WWW in DSVM

by

SARAVANAN COIMBATORE

A thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba

0-612-23261-1

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

WWW in DSVM

by

Saravanan Coimbatore

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

Master of Science

R. Saravanan Coimbatore © 1997

WWW in DSVM

Saravanan Coimbatore

Master of Science, 1997

Department of Computer Science, University of Manitoba

Abstract

The World Wide Web (WWW) can be viewed as a very large distributed system consisting of multiple servers and clients. This thesis presents a model and prototype implementation of a system that applies distributed system management techniques to the WWW. This system (called WWW in DSVM) provides a large distributed virtual memory that is shared across a set of browsers and servers in a network. The model is based on the Distributed Shared Virtual Memory (DSVM) project under development at the Advanced Database System Laboratory (ADSL) of the University of Manitoba. WWW in DSVM provides sharing of retrieved documents among browsers through its underlying shared virtual memory paradigm. The major benefits of this approach are:

1. *Improved document retrieval in light of server and link failures,*
2. *Increased speed of access since documents can be retrieved locally from other browsers,*
3. *Reduced network load and server load since a separate access to a server by each browser is no longer required.*

A prototype WWW in DSVM system is implemented and has a demonstrated capability of sharing documents among browsers in a local network, in an intranet, and across the Internet.

Acknowledgments

First, I would like to thank my supervisor, Dr. Randal Peters for showing me the right directions in research; needless to mention his pleasing smile and motivating ideas. I would also like to thank my thesis committee members Dr. Peter C.J. Graham (for his useful comments) and Dr. David Blight, for their time and remarks in preparing for the thesis defense. Finally, my thanks goes to my parents and friends for their emotional support and encouragement all through.

Contents

List of Figures

Chapter 1

Introduction

The World Wide Web (WWW or Web) can be characterized as a large collection of servers and clients that are interconnected by the Internet. Increased use of the WWW has lead to network congestion and high server loads. In part, these problems are attribute to the basic operation of the WWW which requires each client to individually contact the appropriate server for a document, regardless of the client and server locations. For example, if two clients located on a desk next to one another wish to retrieve the same document from a server across the world, they must each contact the server and the document must be transferred across the Internet twice (ie., once to each of the two clients). There is no cooperation between the clients and the server in the retrieval of documents. If instead the document can be sent to the first client and the second client retrieves it from the first, there will be reduced network traffic and server loads. This thesis develops an architecture and prototype implementation of a system that provides this kind of cooperation and functionality.

1.1 WWW overview

From its beginnings at CERN [2], research has been conducted to improve the performance of the WWW. The WWW provides a graphical, easy-to-navigate interface for viewing documents on the Internet. These documents, as well as the links between them, comprise a "Web" of information. The Web lets one jump or "hyperlink" from one document to others. The Web can be thought of as a large library of information with an *ad hoc* organization. Web sites are like the books, and documents are like specific pages in the books. Pages on the Web are comprised of multimedia information such as images, movies, sound, application code, and so on. These pages can be located and accessed anywhere in the world.

The Web is a very sophisticated and elegant source of hypermedia documents. The Web offers not only access to an impressive magnitude and depth of information, but also a standardized hypermedia format, which makes navigation of the Web quick, intuitive and consistent, regardless of computer platform. This interactive interface places the Web far beyond other wide-area services. The servers in the WWW are the workstations that store and transfer their documents to clients that request them. The clients in the WWW are the browsers (eg., Netscape, Internet Explorer) that make requests and receive documents from the servers. Documents in the WWW are transferred using the Hypertext Transfer Protocol (HTTP) which is an application-level protocol for distributed, collaborative, hypermedia information systems. Documents in the WWW are coded using the HyperText Markup Language (HTML) [4]. Hypertext differs from regular text in only one regard: hypertext incorporates markup tags that specify formatting or special characteristics of the document. One of these tags is a hypertext link that provides information to jump to other documents. For example, a hypertext document may include references to a glossary, so that whenever a reader encounters a new term, that word contains a hyperlink to its definition in the glossary that the user can access with a single mouse click.

The Web includes a vast amount of information, from many different categories. The biggest sources of information are universities, colleges, and research centers around the world. Current research results, programs of study, papers, and exhibits represent only a few types of information available from these servers. Government agencies, such as NASA, also have Web servers, and provide text and graphical information regarding current projects. Currently, the Web is the most sophisticated, and readily available information system available on the Internet. The Web has steadily grown since its implementation and will continue to grow.

1.2 Problem Area

The increasing use of the WWW results in high network traffic over the Internet. One solution is to maintain mirror sites for those WWW sites that are expected to face high network congestion. A mirror site is a duplicate of the documents of one server at a geographically distant location. The idea is to distribute network traffic for these documents to different parts of the Internet. It is not practical, however, to have mirror sites for all WWW servers. For those sites that are not mirrored, the only option available to users is to retrieve documents from the original server. This can result in long retrieval times because servers become overloaded or are simply geographically distant.

Other problems that plague the WWW are server failures and communication link failures. When a WWW server fails, or all communication links between a server and a client fail, retrieving documents from that server becomes impossible until the server functions again, or the communication links are restored. The clients may cache these documents, but the cache is private to individual browser sessions. The assumption made in this thesis is that similar sets of WWW documents are typically requested at multiple clients. If clients cache these documents, then the copies are potentially accessible from multiple locations.

A document in the WWW is linked to another document through a hypertext link.

A common way of reaching relevant documents is to traverse through the hypertext links. Problems arise when a document is deleted from the WWW as the Uniform Resource Locator (URL), the "address" of the document becomes invalid and the document cannot be accessed (known as the "broken-link problem"). There can be many other documents referencing a deleted document and users trying to locate a document without knowing that it is deleted can find themselves at a "dead end" after traversing a series of links. Unnecessary Internet connections are made when many users in a local network try to retrieve a document that has been deleted in the WWW. This can be alleviated by sharing document status between users.

A document can also move from one location/URL to another. A possible solution for maintaining links to relocated documents is to introduce forwarding pointers to the new location. However, as documents move, the number of forwarding links increase and traversing through the links becomes cumbersome and inefficient. Traversal of surrogate links, which requires WWW server (re)connections, can be avoided by sharing documents across clients because the results of document traversal performed by one client can be shared by others so that they do not have to individually go through the same process. That is, if one client has traversed the links and retrieved the document, then subsequent requests from other clients can get the document that was retrieved by the first client without traversing the links again.

This thesis presents a model and prototype implementation of a system that applies distributed system management techniques to the WWW. This system (called *WWW in DSVM*) provides a large distributed virtual memory that is shared across a set of browsers and servers in a network. The model is based on the Distributed Shared Virtual Memory (DSVM) project [7] under development at the Advanced Database System Laboratory (ADSL) of the University of Manitoba. WWW in DSVM provides sharing of retrieved documents among browsers through its underlying shared virtual memory paradigm. The major benefits of this approach are :

1. Improved document retrieval in light of server and link failures,
2. Increased speed of access since documents can be retrieved locally from other browsers,
3. Reduced network load and server load since a separate access to a server by each browser is no longer required.

A prototype WWW in DSVM system is implemented and has a demonstrated [23] capability of sharing documents among browsers in a local network, in an intranet, and across the Internet.

The remainder of this thesis is organized as follows: Chapter 2 discusses the related work on improving document access in the WWW. The existing technologies that motivate this model are presented in the Chapter 3. Chapter 4 presents the general model architecture, its operational details, and the design issues. The proof of concept prototype and its execution are presented in Chapter 5. Finally, conclusions and future work are given in Chapter 6.

Chapter 2

Related Work

Several research efforts are ongoing to provide an efficient way of retrieving documents in the WWW despite high network traffic. One mechanism commonly used is to have multiple mirror sites for those servers that are expected to have a high network load. When a client requests a document, the mirror site closest to the client is selected for information retrieval. Automatic selection of a mirror site is done by the main Web site based on the client's location [3]. However, it is not practical to have mirror sites for all Web sites. WWW in DSVM effectively provides a focussed and dynamic mirroring of certain documents to a set of participating browsers. In this thesis, a document retrieved from an overloaded Web server is shared by other browsers in the local network thus avoiding unnecessary delay due to server and network failure for subsequent requests for the same document from other browsers. In other words, a mirror site has been created dynamically inside the local network.

Another popular mechanism to reduce the network load is to cache documents at the client side. WWW in DSVM takes this concept a step further by allowing the sharing of retrieved documents between clients. That is, the clients share their cached documents. The performance benefits of caching documents close to clients have been pointed out by several researchers. Yoshida [32] presents a distributed Web and cache server called MOWS. MOWS loads modules that are present either locally or remotely. These modules perform the duties of Web and cache servers and run as a cluster of distributed Web servers. A request for a local byte code instruction retrieves it from the MOWS system, while a remote byte code instruction retrieves the code from the remote site and executes it. The details of other modules and MOWS servers are maintained in a module called "JMEDirectory" that specifies the URL address of the modules and the location of the directory. The MOWS system communicates with other proxy servers or MOWS systems to retrieve a document if the original server is down. This could introduce communication overhead especially when the MOWS servers and proxy servers are far apart.

Nabeshima [20] proposes a domain cache server that handles access to a particular domain name. Documents are prefetched during light network usage. Document coherency is maintained by using the original server *Refresh* information. A cache replication server services the documents and acts as a HTTP server rather than just a proxy server. However, multiple simultaneous requests to the domain server can result in very high network congestion and performance degradation.

Cache replacement algorithms have been extensively studied in the WWW community to refresh or remove documents in the proxy cache. Wooster and Abrams [30] present two removal algorithms for cached documents that consider the retrieval time for each document and cache only those documents that result in a very long connection time. The algorithms work by estimating the Web page download delays or proxy-to-Web server bandwidth using recent page fetches. The algorithms are compared with the three existing policies (LRU, LFU, and SIZE) using measures like user response time, Web server loads, and network bandwidth consumption. However, network load is a time-variant factor that depends on the number of connections at any given time. A document that was retrieved during low network load may not be cached. If a subsequent request for the same document occurs during a high network load period, then the document has to be retrieved from the WWW.

Another replacement algorithm that considers the retrieval time is presented in [24]. The algorithm tries to maximize the metric called delay-savings-ratio by considering the fact that Web clients tend to retrieve small documents most of the time. However, the performance of the algorithm degrades when larger documents are accessed frequently.

PURL (Persistent URL) [6] is an approach that provides a global naming scheme for addressing the document relocation problem. PURLs are similar to URLs, but PURL addresses are resolved using a PURL resolver. When a document is relocated, the PURL maintainers update the information in the PURL server. Our approach provides a similar global naming space for documents, but our model does not currently provide a global name resolving service as in PURL. However, as part of our future work, we are integrating the WWW servers into DSVM and then *object identifiers* (OIDs) will be similar to PURLs in that they will always reference the correct document in DSVM. Documents in Web servers will exist in DSVM with their addresses resolving to unique OIDs.

Uniform Resource Name (URN)[26] is a naming scheme in which global names are assigned to resources in the WWW. A URN stores the characteristics of the document and a list of the URLs where the document is replicated. The name resolution services in this scheme decompose the URN into URLs. Different resolution services generate different sets of URLs and users can choose the resolution service. Our model also stores a list of node addresses where the document is replicated in the Global Directory of Objects (GDO) entry in DSVM and maintains consistency between document copies. In the future, the GDO entry can be modified to have metadata about the document to support a searching and resolution mechanism.

Kosuge and Morita [17] have used replication of management information in the

WWW to avoid network congestion. The management information is divided into two records, one contains the last modified time of the information resource file and the other contains the location of the mirror sites. The protocol uses the Domain Name Service (DNS) for exchanging information. In our model, the management information is fragmented and replicated across multiple nodes in a local network.

A proxy caching server retrieves and caches documents for the browsers connected to the proxy. In this sense, the proxy cache is shared among the attached browsers. Thus, a document retrieved by one browser may be passed to another browser from the proxy cache without having to access the Internet. This is a centralized approach that does not scale well when the number of browsers wishing to share documents increases. Furthermore, the caching capacity of the proxy is limited and more replacements will occur as the number of browsers increase. Our approach distributes the management of documents across the participants and hence distributes the workload, and provides good scalability as the shared environment grows. Moreover, the participants have a large (2^{64} or greater) shared address space for maintaining retrieved documents.

Ingham, *et al* [15] propose a solution for the broken-link problem in the WWW. In their solution, forward reference mechanisms, a name service, and call back mechanisms are used. The solution is based on an object oriented approach and each resource in the WWW is considered to be an object. When an object moves from one space to another, the first request for the object traverses through the forwarding links. After reaching the target object, the requested object updates itself with the new location of the object (this is known as a *Forward Reference* mechanism). The model manages the broken links (created due to deletion of objects) by using a naming service and call back mechanism. Every object registers itself in the naming service. When an object is deleted, all objects registered in the naming service that refer to the deleted object are informed about the deletion. However, as the number of referencing objects increases, informing these objects about new locations becomes expensive.

In the DCE Web project [18], security issues of WWW documents are discussed. The approach applies the OSF Distributed Computing Environment(DCE) features to the WWW. The environment consists of a DCE Web, which is a set of DCE capable servers grouped together to form a cell. Communication between cells is by Remote Procedure Call (RPC) for DCE capable servers or HTTP for non DCE capable servers. The DCE Web has a naming service in which the server maps the document into a DCE name that keeps track of the location and authorization of the document. The DCE naming services provide name-based document lookup in the DCE Web. A server for a document makes the document available at the same DCE name irrespective of the server location. The browsers use the naming services to bind to the correct server. However, the URL referring to a DCE document includes the DCE name embedded in the URL. Thus, non-DCE browsers have to go through a proxy server to gain access to the DCE Web documents. In this thesis, separate communication mechanisms (RPC, IIOP) can be used between browsers and servers inside DSVM and HTTP can be used to communicate with other Web servers outside DSVM.

Wide area file systems [25] are similar to indices in the WWW. When a client requests a document, the URL is translated into a corresponding database entry and the database is queried. If the information is not found, it is retrieved from the WWW. If the document is available, then it is returned to the user. Wide area file systems support global name space, location transparency, data replication and access control mechanisms. A request for a document inside a Wide area file system queries the local host first and if the document is not present in the local host, it is converted into a URL and retrieved from the WWW. The decision of which documents to cache in the system is left to the user and each document is given a document refresh interval. This is similar to the proxy caching server mechanism which is a centralized querying approach. The system is similar to our approach in the sense that it provides a global naming space and location transparency for documents. However, ours is a distributed approach to manage cached documents that employs a weak consistency mechanism to minimize network traffic and provides better scalability and reliability.

Several researchers have tried to integrate distributed system concepts in the WWW. Pagespace [12] is an approach to support open distributed applications on top of the WWW.

Pagespace implements a set of user agents that reside on every machine to manage user interface, perform user services and form applications , integrate services provided within co-ordination environments like CORBA, DLE and Pagespaces. Pagespace implements a set of agent classes and is built on co-ordination technology (platform for co-ordination of activities among asynchronously working agents), Web interface (for widespread communication and presentation) and Java technology (for uniform processing platform).

Yang and Kaiser [31] propose a scheme for integrating the WWW and object oriented databases to support dynamically customized presentations of WWW information to users. The information is structured based on user needs and the application at hand. For example, a document may be displayed in different formats to different users. A document may display all information in a database to a super-user and may display only selected fields for other users. The browsers are linked to the database by using a HTTP implementation. The HTTP implementation provides the interface to the WWW. User views are dynamically generated by a View Processor. The HTML file is embedded with customer specific instructions. These instructions are interpreted by the view processor to generate customized documents. The major disadvantage of this approach is that the user must modify the source files to include the embedded methods.

Chapter 3

Technical Background

There are a number of technological achievements that motivate this research. This chapter gives a detailed overview of these base technologies and introduces how they are combined into a system that provides improved access and usability of the WWW. The *distributed shared virtual memory system* (DSVM) is used to provide a distributed cache among browsers in a network for sharing documents. A proof of concept prototype of this model that supports HTTP features is developed to illustrate the feasibility of this approach. The distributed shared memory platform for the prototype is provided by Treadmarks. Shared memory consistency is done by using the Treadmarks functions. The interface of the shared memory manager with the WWW is implemented by using the Web interface library provided by Perl.

3.1 Shared Virtual Address Space (SVAS) Model

Traditional virtual memory systems operate on a *Private Address Space* (PAS) scheme where each application has its own address space and applications use messages and files to communicate. The *Shared Virtual Address Space* (SVAS) paradigm [22, 10] has a single, common, virtual address space that is shared among all applications.

In the PAS scheme, each process has its own virtual address space that starts at virtual address zero and accommodates the size required by that process. In a multi-process application, separation of the address space makes it difficult to share data across processes. There are many approaches to minimize this problem. One approach is to use messages and files to communicate changes to other processes. A disadvantage with this approach is the potentially large communication overhead. Another way is to use shared memory segments mapped to the same virtual memory location in all processes. In a PAS system, pointers lose their significance when passed from one application to another because shared data in one application's PAS may appear at a different virtual address in another application's PAS. One possible way of solving this problem is to have a single global address space across all processes to provide uniform memory access as all data appears at the same location for all processes.

In the SVAS scheme, a programmer has the advantage of a single address space which provides ease of programming by eliminating coding of cumbersome communication procedures (eg., sockets, RPC) between applications. Data sharing is easier in the SVAS approach since all data appears to all processes at the same virtual address. In an SVAS system, the addresses are valid across all sites and can be accessed at any site, i.e. the addresses are $context - independent$. Context-independence has advantages both in hardware usage like efficient address translation and in software like sharing, storage and retrieval of structured data containing pointers.

An SVAS system can be page based, shared variable based, or object based. Furthermore, an SVAS can be distributed and shared across nodes (workstations) of a network. In a distributed object-based SVAS system, applications on multiple nodes share an abstract space filled with objects. The distribution of objects is transparent and objects can move freely between sites. Access to a non-resident object will obtain the object from the site that holds the object. Any application can invoke any object's methods (provided proper permissions have been granted), regardless of where the application and object are located. From the programmer's perspective, methods can be invoked without any additional complexity like RPC, function shipping etc., as the objects share the address space. The underlying system may use these techniques to actually execute methods, but this is hidden from the programmer.

Context switching is efficient, as address translations and page table contents are valid across method invocations. The object-based SVAS approach also has more advantages than other SVAS approaches including modularity, flexibility and integration of access and synchronization. In this approach, a distributed object-based SVAS provides the underlying architecture where retrieved WWW documents are managed and shared.

An object-based SVAS system can be designed to support persistence. Since persistent objects are expected to exist for a long time, the address space of the process must be unchained from the process itself. This is possible when we have a large single address space such as those provided by 64-bit architectures (e.g., DEC Alpha, IBM RS-6000, etc.). Moreover, distribution of objects can be made transparent, while the actual operations are performed at the system level.

Although the SVAS system has advantages, there are issues like security that must be addressed. One approach to program securely is through systems that provide hardware protection based on hardware supported protection domains rather than the address spaces [28]. The working environment in this thesis is a persistent object system in a single shared address space environment. Security aspects of this environment are not considered in this thesis.

3.2 Distributed Consistency Protocols

In a distributed environment, information is shared between processes across networks. In such a case, virtual memory can be distributed across machines. Changes in the shared virtual memory of one process should be reflected in the virtual memory of a process running on another machine.

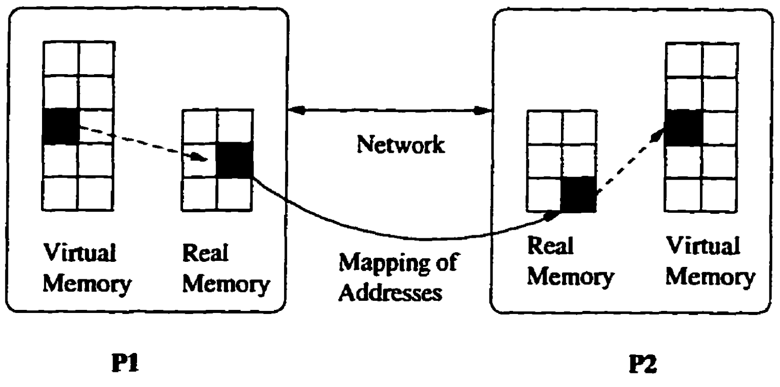


Figure 3.1: Memory Mapping in a SVAS system

Figure 3.1 shows memory mapping between two processes P1 and P2. For maintaining consistency of shared memory between the two processes, the operating system must be actively involved. When there are multiple processes storing the same information, updates to a shared memory location in one process must be propagated to all other processes so that the information remains consistent between processes. Shared memory consistency can be maintained by two mechanisms. In *strong consistency* mechanisms, when a shared memory location is updated, the new value must be propagated to all other processes so that subsequent reads will see the new value. However, frequent writes to data result in frequent updates to cached copies and high associated overhead. The communication overhead can be reduced by using some form of *weak consistency*. In weak consistency mechanisms, the updates are sent to other processes only when demmed "necessary". Two protocols that maintain weak consistency between data items are the "release consistency" and "lazy release consistency" protocols.

Release consistency [33] was first proposed to reduce communication overhead by avoiding unnecessary updates. In this mechanism, consistency of shared data is ensured only at lock-release time, which minimizes the overhead of propagating intermediate updates to other copies. A table indicating data items that are stored in each processor is maintained and explicit control of the virtual memory management is needed to propagate updates. At any point of time, either all processors have up-to-date information or only one process (the one that holds the lock) has the latest information. This protocol has some deficiencies as shown in Figure 3.2.

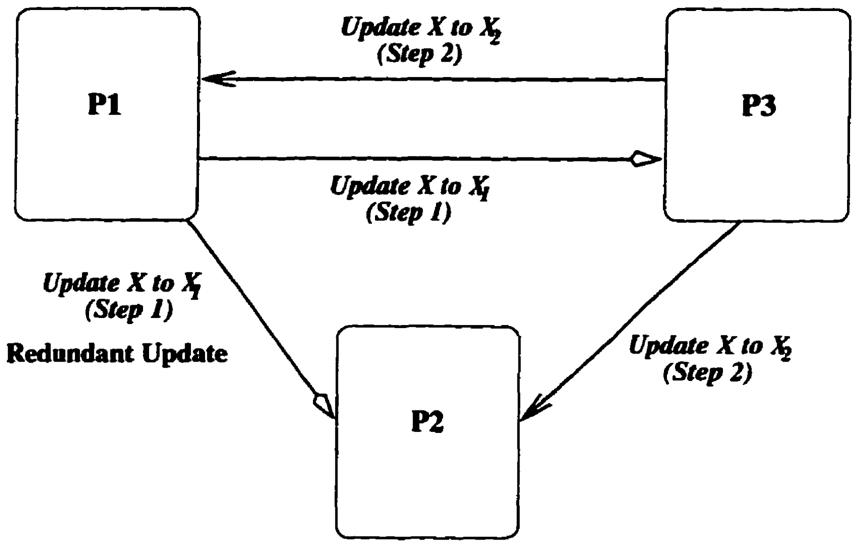

P1
P3
P2
Update X to X_2 (Step 2)
Update X to X_1 (Step 1)
Update X to X_1 (Step 1)
Redundant Update
Update X to X_2 (Step 2)

Figure 3.2: Release Consistency

Figure 3.2 shows the update operations on data item X by three processes P1, P2 and P3. A process (say, P1) locks the data item and updates it (say, $X1$). Upon lock release, the new value $X1$ is propagated to the *two* processes P2 and P3. Now, process P3 locks and updates the data item (say to $X2$) and propagates the value to other processes. The data items at P2 and P1 are updated with the new value. As shown in the figure, propagation of data item X from process P1 to process P2 is unnecessary as the data propagated by P1 to P2 was modified by P3 before P2 reads the data.

This problem was solved by *Lazy Release Consistency* (LRC) [16]. In this approach, consistency is maintained during lock acquisition rather than during lock release. Updates are propagated only to the process that acquires the lock. To provide information about which process has most recently updated the data item, a lock manager maintains the identifier of the holder of the most up-to-date copy of the datum. Thus, when a new process acquires a lock, the "home node" (lock manager) sends the identifier of the last updater to that process. A request is then sent to the last updater (if it is not the same one that just acquired the lock) for that data item. Once the data is received and updated, upon lock release, the lock manager updates itself to record the most recent updater of the data item. By this method, the unnecessary updates in the release consistency protocol are avoided. The proof of concept prototype in this thesis uses Treadmarks to provide consistent shared memory between interconnected nodes. Treadmarks uses the LRC mechanism to maintain shared memory consistency.

3.3 Persistent Storage Management

Persistent data is long-term, non-volatile storage that exists between program invocations to store data that must be preserved for future reference (e.g. databases). A persistent system is one in which data created by a process persists beyond the termination of the creating process. A process creating persistent data need not be concerned with explicit operations to store the data to a file. Subsequent processes accessing the persistent data need not issue explicit file read operations. In a persistent system, the user need not know the file system semantics and operations. The file system calls need not be coded explicitly to store and load persistent data. Also.

there is no need to transform data between in-memory and on-disk data formats (eg., dumping a linked list to disk).

While implementing a persistent system, addresses are transformed into persistent references. A reference to persistent data will consist of an identifier with (typically) more bits than an address. For example, suppose a process refers to a data file "/home/cs/grad/crs/datafile" during its program execution. The data file has to be read into memory for processing. When referring to such a persistent reference, directory details are needed for loading the file into the memory. The persistent reference requires more bits to specify all the necessary information. On the other hand, a memory address consists of a fixed number of bits. When data is in memory, it can be accessed by memory addresses (minimal number of bits) but when it is non-resident, it must be referenced using a larger identifier. Thus, persistent references must be dynamically converted into in-memory references before data can be accessed (this is called *pointer swizzling* [29]).

Swizzling can be eager or lazy, based on whether the objects referred to by an in-memory object are pre-swizzled (eager) or not (lazy). In either case, a table mapping the persistent identifier to the memory address is maintained to avoid multiple mappings of the same object. As the number of objects brought to the memory increases, the size of the table increases. This overhead can be avoided by swizzling objects on a per-page basis at page-fault time [29] when the page is brought into memory. There is no need to keep track of an object's status in a separate mapping table since objects in memory are swizzled as they are brought in during page-fault. If the page is in the memory, page-faults will not occur.

3.4 Treadmarks

Treadmarks is a distributed shared memory software package that provides shared memory between multiple workstations interconnected by a network.

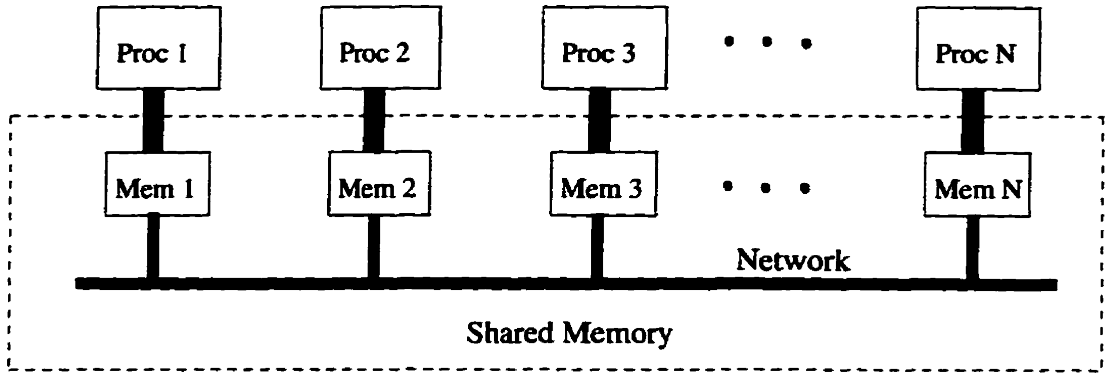


Figure 3.3: Distributed Shared Memory

Treadmarks provides the abstraction of a global shared memory across workstations. The programmer need only be concerned with memory addresses and need not worry about the site where the data is stored or the procedures by which the data is transmitted. The Treadmarks application programming interface has many methods for process creation, destruction, synchronization, shared memory allocation and maintenance. Treadmarks also provides two synchronization primitives (locks and barriers).

A *lock* is a simple form of synchronization object and supports two operations. The *acquire* operation provides control of a lock for a particular process and the *release* operation releases the specified lock. Once a process locks a portion of the shared memory, other processes cannot access that particular shared memory segment. The programmer associates a set of privileges to a lock and a process has to acquire the lock to use the set of privileges. The process releases the lock after it is done with the privileges. For example, a lock can be associated with the privilege of updating an object and another lock can be associated with reading the same object. Thus.

to read or update the object, the requester should acquire the corresponding lock to get the read/write privileges. Locks are used to synchronize multiple writes to a data item by different processes at the same time.

Different processes may be at different program segments during execution of the program. A barrier halts the progress of a process until all of the other processes reach the same state as of the current processes. Ideally, the *barrier* primitive is used by the process that allocates and distributes shared memory between other Treadmarks processes.

The following are some Treadmarks functions used in the development of the prototype WWW in DSVM system.

*Tmk_startup(int argc, char **argv)*	Initiates the Treadmarks processes in the workstations specified in the initialization file .Tmkrc.
Tmk_exit(int status)	Terminates the calling process with value "status".
Tmk_malloc	Allocates shared memory between the Treadmarks processes and returns a pointer to a block of shared memory. Only the memory that is allocated by this function is shared.
*Tmk_free(char *ptr)*	Deallocates the shared memory pointed to by *ptr*.
Tmk_distribute	Distributes the contents of a block of PRIVATE memory on the calling process to every other process. After the call, all processes have the same information in speci-

	fied memory locations. Normally, this function is called by the main process which allocates the shared memory.
TMK_NLOCKS	A constant that contains the number of synchronization objects provided by Treadmarks.
Tmk_lock_acquire(id)	Blocks the current process until it acquires the specified lock. The parameter *id* represents a lock number in the range 0 .. the TMK_NLOCKS - 1
Tmk_lock_release(id)	Releases the lock specified by *id*.
Tmk_proc_id	Variable that contains the current Treadmarks process identifier.
Tmk_nprocs	A constant that contains the actual number of parallel processes after Tmk_startup.

The Treadmarks model requires shared memory management among dynamically invoked processes. Every invocation of a browser is a process that should be able to access the shared memory. Also, at any instant in time, there can be multiple requests from the browsers. To service these requests simultaneously, the shared memory manager must provide shared memory management between dynamically invoked processes and the main process. Treadmarks is used to provide shared memory between workstations to develop the prototype. However, Treadmarks only provides and maintains shared memory between workstations that are specified in the configuration file ".Tmkrc" during system startup. Other processes cannot access the shared memory. Hence, the shared memory manager is designed as servers that store and retrieve documents from the shared memory. Each server services one request at a time and there are multiple servers to service document requests. Another requirement with Treadmarks is that each Treadmarks process that is invoked requires 40MB of swap space.

3.5 DSVM

This thesis is based on the ongoing DSVM (Distributed Shared Virtual Memory) project [13, 7, 14]. The DSVM project environment is a distributed persistent object system and uses the distributed object-based SVAS mechanism as a basis for object sharing. DSVM uses a large virtual address space such as those provided by 64-bit architectures (e.g., DEC alpha) [11] to create a persistent distributed shared memory. Objects are placed in persistent memory and shared transparently between interconnected nodes (workstations) of a network. Implementing such a system has many advantages including simplified programming (both persistence and distribution are transparent) and elimination of swizzling of object references. The object model of DSVM is "vanilla flavored" but supports the object concepts of encapsulation, inheritance and polymorphism. Meta information is used for accessing data in the model. Objects in this model consist of state (attributes) and behavior (methods). Each object is identified by a unique identifier (OID), which is the object's virtual address in the persistent object space. Each object is instantiated from a given type that specifies the attribute structure and method implementations. Objects instantiated from a type belong to the same class. Method executions are treated as nested atomic transactions.

The high level logical system architecture of DSVM is shown in Figure 3.3. The system uses a stub process approach for different legacy systems (eg., relational, file-

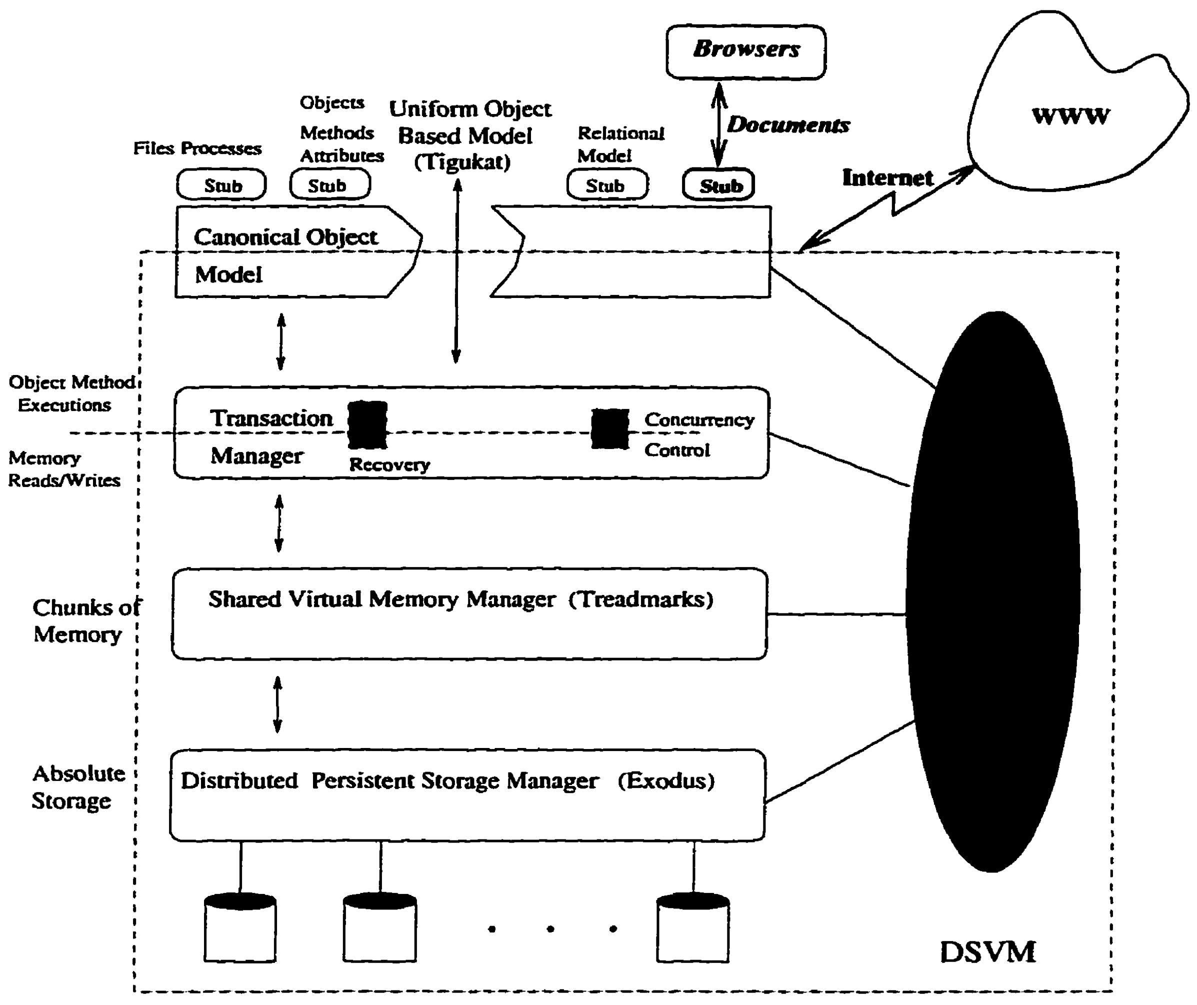


Figure 3.4: DSVM Architecture

based applications) to interact with the canonical object model. The object model is based on the uniform behavioral model of the Tigukat OBMS [21]. The transaction manager deals with the concurrency control and recovery issues of DSVM. The relevant portions in DSVM that are of direct concern to this thesis include the "Memory Manager" and the "Storage Manager". The persistent storage (disks) is managed by the persistent storage manager, which in the prototype DSVM system is implemented as a set of co-operating Exodus [9] servers. The shared memory manager provides a single virtual memory abstraction among processes at all nodes and maintains shared memory consistency across sites. The prototype DSVM system uses Treadmarks [8] to provide shared memory across sites. Treadmarks does not provide persistence, so the prototype implements an interface to the Exodus based storage manager.

Objects in DSVM are visible to applications at all sites. Each site has a DSVM process and methods are executed as threads that access objects in the shared memory. To ensure efficient mapping of virtual memory, *inverted page tables* (e.g. IBM RS-6000) can be exploited. Security and privacy of objects can be provided by software protection mechanisms [27] as well as more efficient hardware protection mechanisms [28] that enforce restricted access to hardware protection domains.

The DSVM system uses an object repository called the Global Directory of Objects (GDO) [19] that provides distribution, efficient data retrieval, and scalability. Nodes in a local network are clustered into DSVM-cells consisting of several nodes per cell [19]. The GDO is fragmented and distributed across the cells and may be replicated on several nodes within a cell. Each cell manages a disjoint range of virtual addresses within the 2^{64} possible addresses (see Figure 3.4). Objects are created in the SVAS with virtual addresses serving as object identifiers (OIDs). After an object is created, it can be made persistent in the SVAS. The GDO fragments use an index structure (eg., B+ tree) for efficient searching and insertion of objects. Each GDO entry consists of the OID, object size, an *exists* field to indicate the status of the object (deleted or not deleted), lock variable to lock the object, pending lock requests queue, and persistent storage location references. Other fields in a GDO entry are optional and depend on the application that is executed on DSVM.

The *Lazy Release Consistency* (LRC) [16] protocol of Treadmarks is used to ensure the consistency of objects between nodes with minimum communication overhead. In the LRC mechanism, a "home node" is defined for each object and the information about the last node that updated the object is stored at this site. Access to an object

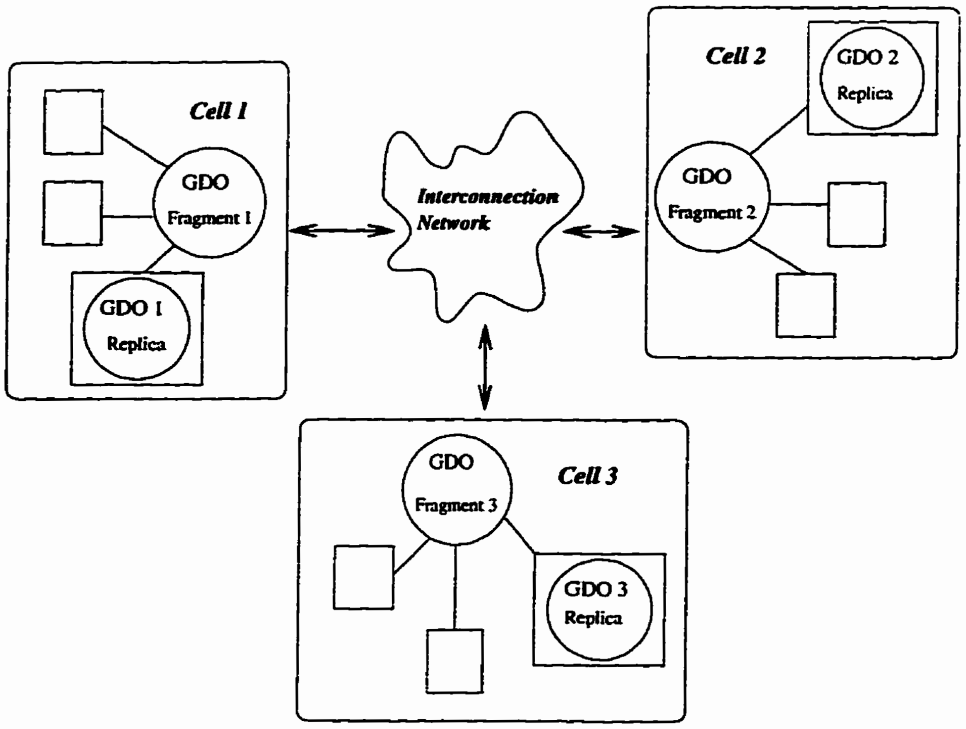


Figure 3.5: Placement of GDO fragments

requests the site that last updated the object from the home node and retrieves the object from this latest updater. In the general architecture for supporting WWW in DSVM, the WWW browsers form the nodes in DSVM and each URL maps to a virtual address (i.e., OID) that represents a document as an object in the SVAS of DSVM.

An object in the object space consists of (i) an indexed jump instruction that invokes the relevant object method, (ii) attributes of the object and (iii) method code for the object. Implementation of the management functions on the shared address space is based on detecting object references and then exerting special control on memory management. During startup, the physical memory and the virtual address space of all processes are empty, but the GDO is accessible at some known location. When a transaction invokes a method of an object, a reference to the virtual address of the object is generated. As the virtual address is undefined, this results in a mapping fault. The GDO is then consulted with the virtual address of the object. If an entry is found, then page tables are constructed using the information in the GDO. After an object is mapped, the page faults are taken care of by demand paging.

3.6 World Wide Web

The World Wide Web (WWW or Web) has progressed from its humble beginnings at CERN [2], as a means to effectively transport research and ideas throughout the organization, to a global service used by millions each day.

The popularity of the Web has sparked efforts in both the commercial and research communities. There are many good commercial browsers that offer user-friendly graphical interfaces to the traditionally hard-to-master resources of the Internet. An assortment of standard protocols (HTTP, FTP, etc.) are integrated into the Web environment. There are extensive research efforts to increase the types of services available on the Web (video, sound, conferencing, Java, etc.) as well as efforts to provide more efficient access to Web documents while reducing network and server loads. This research focuses on the latter by providing a shared memory paradigm where a set of heterogeneous browsers, distributed across a network, share documents with one another and form a co-operative Web environment.

3.6.1 World Wide Web Protocols

The WWW consists of a body of information protocols, standards, and conventions that govern their use. These information protocols can differ greatly from one another but they all allow clients and servers to communicate. Some of the most common WWW protocols and utilities are 1) Resource Addressing, 2) Data Transfer, and 3) Proxy Servers.

Resource Addressing

URLs (*Uniform Resource Locators*) are a standard for identifying objects in the WWW. Every document has a URL that serves as its network-wide address. Documents can form links to others by including their URLs. A URL is a string of characters that uniquely identifies an object in the WWW (like a catalog number in the WWW). The URL describes any object anywhere on the Internet and these objects are accessed using different protocols.

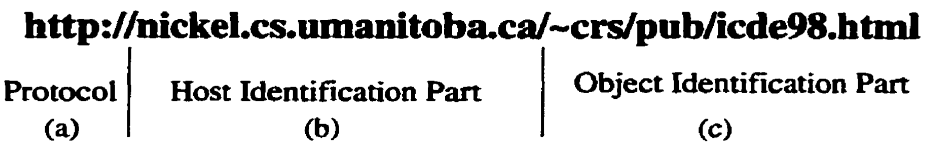

http://nickel.cs.umanitoba.ca/~crs/pub/icde98.html
Protocol
(a)
Host Identification Part
(b)
Object Identification Part
(c)

Figure 3.6: URL Structure

A URL has three basic parts as shown in Figure 3.6:

1. Protocol : Figure 3.6a shows the protocol portion of the URL. This portion specifies the protocol to be used by the browser to communicate with the WWW server. In most cases, the standard protocol is either "HTTP" (for objects on a server) or "file" (for local objects). Other protocols include common Internet protocols such as FTP and Gopher.

2. Host Identification : The host identification portion of the URL represented by Figure 3.6b identifies the server where the required information resides. Normally, the host ID consists of *cluster name*, *domain name*, and *port number*. Most WWW servers use port 80 as the default port. The standard Internet Domain Name Service (DNS) resolves these names into a numerical Internet address. In some cases, some servers may not be registered or the local domain name server may have problems. Alternatively, a system can be specified by

its numerical Internet address, rather than by the combination of system name and domain name.

3. Object Identification : Figure 3.6c identifies the object identification portion of a URL. There are two different types of identifiers. One of them refers to text files (HTML files) and the other refers to executable programs. The file identification consists of i) *directory path* that is always an absolute path and must begin with a slash to separate it from the server, ii) *filename* that specifies the object to be transferred, which is most commonly an HTML file (other types such as an executable program are also allowed, depending on the protocol being used), and iii) *section name* that defines sections in an HTML file. Most HTML files are not broken into sections, so a section name is not normally used.

URLs map to documents in the document tree on a WWW server. The minimal URL that can reach a server is:

http://server-name

If the WWW server is run on a port other than the default port, the nonstandard port number is included in the URL, for example:

http://server-name:1300

During translation of a URL to real files, the URL is scanned for any "virtual paths" by the WWW server. The tilde prefix ($\sim$) followed by the directory name indicates a virtual path. If a virtual path is found, then it is replaced with the real directory and the request is processed.

Data Transfer : HTTP

HTTP (HyperText Transfer Protocol) is a stateless search and retrieve protocol for WWW operations. This generic, object-oriented protocol accommodates distributed.

collaborative hypermedia information systems. A client using HTTP sends a list of the representations it understands with its request, and the server can then ensure that it replies in a suitable way. HTTP allows communication between user agents (browsers) and various gateways. HTTP also employs a disciplined and consistent system for referencing different resources through the use of URLs and URIs (Universal Resource Identifiers - identifiers that include all names and addresses that are short strings referring to objects).

HTTP accommodates several commands or methods to perform a variety of tasks on an object identified by the URL. The GET method retrieves the data identified by a URI. The HEAD method is similar to GET except that the server returns only the document headers; it does not return the document body. The POST method is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the request line. The POST method covers functions such as: 1) annotation of existing resources, 2) posting a message to a bulletin board, newsgroup, or mailing list, 3) providing a block of data, such as the result of submitting a form to a data-handling process, and 4) extending a database through an append operation. The server responds with a 201 code and the status of the request if the entity is created. A valid "Content-Length" is required on all POST requests. A WWW server responds with a 400 (bad request) message if the length of the request message's content cannot be determined. Responses to a POST request are not cached because the client has no way of knowing that the server would return an equivalent response on some future request. This thesis supports the GET, HEAD and POST methods of HTTP.

All HTTP transactions take place over a TCP/IP connection and usually through the default port 80. An HTTP transaction consists of four stages.

1. *Connection* : In this phase, the browser (the client) attempts to connect with

the WWW server. The status of the connection is displayed on the status line on most browsers.

2. *Request* : After a connection is established, the client sends a request to the WWW server specifying the protocol to be used, the required document, and the document types it can understand.

3. *Response* : The server either sends an error message if it cannot fulfill the request or responds with the document if it can fulfill the request. The browser might display a "reading response" message or a "transferring" message on the status line. The server tries to send only those data types that the browser supports and responds first with the type of document that it is sending followed by the document.

4. *Close* : After the server sends the response, either the client, the server, or both close the connection.

An example request for a hypertext link such as:

<A HREF="http://domain.edu/thesis.html"> </A>

will send a request similar to the following:

GET thesis.html HTTP/1.0

Accept: text/plain

Accept: image/gif

Accept: image/x-portable-bitmap

User-Agent: NCSA WinMosaic 1.0

[A blank line containing only CRLF]

The request header defines several fields indicating the requested document and the capabilities of the browser. The first line in the request contains the requested document. The *Accept* lines are client information specifying the data types that the browser can accept. The *User-Agent* specifies the program making the request. The end of the request header is indicated by a single line containing only a CRLF (carriage-return line-feed) pair.

The WWW server responds with the document header followed by the document. A sample response to the above request is given below:

```
HTTP/1.0 200 OK
MIME-Version 1.0
Server: NCSA/1.3
Content-Type:text/html
Last-modified: Thursday, 3-AUG-95 23:37:8 GMT
Content-length: 145
[ A blank line containing only CRLF ]
  <html>
  <head>
  <title> Master's thesis </title>
  </head>
  <body>
  Document text
  </body>
  </html>
```

The first line in the response header represents the state of the response. If the document is not found, then the server responds with a 404 status. If the request is successful, the server responds with a 200 status followed by the document type (line 4), the last modified date (line 5) and the length of the document (line 6). The server may also send some more header details depending on the request. The end of the document header is specified by a blank line containing only a CRLF pair. The actual document is sent following the blank line. For a detailed description of HTTP see [5].

The prototype implementation of the WWW in DSVM supports HTTP features. The queries are formed by the WWW browser and the model uses these queries to retrieve documents from the WWW server. Requests from the browsers are sent to the WWW server without any modifications. The response from the WWW server is sent to the browser without any modification and is processed by DSVM to identify if the response was an error code or a valid document. For example, if the document has been deleted, then the status information in the shared memory is updated to "Document Deleted".

Proxy Servers

A proxy server (sometimes referred to as an application gateway or forwarder) is an application that regulates traffic between a protected network and the WWW. Proxies are often used instead of router-based traffic controls to prevent traffic from passing directly between networks. Many proxies contain extra logging or support for user authentication. Since proxies must "understand" the application protocol being used, they can also implement protocol specific security (e.g., an FTP proxy can be configured to permit incoming file transfer and block outgoing file transfer). There are many ways of protecting a local network from another network or illegal users. A proxy server may also function as a firewall for protecting the local network from illegal users. A firewall performs at least two functions: (1) it blocks traffic from passing between a local network and the WWW, and (2) it permits traffic flow between a local network and the WWW. The proxy server's duties are divided to manage the incoming and outgoing network traffic for better efficiency and to control the incoming and outgoing traffic. For example, one part of a firewall in a company can restrict people from sending resumes to companies outside the firewall and the other part can filter the retrieval of unauthorized sets of documents (like receiving job postings).

A proxy server listens for a request from a client within the firewall and forwards the request to a remote server on the WWW outside the firewall. The proxy server reads the response and sends it back to the client. Operation of a proxy server should be transparent to the users. Proxy servers can be useful in many ways. Proxy servers can permit and restrict client access to the WWW based on the IP address of the client. Documents can be cached by the proxy server. The advantages and disadvantages of caching documents in a proxy server are discussed later in this chapter. Access to the WWW and subnets can be controlled based on the submitted URL. Proxy servers can be used for providing WWW access for companies over private networks (Internet Service Providers). Clients who do not implement Domain Name Services (DNS) can reach the WWW through a proxy server. To use a proxy server, a client only needs the IP address of the proxy server.

Sometimes a browser may not be able to have direct access to WWW resources when it is run on a system behind a protective firewall. Under such circumstances, a proxy server can retrieve the desired files. The proxy server receives the request from the browser in the form of a URL. The proxy server retrieves the requested information, converts it to HTML format (if it is not already HTML) and sends it on to the browser behind the firewall. The proxy server handles all network requests and it is the machine that is directly connected to the WWW.

3.6.2 Caching in Proxy Server

Normally, all browsers in a local network access the same proxy server. When a browser requests a document, the proxy server retrieves the document from the WWW and sends it to the browser. The document may or may not be cached

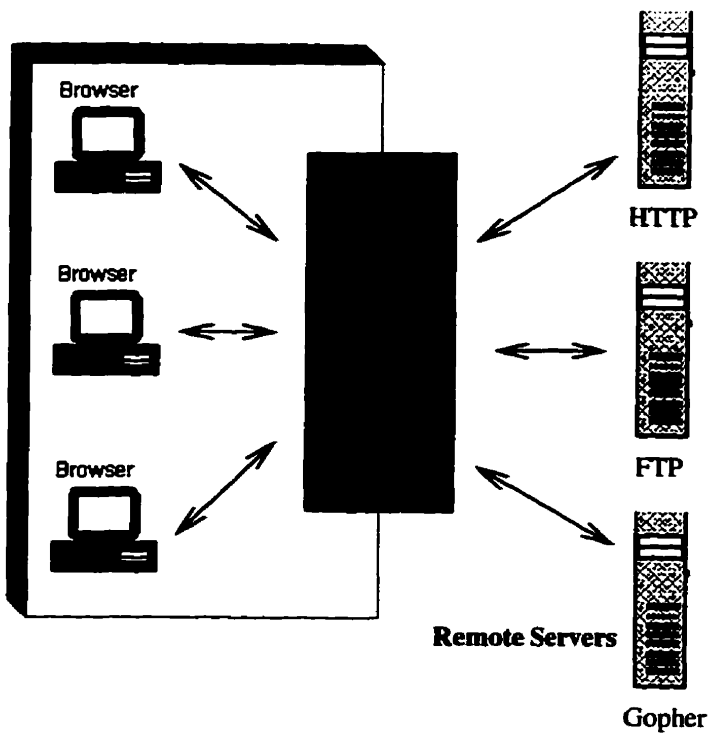


Figure 3.7: Proxy server on a firewall

at the proxy server based on the proxy server setup and the user requirements. If the documents are not cached at the proxy server, and other browsers request the same document, the proxy server retrieves the document from the WWW for each request. If the document is cached by the proxy server, then the proxy server will return the document from its cache after the initial request for that document has been processed. The documents to be cached and the expiry time for those documents are set based on the information provided by the client. Caching a document that a large number of users may access frequently can save network cost and connection time. Furthermore, caching can minimize the use of disk space because only one copy of the document is stored. Proxy servers can be particularly useful when interactive applications have to be executed using information normally found in the WWW. If both client and the proxy server are on the same machine, and the system is configured to use only the local cache, interactive applications can be executed without establishing a WWW connection.

Although caching documents has advantages, it also has some disadvantages. Many documents in the WWW are "living" documents. Determining when such a document might be updated or made obsolete can be a difficult task. A document could remain stable for a very long time and then suddenly change. Conversely, a document could change on a daily basis or even more frequently. This means that an arbitrary decision has to be taken about the expiry time of the documents. Normally, a proxy server performs a conditional check on the cached documents to decide on the validity of the document. The conditions are decided using the information provided by the server and various system parameters.

3.6.3 WWW Transaction without a Proxy

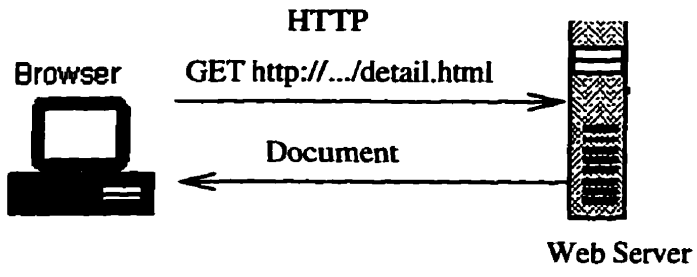


Figure 3.8: Web transaction without Proxy

Figure 3.8 shows the Web transaction when a browser (client) sends a document request to the WWW server without a proxy server. Clients in this situation have a direct connection to a WWW server. A user places a request,

http://www.cs.umanitoba.ca/~user/thesis/detail.html

The client converts the request into

GET /~user/thesis/detail.html HTTP/1.0

and a series of parameters specified by HTTP. The browser sends the request to the server that corresponds to *www.cs.umanitoba.ca* and waits for a response from the server. The request specifies the path (directory tree) where the document (detail.html) can be located in the server. The response is either an HTML document or an error message.

3.6.4 WWW Transaction Through a Proxy

When a browser (client) sends a request through a proxy, it always uses HTTP for the transactions with the proxy server. This is true even when the user wants to access a remote server that uses another protocol (say, FTP). A proxy server acts both as a server and a client. It acts as a server when it accepts requests from a browser and acts as a client when it requests a document from a WWW server. Similarly, even when the client specifies a document on an FTP server on the WWW, the proxy server retrieves the file from the remote FTP server and sends it to the client using HTTP. A Web transaction through a proxy server is shown in Figure 3.9.

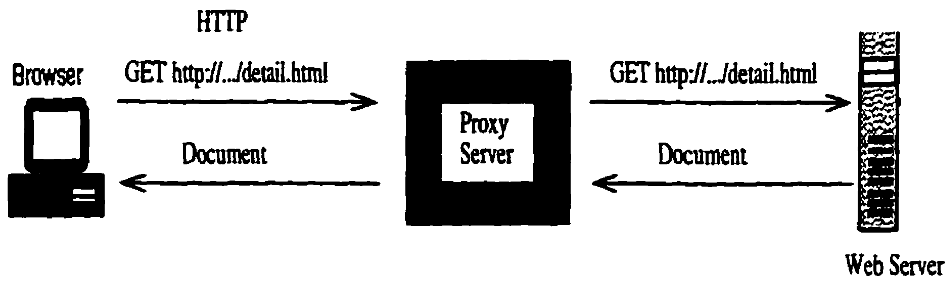


Figure 3.9: Web transaction through Proxy

A user places a request, for example:

http://www.cs.umanitoba.ca/~user/thesis/detail.html

The client converts the request into

GET http://www.cs.umanitoba.ca/~user/thesis/detail.html HTTP/1.0

and a set of parameters specified by the HTTP and sends it to the proxy server. The proxy server sends the request to the WWW server that corresponds to *www.cs.umanitoba.ca* and waits for a response.

3.6.5 WWW Transaction through Proxy Cache

Proxy servers may cache documents in their local memory. When browsers (clients) request locally cached documents, the proxy server returns the document from the cache instead of getting it from the WWW. The operation of a document request through a proxy server that caches documents is shown in Figure 3.10.

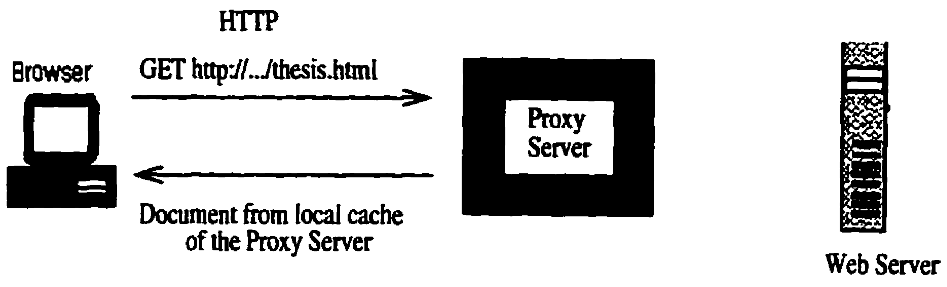

HTTP
Browser
GET http://../thesis.html
Proxy Server
Document from local cache of the Proxy Server
Web Server

Figure 3.10: Web transaction through Proxy Cache

A user places a request, for example:

http://www.cs.umanitoba.ca/~user/thesis/detail.html

The client converts the request into

GET http://www.cs.umanitoba.ca/~user/thesis/detail.html HTTP/1.0

and a set of header information specified by the HTTP and sends it to the proxy server. The proxy server searches its local cache for the document. If the document is available in its cache, the proxy server returns the document to the browser. If the document is not available in the cache, then the proxy server retrieves the document from the WWW, caches it locally and sends it to the browser. Subsequent requests for the same document retrieve it from the local cache of the proxy server.

When the document in the proxy cache is outdated, the proxy server retrieves the document from the respective WWW server. If the WWW server fails or if the transaction was unsuccessful, the document in the proxy cache is displayed with a warning that it is a previously cached copy.

Chapter 4

WWW in DSVM

DSVM functions as a transparent layer between the WWW servers and the browsers. Retrieved documents are stored in the SVAS of DSVM and subsequent requests for documents (even by other browsers) retrieve them from the shared memory. This can speed up document access, reduce network load, and alleviate server load by avoiding reconnections to servers for each browser.

4.1 General Architecture

Figure 4.1 represents the general system architecture of the proposed WWW in DSVM system. The system consists of a **Domain O** (Domain Outside the DSVM) and a **Domain I** (Domain Inside the DSVM).

To the left side of the dashed line is a representation of the current WWW environment. To the right side of the dashed line is the proposed architecture based on DSVM. The communication layer of the browsers in Domain I (*DI*) are modified to send the document request to DSVM. The SVAS is distributed between all the browsers in DSVM. Every browser that is invoked forms a node in DSVM and manages its own address space. All documents retrieved by one browser are visible to

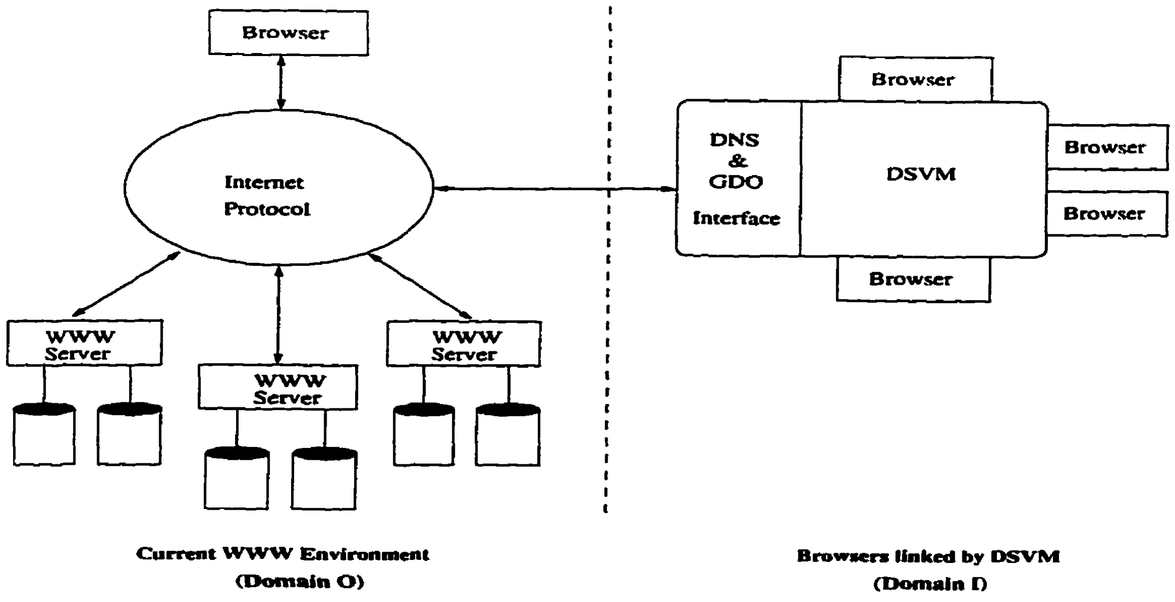


Figure 4.1: WWW in DSVM : Architecture

all other browsers connected to DSVM. Browsers can have their own local cache and their caching policies need not be modified to minimize changes and support greater transparency.

The Domain Name Service (DNS) & GDO interface provides the integration of DSVM with the WWW. This interface receives requests from DSVM, retrieves documents from the WWW based on client requests and sends documents back to DSVM. The interface uses DNS and standard protocols (HTTP, FTP, Gopher) for retrieving documents. Thus, DSVM integrates with the WWW with no modification to the current WWW setup. The DSVM operations and the shared memory management are completely transparent to the user. Browsers receive documents from either DSVM or the WWW.

4.2 DSVM Participation and Scalability

In this model, the browsers function as nodes in the DSVM and a group of browsers form a DSVM-cell (intranet). Browsers in one local network form an intranet and another group of browsers in another network form another intranet. The address space is partitioned and distributed across multiple intranets. The intranet formation can be explained by the following example.

Consider a scenario where there are various departments in an institution. A local network manages the invocation of browsers within its domain (say, the computer science department). Similarly, other local networks (say, the electrical engineering and sociology departments) manage browsers invoked inside their network. Each local network forms an intranet (DSVM-cell) in this model. Eventually, linking all departments in an institution would provide a scalable and efficient system for sharing documents as shown in Figure 4.2a.

When collaboration is conducted between more than one institution, similar sets of documents may be required at various locations. This model scales to larger collaborations and documents can be shared across institutions through the Internet (see Figure 4.2b). In this case, the GDO can be fragmented across multiple institutions. Each institution forms an intranet (DSVM-cell) with respect to the other institutions. Each intranet can be managed locally at each institution. This may introduce issues such as network load on the Internet that must be addressed. To minimize the problems, institutions that are located in the same geographical area and those requiring similar documents can be linked to share documents.

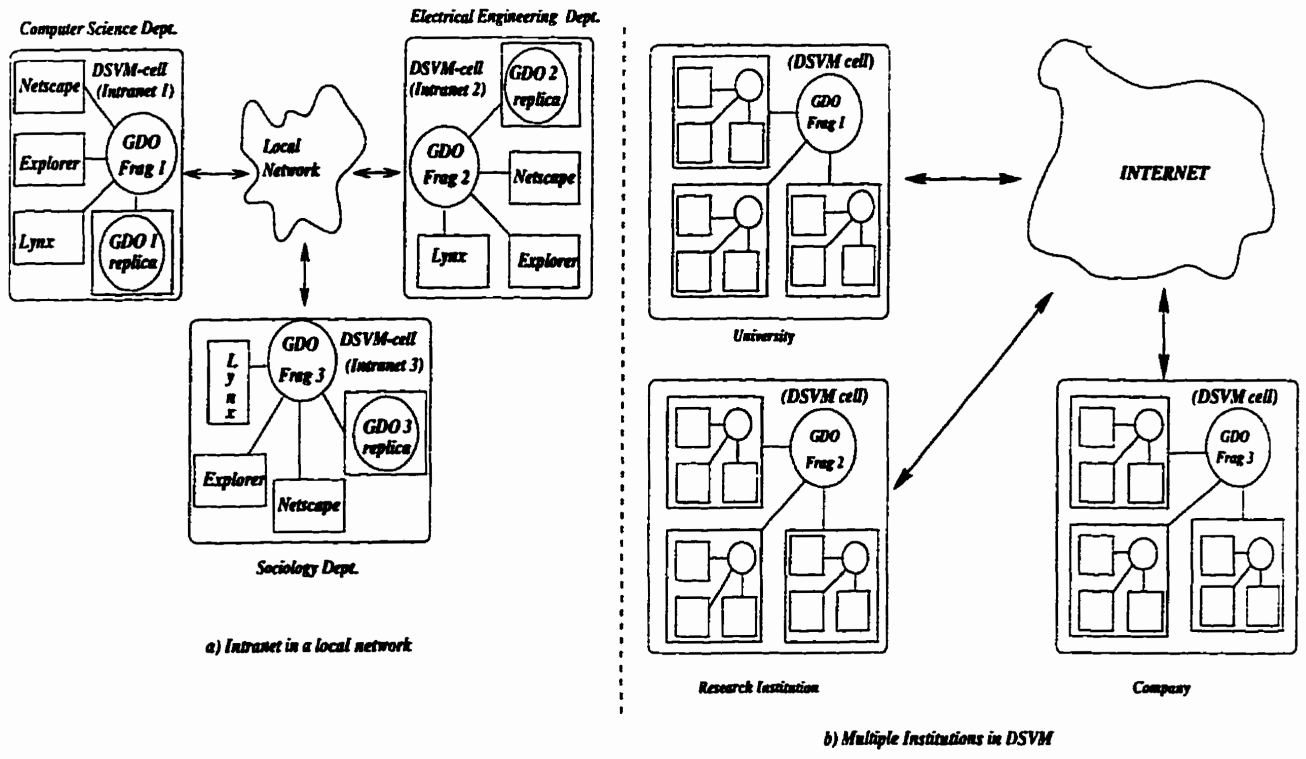
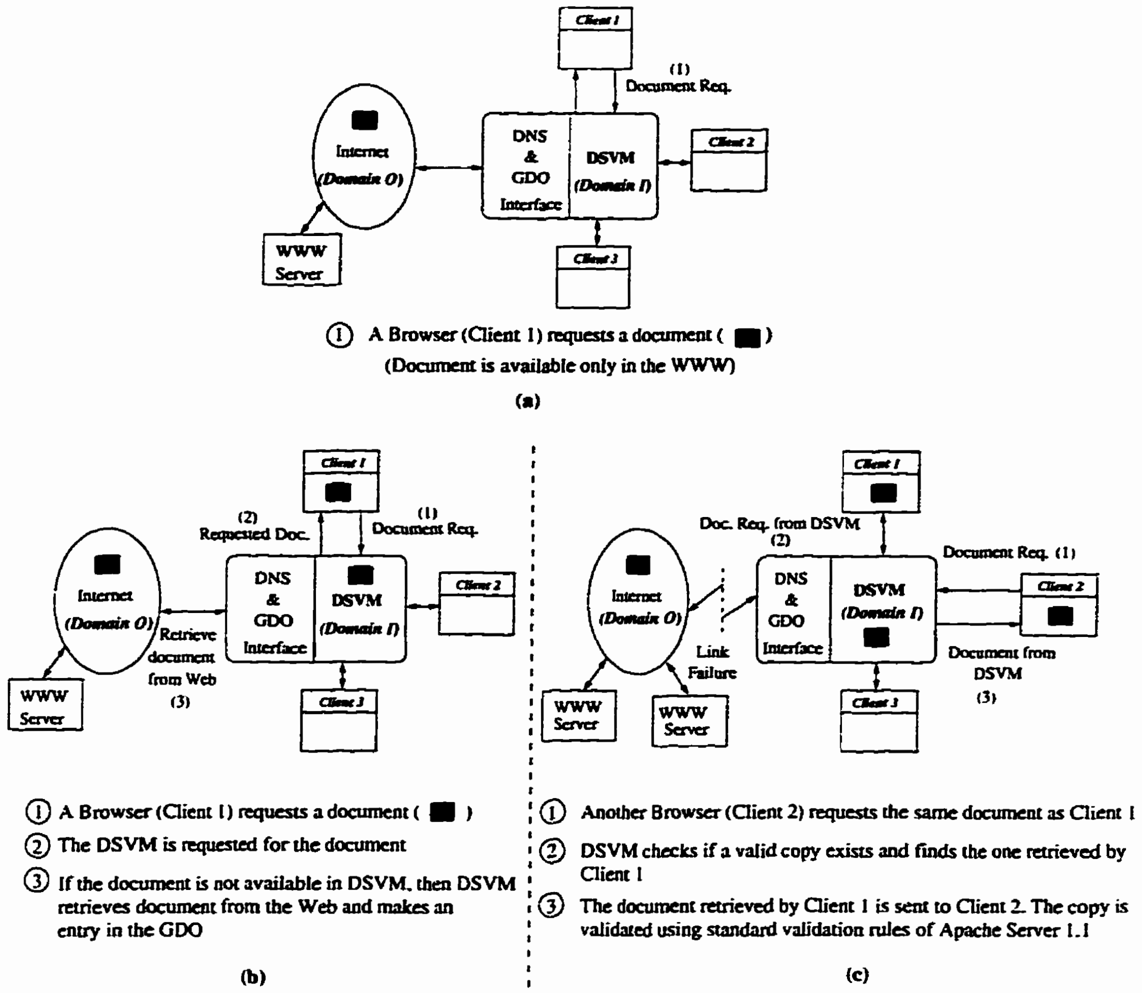


Figure 4.2: Intranet formation

4.3 Operational Stages

Figure 4.3a shows an example operational setup during system startup. The setup consists of three browsers (Client 1, Client 2, Client 3) connected to DSVM. DSVM does not initially contain any documents in the shared memory. Client 1 sends a document request to DSVM. The requested document is available in the respective WWW server in Domain O (DO) only. The DSVM checks the GDO for an entry that corresponds to the requested URL. Initially, since the document has not been retrieved by any other client, DSVM will not find an entry in the GDO. The request is then passed to the DNS & GDO interface which retrieves the document from the respective WWW server using standard protocols. The document is sent to the client and the GDO is updated with the document details. Figure 4.3b shows the model



(a)
① A Browser (Client 1) requests a document (■)
(Document is available only in the WWW)

(b)
① A Browser (Client 1) requests a document (■)
② The DSVM is requested for the document
③ If the document is not available in DSVM, then DSVM retrieves document from the Web and makes an entry in the GDO

(c)
① Another Browser (Client 2) requests the same document as Client 1
② DSVM checks if a valid copy exists and finds the one retrieved by Client 1
③ The document retrieved by Client 1 is sent to Client 2. The copy is validated using standard validation rules of Apache Server 1.1

Figure 4.3: Operational stages

setup after the request has been processed by DSVM. The document copy inside Client 1 shown in Figure 4.3b represents the browsers copy in its local cache after the document is sent by DSVM.

An advantage of having DSVM during a server failure is shown in Figure 4.3c. The WWW server in DO is inaccessible. Another browser (eg., Client 2) sends a request for the same document. The DSVM checks the GDO for an entry that corresponds to the requested URL. An entry is found in the GDO as the document had been retrieved previously by Client 1. The entry is validated using the standard validation rules of the Apache Server [1]. The model uses the validation rules of the Apache Server [1] since the rules have been commonly accepted by the WWW community and cache replacement policies are not a focus of this thesis. If the document is valid, DSVM returns the document from the shared memory. If the document in DSVM is outdated and the communication link is down, then the system returns the document in DSVM with a warning that it is a previously cached copy. The browsers may have their own cache maintenance and validation mechanisms. DSVM does not interfere with the validation mechanisms of the browser.

4.4 Design Issues

The documents in the WWW are considered to be objects in this model. Each URL is mapped to a virtual address (OID) and all operations like access, refresh, deletion, and relocation of documents are methods on the objects. The data structure in this model must provide fast searching and minimal creation overhead. The GDO is an index structure based on a B+ tree that provides efficient insertion and searching of objects. Each entry in the GDO contains mapping information between the OID (virtual address) and the URL, the address range of the document in the address space, the status of the document (deleted or non-deleted), the latest retrieval time, the update frequency of the document, header details of the protocol, and details of the user who retrieved the document from the WWW. Storing user details can also be used for security purposes to trace the users accessing a particular document.

A document retrieved from the WWW and stored in DSVM can become outdated. Validating a document can be done in many ways. Ideally, the system validates a document based on the information provided by the WWW server (like last modified date and document expiry date). Although the validation strategy largely depends on the information sent by the WWW server, the user can also set an expiry time for each document in the browser's memory cache.

Browsers may store documents in their local cache. Browsers may not send requests to DSVM for documents that are locally cached in the browser (depending on the browser setup). Instead, the browser displays the cached document. If the document is outdated in the browsers cache or if the document is paged out of the cache, then the browser sends a document request to DSVM. The parameters set in the browser (say, "retrieve documents everytime" option in Netscape) results in the validation of the DSVM copy every time the document is requested. A document that is outdated in DSVM will always be outdated in the local cache of a browser. This is because DSVM uses the information sent by the client and the server to validate an entry. The converse, however, is not true. If a document is outdated in the local cache of a browser, it need not necessarily be outdated in DSVM. Another client may have requested the same document and DSVM may have retrieved the latest copy from the WWW. When a document in a browser's cache is outdated, it is guaranteed that the DSVM copy is either the same or a more recent version. If the document in DSVM is the same copy as that of the browser, then the document is retrieved from the WWW and sent to the user. The advantage of this approach is that if the document is valid in DSVM (ie., the latest copy has been retrieved by another browser), then it is sent to the browser from the shared memory without making an Internet connection. This is because the validation rules of a document are based on the latest retrieval date and time.

Documents that are retrieved and stored in the shared memory may not be accessed repeatedly without being refreshed. Documents in the shared memory can be refreshed either automatically or whenever the documents are outdated. Documents can be automatically refreshed by having a process that retrieves the documents from the WWW after a particular time frame and updates the shared memory. In our model, validating and refreshing a document is done when the document is requested by a browser rather than automatically refreshing the documents. If a document copy in the shared memory is outdated and the corresponding WWW server is down, then the system displays the copy of the document in the shared memory with a warning that the document is a previously cached copy.

When a WWW server or all the links between the WWW server and a client fail, and if the server documents have been retrieved previously in DSVM, subsequent access to the documents by clients will retrieve them from the shared memory. Even if the document in the shared memory is outdated, the same document can be displayed to the user with a warning that the document is outdated and the server link has failed. Thus, when a WWW server fails, the user will get a relevant document. The naming scheme of the WWW will remain the same as that of the conventional URL scheme with the DSVM as a transparent layer for querying the WWW. The advantage is that users need not modify URLs to access the DSVM.

Scalability is another issue to consider when designing a system in the WWW. Workstations based on 64-bit architectures provide a very large virtual address space that can accommodate many WWW documents. For example, at the rate of 100MB/sec, it would take over 5000 years to consume the address space. Furthermore, with the likely availability of 128-bit architectures in the near future, such address spaces will be sufficient to accommodate the expansion of the WWW.

Performance of the model while adding a client depends on the scalability of the DSVM. When WWW clients are added to (or removed from) the DSVM, only simple changes to the underlying structures are required [19]. As the GDO is partitioned and due to the load balancing features of the DSVM, the system will scale well as new nodes are added.

A document may be relocated from one URL to another in the WWW with a forwarding pointer to the new location. After a document is retrieved from the WWW, if the document is relocated, then subsequent requests for the document retrieve it from the virtual memory without traversing the surrogate links. If the document in the memory is outdated, heuristics can be used for automatic traversal of surrogate links to retrieve the relevant document from the WWW. One method for automatic traversal of surrogate links is to scan the incoming document for phrases like "*document relocated*". If the system locates such a phrase, then the document is scanned for a hypertext link after the phrase and a request is sent to the corresponding server for the new document. Heuristics for improving access to relocated documents is not a focus for this thesis. Using DSVM, however, any results in this area can be shared among participating browsers so that they do not have to go through the same procedures.

If a document requested by a client has been deleted on a WWW server and the document had been retrieved by a client in DSVM before the deletion, then requests for the document retrieve it from the shared memory until it becomes invalid. When the document in the shared memory is outdated, subsequent access to the document changes the status of the corresponding entry in the GDO to 'document deleted'. Subsequent access to the same document will retrieve the status information from the GDO and thus, deletion of a document can be known prior to accessing the WWW server. The entry for a deleted document will have the default values for validating the entry, thus enabling the user to verify if the document is re-created in the WWW. Furthermore, a user may find that a document is deleted after traversing a series of "redirecting" hypertext links (the broken-link problem). The link traversal information and the document status (ie., deleted) can also be shared by other users trying to access the same document. Thus, traversal of surrogate links is avoided and the user will know the status of a document without making a series of Internet connections.

Chapter 5

Proof of concept prototype

The complete implementation of the logical design of WWW in DSVM depends on the implementation of DSVM, which is ongoing. A proof of concept DSVM prototype [7] has been developed using Treadmarks [8] as the shared memory manager and Exodus [9] as the persistent storage manager. The prototype WWW in DSVM system builds on this and manages WWW documents as objects in DSVM. GDO entries contain the URL, the latest retrieval date, update frequency of the document, status of the document, and HTTP header details.

Figure 5.1 shows the prototype architecture of WWW in DSVM. To provide interoperablity for multiple browsers, and to avoid being browser specific, proxy servers are used to provide an interface between DSVM and browsers in the prototype. A proxy server, apart from functioning as an interface between browsers and DSVM, also performs the duties of the DNS & GDO interface. A proxy server receives the document requests from browsers and sends them to the DSVM. If the document is not found in the shared memory, then the proxy server retrieves the document from the WWW server and sends it to the browsers and DSVM. Linking a browser to a proxy server involves only setting the proxy server option of the browser to a respective proxy server. All commonly available browsers can link to a proxy server so the

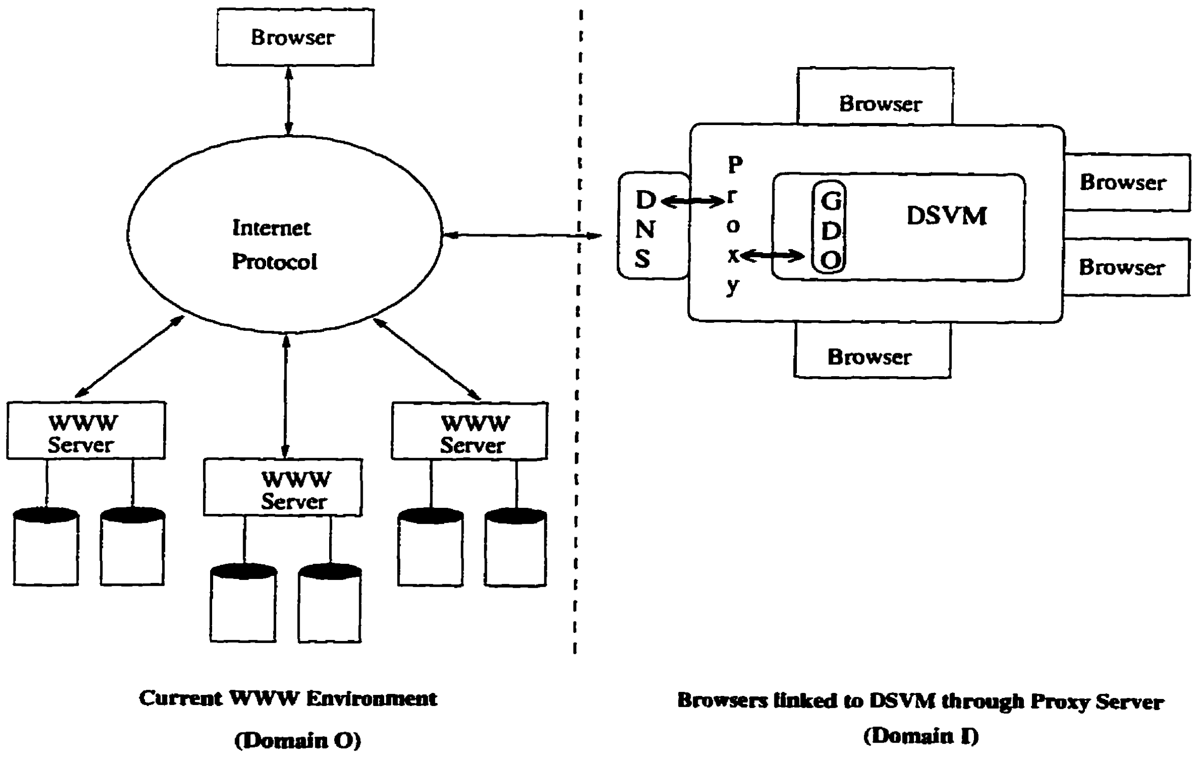


Figure 5.1: Prototype Implementation

prototype supports a heterogeneous set of browsers. The proxy servers interact with the shared memory manager (ie., Treadmarks) for managing documents in the shared memory.

Figure 5.2 shows a simple prototype setup demonstration in our Lab. The prototype consists of 1) Proxy Servers, 2) Treadmarks servers and 3) the DNS & GDO interface. Treadmarks (TM) is used to provide shared memory between workstations. Since TM does not support dynamic process management, it is installed among the participating workstations (eg., *copper*, *calcium* and *barium*) during startup. These workstations correspond to the nodes in DSVM system and manage the documents in the shared memory. All TM servers are iterative servers that service only one request at any instance. All requests are serviced in a FIFO fashion by the TM server. There are two types of requests that can arrive from proxies. Proxies may either request

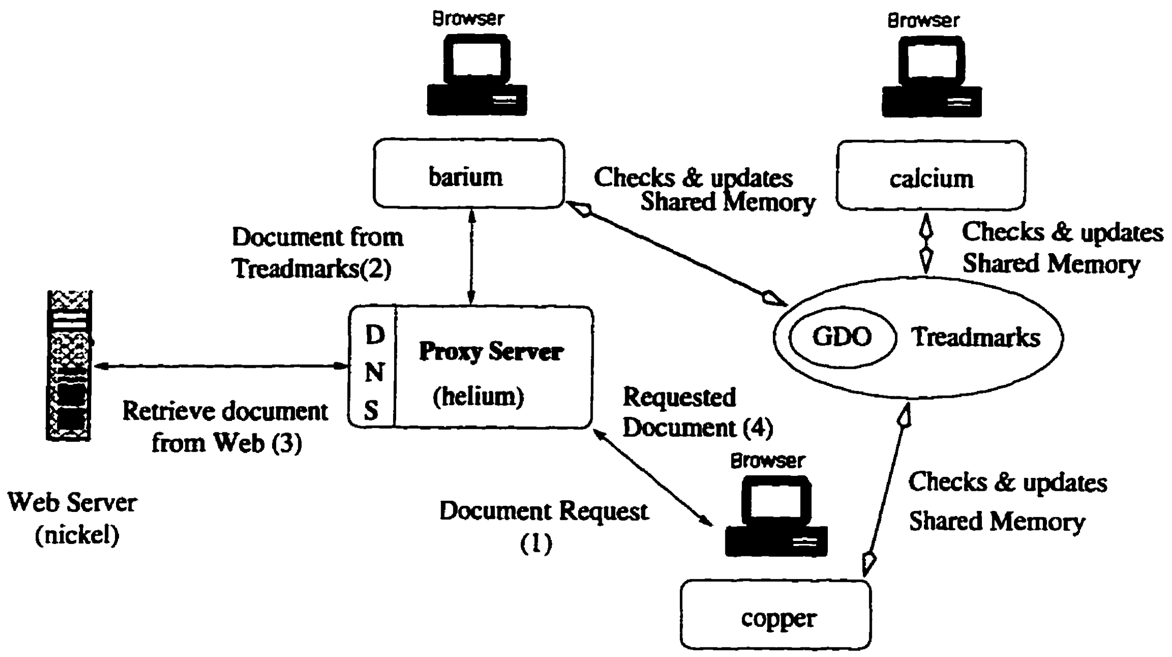

Browser
Browser
barium
Checks & updates Shared Memory
calcium
Checks & updates Shared Memory
Document from Treadmarks(2)
D N S
Proxy Server (helium)
GDO
Treadmarks
Retrieve document from Web (3)
Requested Document (4)
Browser
Checks & updates Shared Memory
Web Server (nickel)
Document Request (1)
copper

Figure 5.2: Prototype Setup

an object from DSVM or may send an object to be updated in the shared memory. The shared memory is checked for an object based on the request or is updated with an object as sent. Synchronization primitives (locks, barriers) are used to maintain consistency. Requests from proxy servers can be sent to any TM server. The TM servers all have similar processing characteristics in our prototype and we attempt to distribute the workload evenly by randomly selecting the TM server where the request is sent. This is a simple approach that scales well and allows redistribution of workload by redistributing the probability of a server being selected. A more elaborate load balancing approach can be incorporated and is a topic of future research and fine tuning of the system.

The browsers are linked to DSVM through proxy servers (in the example setup, there is only one proxy, the machine *helium*) that waits for requests from browsers. The DNS & GDO interface is integrated with the proxy servers to retrieve documents from the WWW. A proxy server is a multi-threaded server that invokes a thread for each request from a browser. Multiple proxy servers wait for browser requests and there can be any number of nodes linked by DSVM to manage shared documents. This creates the shared distributed environment shown in the logical design of Figure 4.1 where multiple browsers are linked to the shared memory.

5.1 Software Module Relationships

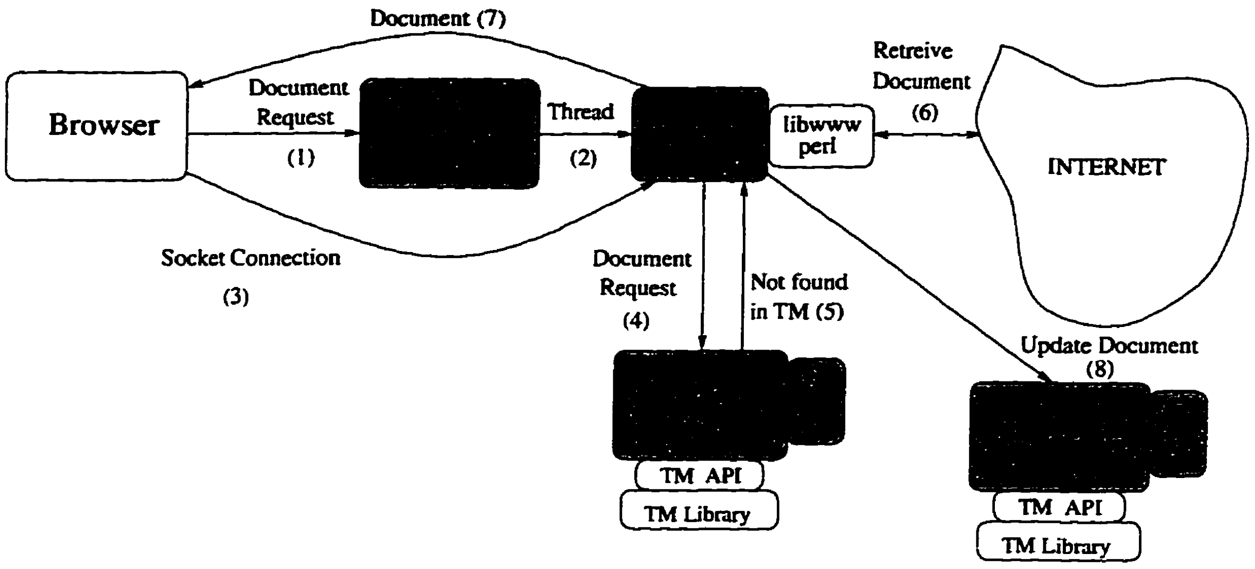


Figure 5.3: Software Modules Relationship

Figure 5.3 shows the relationship between the software modules of the prototype. The grayed components represent the modules written to develop the prototype. The other components are existing software and systems that interface with the prototype. The prototype uses the commercially available Treadmarks software package and its Application Programming Interface (API) for providing shared memory between workstations. Portions of the API used for this prototype are described in Section 3.4. The prototype also uses the LWP (Library for WWW in Perl) library of Perl for interfacing with the WWW and to retrieve documents from the WWW. The proxy server is coded in Perl and provides the interface between browsers, TM servers, and the WWW. The interface between the browsers and the proxy server is provided by receiving requests from the browsers and assigning a thread process to manage each request. The interface between the proxy server and the WWW is integrated in the proxy server and uses the LWP library to retrieve documents from the WWW. The proxy server uses socket communication to interface with the TM servers for providing shared memory services. The data structure manipulation in the shared memory is integrated in the TM server which reads and updates the GDO structure using various TM functions. The following table gives a list of the software modules that were available and the modules that are written to develop the prototype.

Software Modules	Available	Written
Browsers	X	
Treadmarks Package	X	
WWW Interface		X (Using LWP library in Perl)
Proxy Server		X
TM server		X
GDO Interface		X

5.2 Execution and Event Trace of the Prototype

Figure 5.4 shows the event trace of the prototype. A user enters a request for a document. If the browser finds a valid document copy in its local cache, it is returned without any involvement by DSVM. If the document is not cached by the browser, or if the copy is invalid, then the browser sends the request to a proxy server. The proxy server invokes a thread to service the request and continues to listen for other requests. The thread sends the request to any of the TM servers. The selected TM

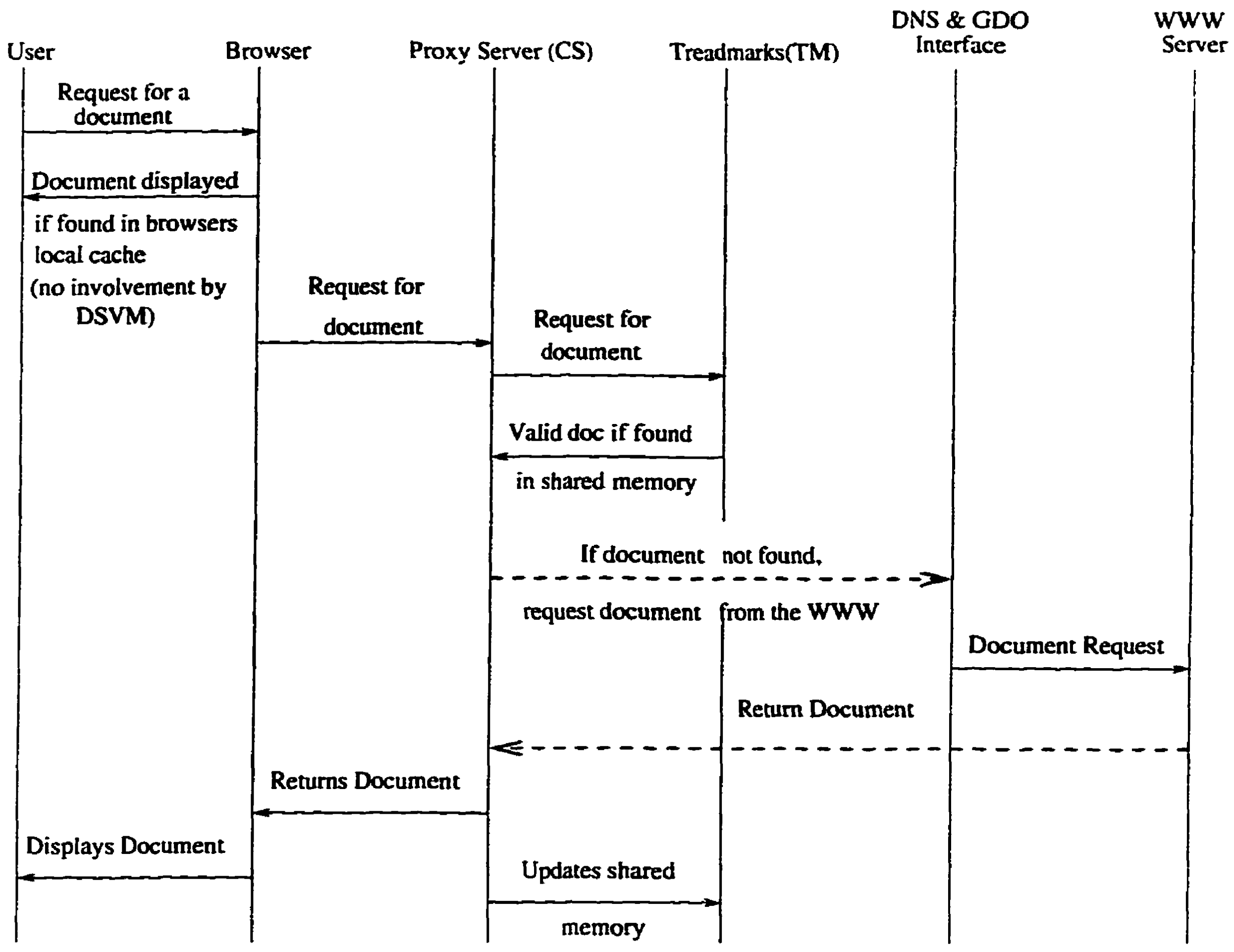


Figure 5.4: Event Trace Diagram

server checks the shared memory for a valid copy of the document. If a valid copy is found in the shared memory, then the TM server sends the document to the thread, which in turn sends it to the browser for display. If the document is not found in the shared memory, or if the document copy in DSVM is outdated, then the thread requests the DNS & GDO interface to retrieve the document from the WWW server. After receiving the document from the interface, the thread sends the document to the browser for display and then selects (a possibly different) TM server for updating the shared memory with the retrieved document. The update instruction sent to the TM server consists of a line containing only "UPDATE", followed by the document header and the document.

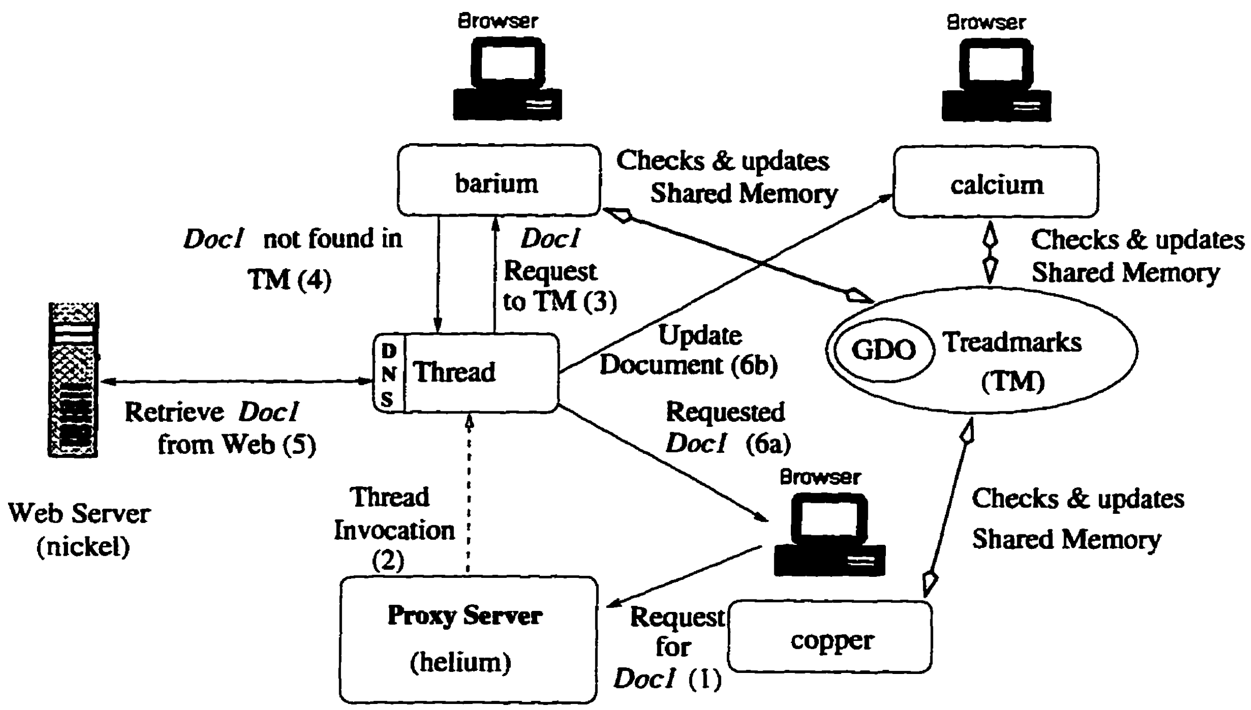


Figure 5.5: Prototype Execution : Stage 1

Figure 5.5 shows the first stage of the prototype execution. Domain O is attached by an HTTP server (*nickel*) that waits for requests on port 8080. Browsers can be invoked at any workstation in the network. The proxy server option of the browsers is set to "point at" the proxy server (*helium*) at port 8080.

Step 1 User requests a document by entering the URL of the document (say, at *copper* for *Doc*1). The browser sends a request in the following format to the proxy server (*helium*).

*GET Doc*1 *HTTP/1.0*

Referer: http://www.umanitoba.ca/dept.html

Proxy-Connection: Keep-Alive

User-Agent: Mozilla/3.0 (X11; I; SunOS 5.5 sun4u)

Host: www.cs.umanitoba.ca

*Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */**

Line containing only CRLF

Step 2 The proxy server receives the request, creates a thread to service it, and continues to listen for other requests. The thread process manages all operations on behalf of the request.

Step 3 The thread sends the request to one of the TM servers (eg., *barium*) by randomly selecting an entry from the configuration file *.Tmkrc*.

Step 4 The TM servers are modeled as iterative servers and service the requests in a FIFO (First In First Out) fashion. The TM server receiving the request checks to see if it is a GET instruction or an update instruction. If the request is a GET instruction, the TM server checks the shared memory for the document by searching through the GDO. Initially, since the document has not been retrieved previously, the TM server does not find the document in the shared memory and replies with a "NOT FOUND" message.

Step 5 The thread receives the "NOT FOUND" reply from the TM server and sends the document request to the respective WWW server. The DNS & GDO interface to the WWW is integrated with the proxy server to avoid multiple requests manipulation at the interface level. The WWW server responds either with the document or with an error code.

Step 6 The thread sends the response (error code or the document) to the browser and also to a selected TM server (eg., *calcium*). The update message to the TM server includes a line containing "UPDATE" followed by the document headers and the document itself. The TM server that receives the information checks the shared memory for the URL. If the document entry is found (this means that the document was outdated in memory), the TM server updates the entry with the latest content. If the document entry is not found (ie., the document is retrieved for the first time), then the document is stored in the shared memory (GDO) as a new document. All subsequent requests for this document from any browser will retrieve the document from the shared memory until the document becomes outdated (see validation rules in Section 5.3).

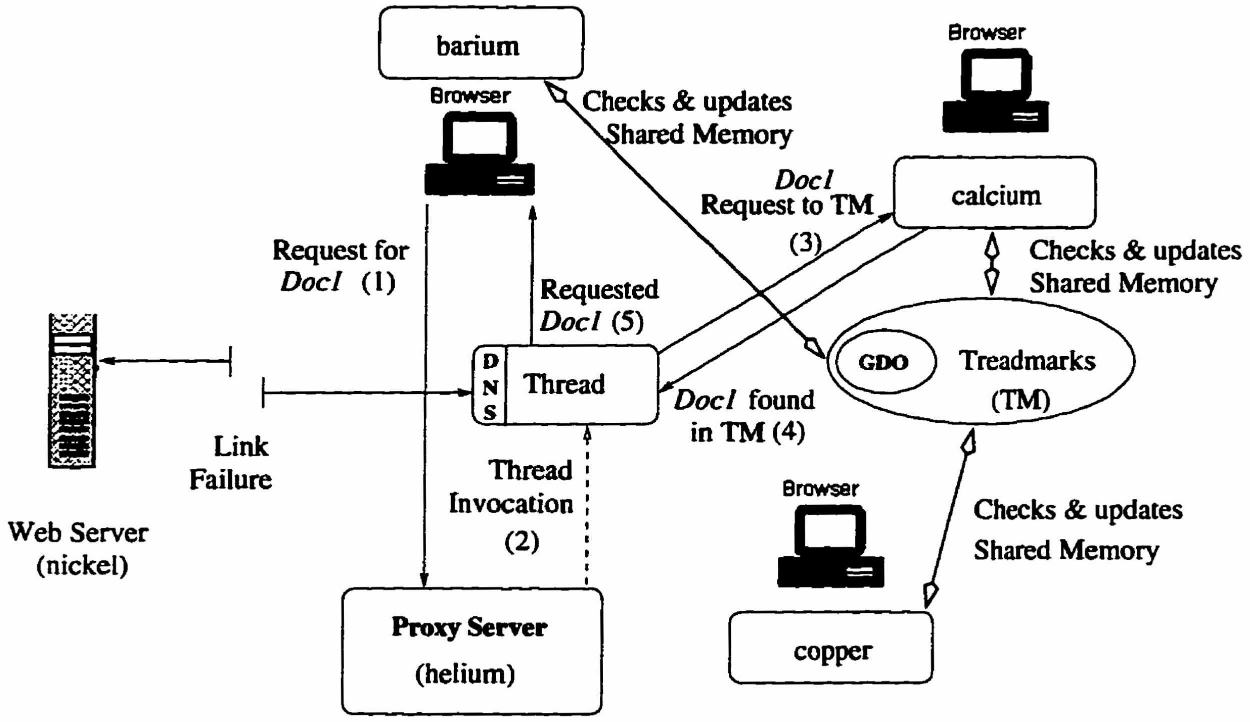


Figure 5.6: Prototype Execution : Stage 2

In stage 2 (see Figure 5.6), the communication link with the WWW server (*nickel*) is down. In such a case, no documents at the server can be retrieved until the link is re-established. Another client (eg., *barium*) requests the same document (*Doc*1) during the link failure. A separate thread is invoked that manages all the operations of this request. The thread sends the request to a TM server (eg., *calcium*). The TM server finds the document that was retrieved previously by *copper* (Figure 5.5) in the shared memory. The document is validated using the Apache Server validation rules (see next section). If the document is not outdated, the TM server returns the document to the thread, which in turn sends it to the browser. If the document is outdated in the shared memory and the Web server is unreachable, then the thread retrieves the document from the TM server and sends the document to the browser with a warning that the displayed document is a previously cached copy.

5.3 Validation Rules

The validation techniques used in this thesis are derived from the validation rules of the Apache Server mainly because the rules work well and are commonly accepted. Documents that are cached in the shared memory can become stale. The system must verify the validity of the cached document before sending the document to the browser. There are many methods of validating a document. One such method is to verify the cached copy with the original document from the WWW. However, it is not practical to check each of the cached documents with the original documents in the WWW before sending them to the client. A practical method of validating a cached document is to use some standard validation rules. These validation rules differ between systems. However, nearly all systems use the information provided by the WWW server with the document to validate that document.

5.3.1 The Apache Server Rules

The following are some of the Apache Server validation criteria that are used in this thesis. Some of the rules are extended and modified to suit the goals of this thesis.

1. *CacheLastModifiedFactor <factor>* : If the originating WWW server did not supply an expiry date for the document, then estimate one using the formula

   expiry-period = time-since-last-modification $*$ <factor>

   For example, if the document was last modified 10 hours ago, and <factor> is 0.1, then the expiry period will be set to 10*0.1 = 1 hour.

2. *CacheMaxExpire <time>* : Cachable documents will be retained for at most <time> hours without checking the originating server. Thus, documents can be at most <time> hours out of date. This restriction is enforced even if an expiry date was supplied with the document.

3. *CacheDefaultExpire <time>* : If the document is fetched via a protocol that does not support expiry times, then use <time> hours as the expiry time.

4. *NoCache <word/host/domain list>* : The NoCache option specifies a list of words, hosts and/or domains, separated by spaces. HTTP and non-password FTP documents from matched words, hosts or domains are not cached by the proxy server. The proxy module will also attempt to determine IP addresses of list items, which may be host names during startup, and cache them for match test as well.

5.3.2 WWW in DSVM Validation rules

The WWW in DSVM validation rules use the header information provided by the WWW server to validate a document in the shared memory. Some of the header details provided by the WWW server that are used in the validation rules are the *Refresh* : T *seconds*, *expiry_date*, and *last_modified date* header lines. The validation rules given below are checked in the given sequence to validate a document in the shared memory. If at least one rule returns true, then the system returns the document in the shared memory to the browser.

1. *Pragma* : *no-cache* : A document is not cached in the shared memory if the proxy server gets instruction from the browser or the WWW server that the document should not be cached (by using *Pragma* : *no-cache*). Using the *Pragma* option, the client specifies that the document should not be cached because it is a dynamically changing document. Thus, the proxy server retrieves the document from the WWW everytime the document is requested.

   ```
   if (pragma line in Document header)
       Always return document outdated
   ```

2. *Refresh* : T *seconds* : This validation rule is not based on the Apache Server validation. This validation is based on browsers' ability to refresh documents automatically. The value T is the *document refresh period*. The valid time frame of a document is decided by the information sent by the WWW server in the header information. The WWW server may specify that a document gets updated every T seconds and has to be automatically refreshed once in T seconds. This is specified by the line (*Refresh* : T *seconds*) in the document header by the WWW server that sends the document. In such a case, the browser automatically sends a request for the same document every T seconds.

   ```
   if (document refresh period is available in document header)
     if (current_time - document_retrieval_date) > document refresh period)
         Return document outdated
     else
         Return document copy is valid
   ```

Caching this type of a document and sharing the information with other browsers can result in reduced Internet server connections. For example, if a browser retrieves a document that has a refresh period of T seconds and another browser requests the same document within the T seconds, the latter browser will receive the document from the shared memory rather than from the Web.

Suppose a document is retrieved from the WWW with refresh time $T = 120$ and the document was retrieved at 11:00am. The document is valid in DSVM until 11:02am. Suppose another browser requests the same document at time 11:01am. If the DSVM sends the document with the refresh time as 120 seconds (ie., without any modification to the header information sent by the WWW server), then the document in the second browser will be valid until 11:03am, which is obviously incorrect. To maintain the life span of the document in all the browsers to the appropriate T seconds, the system modifies the refresh period of the document to the difference between the current time and the document expiry date. In the above example, when the second browser requests the document at 11:01am, the DSVM sends the document copy from the shared memory with the document refresh period as $T = 60$ (Document expiry time in DSVM (11:02am) - Document requested time (11:01am)). Thus, document copies in all the browsers become stale after T seconds from the time the document was retrieved from the Web. The modification of the refresh time is performed to synchronize the expiry time of the document copies that are cached in the browsers so that all browsers have the same expiry time for the document.

3. *expire-date* : If the above header information is not available, and the document has an expiry date, then the document in the shared memory is validated using the *expire-date* sent in the document header by the WWW server. Every time a document request is received, the expiry date of the document is checked

with the current time. The document is retrieved from the WWW if the copy has expired. If the document cannot be retrieved from the WWW, then the document in the shared memory is displayed to the user with a warning that the document is a previously cached copy.

```
if (expire_date line is present in document header)
  if (expire_date < current_date)
      return document outdated
   else
      return document copy valid
```

4. *last_modified date* : If the document header does not contain any of the above header fields, then validation of the cached copy is done by using the last_modified date of the document. The validation period of a document can be increased or decreased by either increasing or decreasing the value of the system setup parameter ¡factor¿. This is similar to the Apache Server rule 1.

```
if (last_modified date exists)
  time_hrs = current_date - last_modified date
  expiry_period = time_hrs * ¡factor¿;
  if ((document_retrieval_date + expiry_period) > current_date)
      return document outdated
  else
      return document copy valid
```

5. *Default Expiry Period : EXPIRY_PERIOD* : When the server sends incomplete header information omitting expiry_date, last_modified date, or refresh_period,

then the user can still set an expiry time for the document by setting the parameter EXPIRY_PERIOD to the required time.

```
if ((document_retrieval_date + EXPIRY_PERIOD) > current_date)
   return document outdated
else
   return document copy valid
```

5.3.3 Performance

To estimate the efficiency of the prototype, its performance can be compared with and without DSVM caching. An accurate formal analysis of the performance gain depends on many factors including network load, WWW server workload, and finalized implementation of DSVM. Moreover, the network load and the WWW server workload are time-variant factors. Due to the significant undertaking required for a formal analysis, only an *ad hoc* comparison of times taken for retrieving documents with and without shared memory was performed.

The prototype was tested by retrieving documents from sites that had high network load and performing an ad hoc comparison with the time taken for subsequent requests for the same document from other browsers connected by DSVM. The documents were displayed from DSVM and there was a noticeable reduction in time when DSVM was used. Advantages of the DSVM system during a WWW server failure were demonstrated using a local HTTP server (*nickel*). Documents from *nickel* were retrieved by a browser in one node and then the server was killed. Subsequent requests for the same server documents from other browsers in the network retrieved them from DSVM. The prototype has been installed on a number of workstations in a local network and has been demonstrated to link browsers sharing documents within the local network, in a departmental intranet, and across the Internet.

Chapter 6

Conclusions and Research Directions

A distributed WWW document sharing mechanism between browsers is presented in this thesis. A proof of concept prototype has been developed and illustrated [23]. The system (called *WWW in DSVM*) provides a distributed memeory cache between browsers that helps to reduce network load and server load, and also minimizes the impact of server and communication link failures on the retrieval of documents. An ad hoc comparison of times needed to retrieve documents with and without shared memory was performed and the tests show noticeable gains comensurate with the above goals. An accurate formal analysis of the performance gains depends on various factors like network load, WWW server workload, and final implementation of DSVM. Since this is a significant undertaking, a formal performance analysis was not a part of the thesis.

This model can be used in alleviating the number of hits on a frequently accessed WWW server which in turn alleviates the network load to that server. A typical example of this scenario is the Cable News Network (CNN) Web server. Following the landing of Pathfinder on Mars, the CNN server and related sites (eg., NASA) received many extra hits from people attempting to get information and pictures about the event. This situation represents a typical application of this model. Assume that there is a WWW in DSVM network connecting all the browsers in the University of Manitoba campus. If one user retrieves a document and images from the CNN server, all the other users in the University can retrieve the document locally from DSVM without connecting to the CNN server. If there were a large number of installed WWW in DSVM systems around the world, this would result in a reduced number of hits on the servers as well as reduced Internet network load.

Some of the other application environments where this model can be used to improve performance include : (1) Internet Service Providers (ISPs) cooperating to provide better service to a local customer base, (2) corporate intranets where employees in certain project cliques work with similar sets of WWW documents, and (3) education in Internet training where instructors direct hundreds of students in a classroom to access certain WWW pages to demonstrate certain functionality (hundreds of students requesting the same page simultaneously can cause detrimental server and network load). With WWW in DSVM, the documents are retrieved once from the Internet and the other browsers then retrieve the document locally from the DSVM.

In all the above application areas, performance of WWW in DSVM should offer improvements over the conventional WWW approach since the number of Internet connections are minimized, which minimizes the problems due to network load. By sharing documents across browsers, the model also alleviates problems due to link failures and Web server failures.

This system can be extended to integrate the WWW servers into DSVM. When WWW servers are integrated into DSVM, the system can use different communication protocols like RPC or IIOP for better performance inside DSVM. Requests from browsers linked by DSVM for documents residing in DSVM can also be serviced by using specialized protocols for improved performance. Communication with WWW servers outside DSVM can be done using standard Internet protocols.

Requests from browsers in DSVM can also be customized for read-only data (eg., image files) for better performance. When a document is retrieved from the WWW, one of the main reasons for time delay is the transfer of image files. An HTML document in DSVM may refer to many image files. Every time the document is requested, the referenced image files are also validated with the validation rules. If the image files in DSVM are invalid, then they are loaded from the WWW and this results in time delays. Users can be given an option at retrieval time to specify whether the image files in DSVM should be validated with the WWW. Based on the users' decision, the image files are either verified and loaded from the WWW or loaded directly from DSVM. This will reduce the network load and provide better document retrieval times for users who are interested in seeing the latest *textual* components of the documents.

Another possible extension of this project is to provide an indexed searching mechanism inside the DSVM. After a document is retrieved from the WWW, DSVM can store metadata about a document in the GDO entry along with other details. Metadata can be extracted by using the information provided by the server with the document headers. If the user wants to query DSVM documents for a particular keyword, then the metadata in the GDO entries can be searched for the keyword. The list of URLs that contain the keyword and its corresponding metadata can be returned to the user in the form of a dynamically generated HTML file. Users can then select the required documents from the list. Requests for documents that are valid in DSVM receive them directly from DSVM. Other documents are retrieved from the WWW. If the keyword cannot be found in the metadata, then the user can be consulted to perform a full text scan of documents in DSVM.

The definition of heuristics for automatic traversal of hypertext links to relocated documents is a potential extension to this thesis. The results of the heuristics can be shared among browsers to avoid repeated traversal of surrogate links. Currently, the documents in the shared memory are validated and refreshed only when the page is accessed. Document refreshing mechanisms can be extended to refresh documents automatically based on the system workload. The documents that are frequently accessed can be refreshed during low system workload to increase the efficiency of the system.

In summary, this thesis presents a model that applies distributed shared memory system concepts to the WWW. The main focus of the thesis is to reduce network loads, server loads, and minimize the problems of retrieving documents due to communication link failures and server failures. These goals are achieved by providing a distributed caching mechanism between browsers in a local network to share documents. This model reduces network traffic and server loads by reducing the number of Internet connections required to retrieve documents in certain shared environments. This model can be extended to integrate WWW servers into DSVM and an indexed searching mechanism can be provided to search documents inside DSVM for better performance.

Bibliography

[1] Apache Proxy Server. http://www.apache.org/.

[2] CERN Laboratory Home Page. http://wwwcn.cern.ch/pdp/ns/ben/TCPHIST.html.

[3] Comprehensive Perl Archive Network. http://www.perl.org/CPAN/.

[4] HyperText Markup Language. http://www.w3.org/pub/WWW/MarkUp/Wilbur/.

[5] HyperText Transfer Protocol. http://www.ics.uci.edu/pub/ietf/http/rfc1945.html.

[6] Persistent URL Home Page. http://purl.oclc.org.

[7] K. Barker, R. Peters, and P. Graham. Distributed Shared Virtual Memory for Interoperability of Heterogeneous Information Systems. In *OOPSLA Workshop on Interoperable Objects - Experiences and Issues*, October 1995.

[8] P. Keleher C. Amza, W. Zwaenepoel A.L. Cox, and R. Rajamony. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[9] M. Carey, D.J. Dewitt, and S.L. Vandenberg. A Data Model for EXODUS. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 413–423, September 1988.

[10] J.S. Chase, H.M. Levy, and M.J. Feeley. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer systems*, 12(4):271–307, November 1994.

[11] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. How to use a 64-bit Virtual Address Space. Technical Report UW-CSE-92-03-02, Dept. of Computer Science, The University of Washington, 1992.

[12] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. In *Fifth International World Wide Web Conference*, France, May 1996.

[13] P. Graham and K. Barker. Distributed Object Base Implementation Using a Single, Shared Address Space. *Proc. Mid-Continent Information Systems Conference*, pages 62–77, May 1993.

[14] P. Graham, K. Barker, S. Bhar, and M. Zapp. A Paged Distributed Shared Virtual Memory System Supporting Persistent Objects. Technical Report TR 92-07, The University of Manitoba, 1992.

[15] D. Ingham, S. Caughley, and M. Little. Fixing the Broken-link Problem: The W3Objects Approach. In *Fifth International WWW Conference*, France, May 1996.

[16] P. Keleher, Alan L.Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. Technical Report TR 92-07, Rice University, 1992.

[17] K. Kosuge and M. Morita. An Implementation of Management Information Exchange Using DNS for Replicated Information Resources on WWW. In *Proc. Japan WWW Conference'95*, Japan, November 1995.

[18] S. Lewontin and M.E. Zurko. The DCE Web Project: Providing Authorization and Other Distributed Services to the World Wide Web. In *Second International WWW Conference*, Chicago, U.S.A., October 1994.

[19] J. Mathew, P. Graham, and K. Barker. Object Directory Design for a Fully Distributed Persistent Object System. In *Proc. Object Oriented Database Systems Symposium of the Engineering Systems Design and Analysis Conference*, Montpellier, France, July 1996.

[20] M. Nabeshima. The Japan Cache Project: An Experiment on Domain Cache. In *Sixth International World Wide Web Conference*, Santa Clara, California, USA, April 1997.

[21] M.T. Öszu, R. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. TIGUKAT: A Uniform Behavioral Objectbase Management System. *The VLDB Journal*, 4(3):445–492, 1995.

[22] B. Ozden and A. Silberschatz. The Shared Virtual Address Space Model. Technical Report TR-92-37, The University of Washington, 1992.

[23] C.R. Saravanan and R. Peters. Internet Innovation Centre Workshop, Winnipeg, Manitoba., 1997. *WWW in DSVM* System Demonstration.

[24] P. Scheuermann, J. Shim, and R. Vingralek. A Case for Delay-Conscious Caching of Web Documents. In *Sixth International World Wide Web Conference*, Santa Clara, California, USA, April 1997.

[25] M. Spasojevic, C.M. Bowman, and A. Spector. Using Wide-Area File Systems Within the World-Wide Web. In *Second International WWW Conference*, Chicago, U.S.A., October 1994.

[26] The URN Implementators. Uniform Resource Names. *D-Lib Magazine*. February 1996.

[27] J. Vochteloo, S. Russell, and G. Heiser. Capability-Based Protection in a Persistent Global Virtual Memory System. Technical Report SCS&E Report 9303, School of Computer Science, University of New South Wales, March 1993.

[28] J. Wilkes and B. Sears. A Comparison of Protection Lookaside Buffers and the PA-RISC Protection Architecture. Technical Report HPL-92-55, Hewlett Packard, March 1992.

[29] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time : Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware. In *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOOS92)*, pages 364–377, 1992.

[30] R. P. Wooster and M. Abrams. Proxy Caching That Estimates Page Load Delays. In *Sixth International World Wide Web Conference*, Santa Clara, California, USA, April 1997.

[31] J.J. Yang and G.E. Kaiser. An Architecture for Integrating OODBs with WWW. In *Fifth International WWW Conference*, France, May 1996.

[32] A. Yoshida. MOWS: Distributed Web and Cache Server in Java. In *Sixth International World Wide Web Conference*, Santa Clara, California, USA, April 1997.

[33] W. Zwaenepoel, K. Bennett, and J.B. Carter. *Operating Systems of the 90s and Beyond*, pages 56–60. Springer-Verlag LNCS 563, 1991.