consistency with implication of strict consistency mentioned above. While being potentially slow and costly, Sombrero can support this model for cases, such as distributed synchronization, where reading an out-of-date synchronization variable value is unacceptable. If all read tokens are withdrawn prior to the granting of a write token, writes take place as needed, and page invalidates are propagated for the written pages. The granting of read tokens is delayed until all page invalidates are complete.

Another possible solution, essentially "Lazy Token Management," is for the transfer message to carry with it an invalidate list which could enhance the performance of the invalidate operation. This would occur when the write token is transferred, whether accompanied by a read token revocation or not.

Finally, it should be noted that there are a large number of situations in which sequential consistency is neither required nor desired. A good example is *writes* to a Memory Object in a loop inside a critical section protected by a semaphore [Tanenbaum 325]. Nobody else cares about the intermediate values. This is a principal reason why weaker forms of consistency such as entry and release were invented.

9 CONCLUSION AND VALUE OF RESEARCH

9.1 Direct Observations from Simulation

9.1.1 Passive Servers

Lack of Windows support for Sombrero-style support for passive servers and thread migration across memory objects complicated the simulation. In order for consistency to be maintained in the Windows 2000 simulation, memory access attempts that had to wait for the Consistency Service had to block using Win32 synchronization objects until the Consistency Service had finished its work and unblocked the accessing thread. In Sombrero, threads implementing passive services can migrate around the network performing Consistency Service tasks until the job is done and then return and resume the stalled read, write, or execute.

9.1.2 Synchronization

Currently, operating systems provide synchronization mechanism primitives such as mutexes and semaphores to protect data that is being accessed by more than one thread or process on a machine. When the same data is to be accessed by different threads or processes on different machines, the effort is compounded. Typically, another level of locking mechanism would need to be deployed, and this locking mechanism would need the aforementioned primitives in its own implementation. While synchronization is still necessary in Sombrero, the same virtual address locking mechanism can be used without regard to node location. Sombrero, as well as applications running on it, can take advantage of the single address space. Instances of synchronization variables have the same virtual address across the network and can be used when protecting any data whether distributed or not.

9.1.3 Language Support

The implementation of the simulation accented the differences in language support between Sombrero and a MASOS. To represent each memory object on each node, a Sombrero user need not do anything beyond present the Virtual Address to the system. In a MASOS, the memory object has to be declared on each simulated node using "new" and/or VirtualAlloc. The C++ "new" operator supports the specification of a Virtual Address where the new data item will be located[6] via the <new> standard header [Stroustrup p255]. Under Windows 2000 using Microsoft's Visual C++ compiler, this syntax is not supported requiring the developer additional effort to locate items at specific locations. Special consideration is required for initialization, construction and destruction. Even the incomplete functionality of this compiler and library would be no problem in Sombrero since objects don't have to be allocated or represented explicitly on several nodes when distributed. The Virtual Address is the name and is valid throughout a Sombrero Network.

9.2 Related Observations

9.2.1 Paradigm Shift

The simulation of Sombrero Consistency Services on a Windows platform demonstrates some of the advantages that are obtained in a single address space under the Sombrero SASOS. The simulation has an implementation of a Sombrero Communications Service interface. This implementation utilizes a central host application which maintains a map of sockets to node addresses. Each destination has a

[6] The normal syntax for the operator *new* is: `T* pT=new T;` The "placement syntax" allows allocation of the new item at a specified address A: `T* pt=new (A) T;` If successful, pt equals A after the call.

IP/Port number pair. Every communication requires translations from Virtual Address to socket and from socket to MAC address on the source, and back from MAC address to socket and to Virtual Address on the destination. The common use of these types of translations is taken for granted in networked computer systems today. The elimination of multiple namespaces as a means to protect and direct data in Sombrero is part of its use of the properties of a Single Virtual Address Space [Skousen 10th International Conference on Parallel and Distributed Computing Systems]. In the foregoing example, it is theoretically possible to assign a MAC address within the VA range of a hosting node to a network interface card (NIC). Some NIC manufacturers allow assignment of their 6-byte MAC address, which could make this a reality. If this were done, it would eliminate more namespace translations for communications on a Sombrero network.

This is an example of an implementation that is very straightforward in a single large multi-level distributed namespace and not in networked, process-oriented systems. There is a paradigm shift; or change in "world view" on what can be done and how to do it in a Single Virtual Address Space [Miller Using a single Address Space Operating System …].

9.2.2 Hardware Control

In a Single Address Space Operating System (SASOS), memory, threads, owners, users, and hardware all share the same virtual address space. Sombrero can give system-wide location independence to hardware, which would allow the movement of a physical device to another node without requiring reprogramming. Typically, machines consist of a group of sensors, motors, I/O modules, controllers, and computers tied together with a variety of bus and network architectures. Whenever a device on a machine is changed or moved – perhaps to better balance computer resources – a significant amount of

programming can be required. With a SASOS, hardware can be moved, and the controlling program can use the hardware at its new location with no programming changes whatsoever. The only possible change might be to invalidate the pages on the old node that allowed the hardware to be accessed at its old location. This behavior can be related to Plug and Play, but is better because the drivers would migrate in addition to the hardware recognition provided by Plug and Play.

9.3 Optimizations to Consider and Areas of Further Research

- If a read-only bit or flag were to be included in the MOCB, read tokens could be granted unconditionally, without the need to find and unset the write token. This would save the time needed to traverse the Last Known Writer Graph.
- Dynamic allocation and resizing of Regions.
- Investigation of the interaction between thread migration and the local page update/invalidate mechanisms.

REFERENCES

Bianchini, R., et al. "Hiding Communication Latency and Coherence Overhead in Software DSMs." Proceedings from the Seventh International Conf. on Architectural Support for Programming Languages and Operating Systems, 1996. Cambridge, MA. New York: ACM Press, 1996.

Chase, Jeffrey S. "An Operating System Structure for Wide-Address Architectures." PhD Dissertation, TR 95-08-06. U of Washington, 1995.

Heiser, G., K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. "Implementation and Performance of the Mungi Single-Address-Space Operating System." Software: Practice & Experience 28.9 (1998): 901-928.

Iftode, Liviu, Jaswinder Pal Singh, and Kai Li. "Scope Consistency: A Bridge between Release Consistency and Entry Consistency." Proceedings from the Eighth ACM Symposium on Parallel Algorithms and Architectures, June 1996. Padua, Italy. New York: ACM Press, 1996.

McLaughlin, Don. "Scheduling Fault-Tolerant, Parallel Computations in a Distributed Environment." PhD Dissertation. Arizona State University, 1997.

Miller, Donald S. "Using a single Address Space Operating System for Distributed Computing and High Performance." IEEE International Performance, Computing, and Communications Conference. Scottsdale, Arizona. February 1999.

Patil, Milind H. "Distributed Scheduling in Sombrero, A Single Address Space Distributed Operating System." M.S. Thesis. Arizona State University, 1999.

Skousen, Alan C. "Sombrero: A Very Large Single Address Space Distributed Operating System." M.S. Thesis. Arizona State University, 1994.

Skousen, Alan C., and Donald S. Miller. "Implementing a Very Large Single Address Space Across Multiple Nodes: Memory Partitioning, Protection Domain Migration, Kernel Replication, Consistency and Fault Tolerance." Computer Science and Engineering Dept., 64 Bit OS Group White Paper 10, TR-95-021. Arizona State University, Jan. 1996.

---. "The Sombrero Single Address Space Operating System Prototype A Testbed for Evaluating Distributed Persistent System Concepts and Implementation." Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, June 2000. Las Vegas. CSREA Press , 2000.

--- . "Operating System Structure and Processor Architecture for a Large Distributed Single Address Space." Proceedings of Tenth International Conference on Parallel and Distributed Computing Systems, Oct. 1998. Las Vegas. Calgary AB: ACTA Press Inc., 1998.

---. “Resource Access and Protection in the Sombrero Protection Model, Software Protection Data Structures and Hardware Range Protection Lookaside Buffer.” Computer Science and Engineering Dept., OS Group White Paper 2, TR-95-013. Arizona State University, Jan. 1996.

---. “The Sombrero Distributed Single Address Space Project.” Proceedings from the Second USENIX Windows NT Symposium, Aug. 1998. Seattle, WA. Berkeley, CA: USENIX, 1998.

---. “The Sombrero Operating System. An Operating System for a Distributed Single Very Large Address Space General Introduction.” Computer Science and Engineering Dept., OS Group White Paper 0, TR-96-005. Arizona State University, April 1996.

---. “Using a Distributed Single Address Space Operating System to Support Modern Cluster Computing.” Proceedings of 32nd Hawaii International Conference on System Sciences, Jan. 1999. Maui, HI. Honolulu: University of Hawaii: 1999.

---. “Using a Single Address Operating System to Support Client/Server Distributed Computing.” Computer Science and Engineering Dept., TR-97-019. Arizona State University, Dec. 1997.

---. “Using a Single Address Space Operating System for Distributed Computing and High Performance.” Proceedings from 18th IEEE International Performance, Computing, and Communications Conference, Feb. 1999.

Stroustrup, Bjarne. The C++ Programming Lanugage, Third Edition. Murray Hill, NJ:Addison-Wesley, 1997.

Tanenbaum, Andrew S. Distributed Operating Systems. Englewood Cliffs, NJ: Prentice Hall, 1995.

Zhous, Y., et al. “Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation.” Proceedings from the Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, June 1997. Las Vegas. New York: ACM Press, 1997.

ADDITIONAL READING

Fu, Mei Mei. "A Memory Mapped Persistent Object Oriented System with a Concurrent Programming and Execution Environment." Ph.D. Dissertation Arizona State University, 1994.

Tanenbaum, Andrew S., et al. "AMOEBA." Communications of the ACM 33.12 (1990).

Thapar, M. and B. Delagi. "Cache Coherence for Large Scale Shared Memory Multiprocessors." Proceedings from Second ACM Symposium on Parallel Algorithms and Architectures, May 1990. Island of Crete, Greece. New York : ACM Press, 1990.

Yeung, Donald, John Kubiatowicz, and Anant Agarwal. "Multigrain Shared Memory." ACM Transactions of Computer Systems 18.2. New York: ACM Press, May 2000.

APPENDICES

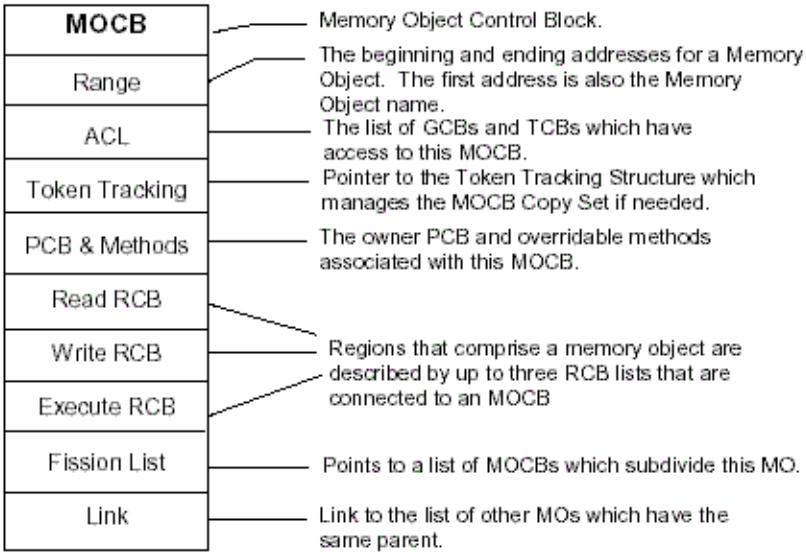
APPENDIX A

GLOSSARY AND LIST OF ABBREVIATIONS

Authorized Agent	A domain with the authority to execute some program.
callback	A scheme used in event-driven programs where the program registers a subroutine (a "callback handler") to handle a certain event. The program does not call the handler directly but when the event occurs, the run-time system calls the handler, usually passing it arguments to describe the event. Source: The Free On-line Dictionary of Computing, © 1993-2001 Denis Howe
Copy Set	The set of all MOCB copies for an object
Copy Set Graph	The logical link of pointers between MOCB copies for an object. Directed acyclic graph consisting of pointers connecting MOCB copies.
CREW	Concurrent Reader Exclusive Writer consistency semantics
CSP	Copy Set Parent. A pointer to the parent of a node on the Copy Set Graph.
DMA	Direct Memory Access. Access to memory that bypasses the usual processor-to-bus path
LKW	Last Known Writer
MOCB	Memory Object Control Block
MPC	Modified Page Cache
Node	Logical Computer representing a contiguous range of VA space. A physical computer may contain more than one node.
vTable	A vTable is a list of function entry points which are well known. The vTable translates the well-known entry point to the implementation address.
VA	Virtual Address
zero-pages	Pages which return all bits equal to zero when accessed. The data is not required to exist in storage.

APPENDIX B

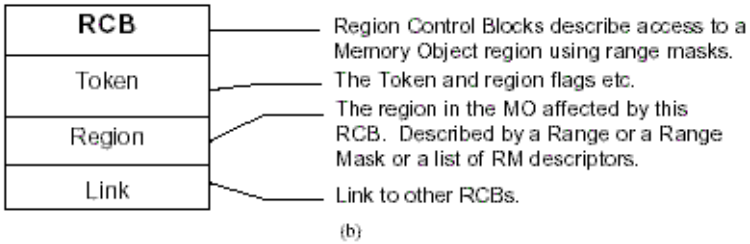
SELECTED FIGURES



(a)

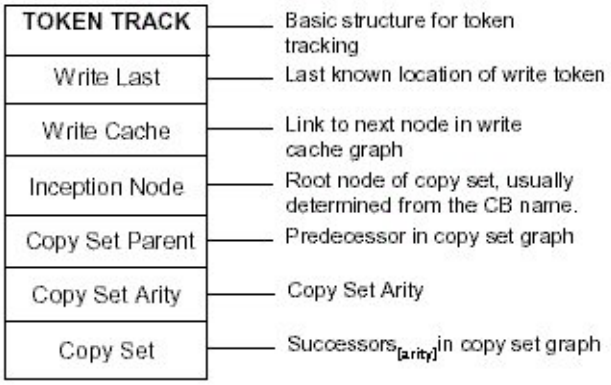
B.1 MEMORY OBJECT CONTROL BLOCK

Note: GCBs contain information about GPDs and TCBs contain information about CPDs. A PCB is a "Principal Control Block." A "Principal" is a controlling entity similar to an owner or user.



(b)

B.2 REGION CONTROL BLOCK



(h)

B.3 TOKEN TRACKING STRUCTURE

Note: In this thesis, “Write Last” is called “Last Known Writer” and “Write Cache” is called “Modified Page Cache.”