A Robust Persistent Storage Architecture for ACE

By

Sivaprasath Murugeshan

Bachelor of Engineering
Computer Science and Engineering
Anna University, Chennai, India, 1998

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

_____
Dr. Jerry James
(Committee Chair)

_____
Dr. Arvin Agah
(Committee Member)

_____
Dr. Susan Gauch
(Committee Member)

_____
Date of Acceptance

Abstract

In a pervasive computing environment, computation and storage are distributed across a heterogeneous network. A persistent data store is necessary to make the environment robust and to meet long-term storage requirements. It can be used to store personal workspaces of the users. It should provide reliable storage in spite of failures in parts of the system.

The persistent storage architecture consists of servers that are robust and highly available. We use “peer-to-peer” server architecture. The servers are multithreaded in order to process requests from clients and peer servers concurrently.

The clients send requests to the servers to perform desired operations on objects and namespaces. Objects are uninterpreted sequences of bytes. A namespace is a collection of objects. The persistent store provides multiple namespaces in which objects can be stored. It provides a consistent view of objects and namespaces.

The possible orders of interleaving of different atomic steps in different threads and the actions taken upon receiving different types of messages arriving from other clients or servers are studied. Robustness of the server is verified by proving that consistency is maintained irrespective of the interleaving of different atomic steps.

Acknowledgements

I would like to express my sincere gratitude to Dr. Jerry James, my advisor, for guiding me in my thesis work. He has been a source of inspiration all through my stay in KU and the meetings that we had helped me to gain a good understanding of Distributed Computing. I thank him for his timely suggestions and encouragement.

I would like to thank Dr. Arvin Agah and Dr. Susan Gauch for serving on my thesis committee. I thank Dr. Gary Minden for providing resources for this work and helping us with several useful suggestions during our team meetings.

Thanks to Kalicut Ramakrishnan for working with me in this project and the discussions we had were very helpful. Thanks to Mangal Singh and Jedrzej Miadowicz for helping me to understand the theoretical aspects of Distributed Computing. Thanks to all the members of ACE team.

Finally, I would like to thank all those in ITTC for providing a wonderful to atmosphere to work. Thanks to all my friends who made my stay in KU a memorable one.

Table of Contents

Chapter 1

Introduction 1

1.1 ACE 3

1.1.1 ACE Overview 3

1.1.2 Persistent store 4

1.2 THESIS ORGANIZATION 4

Chapter 2

Background and Related Work 6

2.1 BACKGROUND 6

2.1.1 Design considerations 6

2.1.2 Consistency model 7

2.1.3 Failures 9

2.1.4 Programming model 10

2.1.5 Issues in multithreaded programs 11

2.1.6 Properties of the system 11

2.2 RELATED WORK 12

Chapter 3

Design 14

3.1 CLIENT 16

3.2 SERVER 17

3.3 STORE 19

3.4 OBJECT COMMANDS 19

a) store_object *20*

b) retrieve_object *20*

c) list_objects *21*

d) delete_object *21*

e) store_unique_object *22*

3.5 NAMESPACE COMMANDS 23

a) create_namespace *23*

b) list_namespaces *23*

c) delete_namespace *24*

d) clear_namespace *24*

3.6 CONSISTENCY MODEL 25

3.7 RESTART MECHANISM 25

3.8 SERVER JOINING 26

3.9 SERVER LEAVING 27

Chapter 4

Implementation details 28

4.1 TWO PHASE COMMIT 28

4.2 FAILURE DETECTION 29

4.2.1 Crash failures 29

4.2.2 Disk failures 29

4.3 IMPORTANT DATA STRUCTURES 30

4.4 MUTEX VARIABLES 33

4.5 CONDITION VARIABLES 34

4.6 THREAD STRUCTURE 34

4.6.1 'main' thread 35

4.6.2 'client_receive' thread 35

4.6.3 'peer_send' thread 37

4.6.4 'peer_receive' thread 38

4.7 DIRECTORY STRUCTURE OF THE PERSISTENT STORE 44

Chapter 5

Properties of the system 45

5.1 ASSUMPTIONS 45

5.1.1 Thread package 45

5.1.2 Communication mechanism 45

5.2 INVARIANTS 46

5.3 PROPERTIES 48

5.4 LIMITATIONS 51

Chapter 6

Conclusions and Future work 53

6.1 CONCLUSIONS 53

6.2 CONTRIBUTIONS 53

6.3 FUTURE WORK 53

6.3.1 Different network protocols 54

6.3.2 Different consistency models 54

6.3.3 Security issues 55

References 56

Appendix

Test results 58

Chapter 1

Introduction

The tremendous advances made in the domains of computation, storage and communication technologies have diversified the computing environment with the introduction of myriad devices of different dimensions and capabilities. A wide variety of software is used to harness the power of these resources. In this scenario, it becomes necessary to explore ways to make such resources more accessible to the users in different physical locations without violating the security and privacy assurances given by stand-alone devices. The solution for this problem lies in pervasive computing.

The idea of pervasive computing is to integrate myriad computational devices as diverse as a workstation and a PDA into a heterogeneous networked environment. In this environment, computation and storage are distributed across the network. The environment is more robust and user-friendly than traditional computing environments. New devices are easily accommodated in the network. Also, the devices are more accessible to remote users. The existence of different devices and the processes of computation and storage are transparent to the user. The users' personal workspaces are stored across the network and are accessible after proper authentication.

Pervasive computing provides several research opportunities in various domains of computer science. Some of the challenges are illustrated here.

Persistent workspaces that are accessible from different devices should be provided for users. These workspaces have to be stored across the network to ensure high availability and fault-tolerance. When storage is distributed across the network, ensuring data consistency is an important concern. Storage architectures for pervasive computing environments have to be designed to meet the requirements mentioned above, viz., fault tolerance, high availability and data consistency. This makes the computing environment more robust and useful.

Communication across a pervasive computing environment involves a huge amount of data being transferred across the network. Conventional network protocols may introduce performance bottlenecks in such cases. Low latency network protocols that can be used with many devices help improve the performance of the network. User-level network protocols provide better latency by reducing the processing overhead in the communication endpoints. Also, communication across the network has to be secure. Data encryption and user authentication methods are needed for ensuring security and privacy in the environment.

A wide variety of services are provided for the users. Service discovery protocols are required to enable the user to utilize the services provided by the environment in an efficient manner. Face recognition and voice recognition help in authenticating users

to access services. Such capabilities make the pervasive computing environment more user-friendly.

1.1 ACE

1.1.1 ACE Overview

The objective of Ambient Computational Environment (ACE) project is to make computing pervasive by embedding myriad devices into a heterogeneous network and providing services to users in different physical locations. Users are relieved from the burden of being in physical contact with all the devices they want to use. After proper authentication, users can access the services provided by ACE.

The services provided by ACE include fingerprint identification, audio capture and play, video capture and play, projector control, etc. Services are the basic mechanisms for controlling the functioning of different devices. The services are registered in the ACE service directory. The ACE daemon is the software operating in the background that provides services to the user. Various services in the ACE environment are implemented as ACE daemons.

The vision of the ACE project is to create ‘smart rooms’. In a smart room, the user can authenticate himself with a ‘smart device’ such as a fingerprint identifier, biometric identifier, etc., and get access to his workspace. His workspace contains snapshots of the applications that he had been running during his previous login session. From his workspace, he can run the desired applications. The ACE

infrastructure provides these services in such a way as to relieve the user from knowing the intricacies of the actual devices.

1.1.2 Persistent store

In an ACE environment, several situations arise in which a persistent data store is desirable. For example, user contexts should survive the termination of any particular program, machine failures, and network failures. The persistent store should be robust, fault tolerant and highly available. The data that the users want to store can be anything; e.g., text files, binary files, user contexts, etc. We use the term ‘object’ to indicate an uninterpreted sequence of bytes. Each object is associated with a unique name. A namespace is a collection of objects. The persistent store consists of multiple namespaces.

Objects are stored in different machines that act as servers. Redundancy of stored data makes it necessary to ensure data consistency. Operations on the persistent store are performed such that a consistent view of objects and namespaces is maintained across the environment. Failure detection and recovery mechanisms make the environment robust. A well-defined interface is provided to the clients for storing and retrieving data in the persistent store.

1.2 Thesis Organization

The thesis is organized as follows.

Chapter 2 describes the background and related work. Storage architectures that are relevant to pervasive computing, e-business and e-services are described. Research work in the areas of consistency models and analyzing properties of distributed systems are described.

Chapter 3 describes the design issues. Various entities in the architecture of the persistent store and the interaction among those entities are discussed. Commands that are supported by the persistent store are discussed in detail. The consistency model chosen for the persistent store is explained.

Chapter 4 describes the implementation details. The persistent store servers are multithreaded. The functions of different threads are discussed. Mutex variables and the data structures they protect are discussed.

Chapter 5 describes the properties of the system. Assumptions about the network and software used, properties guaranteed by the system and the limitations of the system are discussed.

Chapter 6 deals with the conclusions of this work and possible extensions to it.

Chapter 2

Background and Related Work

2.1 Background

Replication of services is commonly used to achieve robustness, fault tolerance and high availability. Replicated servers can act as web servers, file servers, name servers, etc. Storage architectures for pervasive computing environments are based on the concept of service replication. This ensures that the storage is persistent and is available for clients' access even when failures occur in parts of the environment.

Servers that are kept in different physical locations perform the same task of storing and retrieving data. A well-defined interface is presented to the client for the services provided by the storage architecture. Clients are aware of the existence of multiple servers and have a complete list of servers. A server can be chosen at random and a service be requested. In case of failures, other servers can be contacted.

2.1.1 Design considerations

The serious issues that have to be considered during the design of a persistent storage architecture are consistency guarantees provided to the clients in the environment, synchronization among the servers, servers being aware of the status of other servers in the environment and the way stored data is organized in a disk.

2.1.2 Consistency model

Servers that are distributed across a network perform operations as desired by the clients. While performing such operations, they synchronize among themselves to ensure that clients' views of the store adhere to certain well-defined rules. The semantics of the abstraction provided by the store defines the consistency model.

Each server performs different operations on stored data. Those operations fall under two categories: 'read' and 'write'. A 'read' operation can be performed by a single server without consulting other servers. A 'write' operation has to be performed after consulting other servers. The sequence of 'read' and 'write' operations in different servers may give different results. A consistency condition can be thought of as a set of acceptable results considering the ordering of operations in a single server and the ordering of operations in real-time.

Strong consistency models typically give an abstraction that is easy for the clients to understand. The set of acceptable orderings are limited. The protocols implementing strong consistency models lay emphasis on perceived 'correctness' rather than performance of the system when operations are performed.

Weak consistency models have the advantage of giving better performance. More orderings are considered acceptable. In some cases, programmers find the server semantics difficult to comprehend.

Strict consistency is the strongest consistency model that we can think of. This model assumes the existence of a global clock. The real-time ordering of the events is the only acceptable ordering. Every read operation should return the result of the most recent write performed. This is possible only when write operations on servers are informed to other servers instantaneously. Propagation delay for a message transmission across a network is always non-zero. So, this model is impractical. Strict consistency is just a conceptual idea that is not implemented in any distributed shared memory system so far.

Strict consistency is not absolutely necessary for distributed applications in general. Programmers can afford to use weaker consistency models for meeting the requirements of the system. If operations performed are seen in the same order by every server, it will be sufficient for many systems. Synchronization primitives may be used when a particular ordering is absolutely necessary. The perceived ordering need not conform to the real-time ordering. Let us consider two consistency models: linearizability and sequential consistency.

Linearizability [1] is weaker than strict consistency. In a linearizable system, an acceptable ordering is one in which the operations are ordered in some sequential fashion consistent with read-write semantics and non-overlapping operations are ordered in the same way as the real-time ordering. Two operations are overlapping if each operation starts before the other operation ends. All ‘write’ operations are seen

in the same order by all servers. When each object satisfies the condition for linearizability, the system as a whole is linearizable [1].

Sequential consistency [2] is weaker than linearizability. In a sequentially consistent system, operations have to be consistent with some sequential order seen by every server, but need not conform to the real-time ordering. The restriction on non-overlapping operations that exists for a linearizable system is relaxed in this model.

The trade-off between desired degree of correctness and performance is the deciding factor for a particular consistency model to be chosen for the distributed storage architecture.

2.1.3 Failures

The degree of robustness of the persistent storage architecture very much depends on the types of failures the system is able to detect and the recovery mechanisms the system has. The system should function correctly even when failures occur in parts of the system. Different kinds of machine or network failures may occur in the system.

Machine failures include crash failures, disk failures, denial of service attacks, etc. Crash failure occurs when the machine acting as a server crashes. Disk failure occurs when a machine is alive, but is unable to read or write to the disk successfully. ‘Denial of service’ attack occurs when a server is flooded with requests from clients in such a way as to impair its performance. These are common problems that can occur in any machine.

Messages sent across a network may be lost or corrupted. Network partitions occur when a network is divided into multiple partitions such that machines in one partition are unable to communicate with those in other partitions. The existence of network partitions may result in inconsistencies among objects stored across the network. Network partitions are very difficult to detect.

2.1.4 Programming model

A server responding to requests from multiple clients and servers needs to perform several tasks concurrently. The programming model for such an application is usually based on either multithreading or event driven programming. In multithreading, concurrency is achieved by allowing different threads to perform independent tasks. In event driven programming, the server behaves like a finite state machine with the transitions to different states being based on the requests received from various clients or servers. Event handlers are invoked to handle different requests. Processing specific requests is the function of the handlers. Generally, event handlers are short-lived.

The multithreaded model is easier to design. Debugging multithreaded programs is difficult because of the limitations of the existing debugger tools. Event driven software often proves difficult to design because of the complex finite state machine that simulates the threads [3]. The choice between the multithreaded model and the event driven model can be decided on the basis of whether the tasks are CPU bound or I/O bound. In the case of multithreading, when one thread is doing I/O operations,

another thread can utilize the CPU. Context-switching and locks increase the overhead with threads. For a program that is either mostly CPU bound or mostly I/O bound, the performance gain due to multithreading may be overshadowed by the context-switching overhead. Under such circumstances, all threads wait for the same resource which defeats the purpose of multithreading.

2.1.5 Issues in multithreaded programs

The general issues in multithreaded programs are mutual exclusion, deadlock and starvation. Data structures that are shared by different threads have to be protected by locks to ensure that operations on such data are atomic. When locks are used to ensure mutual exclusion, the possibility of 'hold and wait' or 'circular wait' may result in the occurrence of deadlock. Starvation occurs when the same thread keeps acquiring the lock leaving the other threads waiting for long periods of time to execute. The issue of starvation has to do with the design of the thread scheduler whereas it is the responsibility of the programmer to ensure mutual exclusion and deadlock freedom.

2.1.6 Properties of the system

'Safety' properties of a system state that the system does not do anything wrong. An example is deadlock freedom. This property ensures that deadlock does not occur in a multithreaded program where shared data is protected using locks. 'Liveness' properties give guarantees about the way certain functions are performed by the

program. An example is starvation freedom. This property ensures that acquiring locks is equally likely for all threads. Safety and liveness properties characterize the behavior of the server.

2.2 Related work

Ninja [4] is a pervasive computing architecture that consists of services that are available across the Internet. Ninja is designed for a heterogeneous network with myriad services available for users that can access them through myriad devices. The architecture of Ninja consists of three basic components: bases, devices and active proxies. ‘Bases’ are the units that store persistent state and that provide scalable services to the users. ‘Active proxies’ store the ‘soft state’ and serve as interfaces between bases and unintelligent devices. Devices can include simple devices with limited intelligence and functionality. Ninja’s communication mechanism is based on Jaguar VIA. This is based on user-level network protocols that achieve low latency communication. Ninja facilitates a solution for robust storage architecture for a pervasive computing environment.

Nile [5] is a distributed computing solution for computationally intensive tasks in high-energy physics. The Nile architecture takes advantage of the parallel nature of the computation involved. The assigned job is divided into subjobs that are allocated to a large number of processors. Process failures, disk failures and network failures

are some of the failures that are taken care of by the system. Nile is an example of a distributed computational environment that tolerates failures in parts of the system.

E-speak from Hewlett-Packard, WebSphere from IBM and Weblogic from BEA provide a basic infrastructure for e-business applications. A scalable infrastructure that can support several applications makes it easier for application developers as they can concentrate on specific functionalities of the applications rather than issues that are common to many applications. These systems facilitate developing scalable, fault-tolerant Internet services.

Mustaque Ahamad and Rammohan Kordale [6] propose “local consistency” based on which scalable consistency protocols can be developed. Two protocols for implementing strong consistency are illustrated. Correctness of the protocols implementing the specified consistency model is explained.

Marios Mavronicolas and Don Roth [7] study the implementation of linearizable read/write objects. Time complexity analysis for read/write operations is done based on different assumptions on timing of the operations. This provides a good understanding of the working of linearizable systems.

Chapter 3

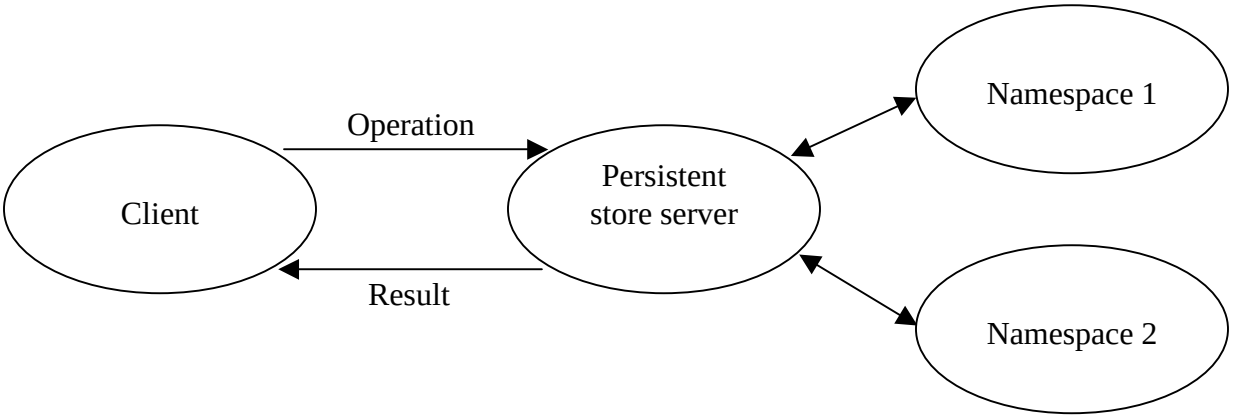
Design

The persistent store provides reliable storage even if some part of it is not functioning. It consists of a set of servers that present a consistent view of storage to the clients. It provides multiple namespaces within which named objects can be stored.

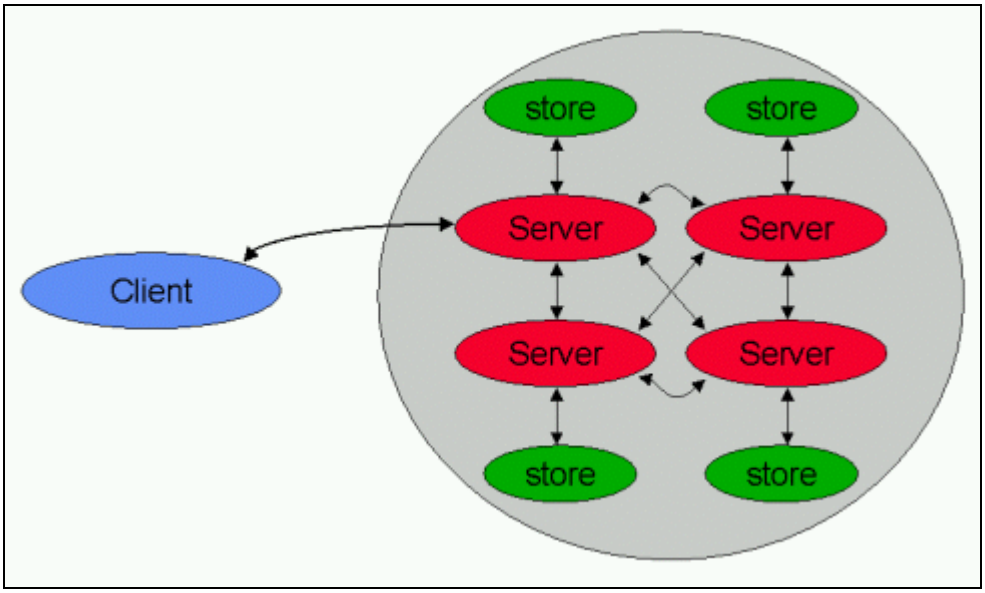
The fundamental unit of storage in the persistent store is referred to as an *object*. An object is an uninterpreted sequence of bytes. Every object has a unique name within its namespace. The object may be a file, a snapshot of the user context or any data that the client wants to store. It need not adhere to any particular format. A *namespace* is a collection of objects. Every namespace has a unique name. The persistent store consists of multiple namespaces, which in turn may contain multiple objects.

Clients contact the persistent store in order to store or retrieve objects by name. The client sends a message requesting that some operation be performed on the objects or namespaces. The server performs the operation and returns the result to the client. While performing such operations, the server makes sure that a consistent view of objects and namespaces is maintained across the set of servers.

The relationship among the various entities is illustrated in the following diagram.



The persistent storage architecture consists of three major components: client, server and store. There are multiple servers, each associated with a distinct store. The servers communicate among themselves when they perform the operations for the clients so that a consistent view is maintained across the stores associated with different servers. The following diagram illustrates the relationship among these components.



3.1 Client

Any machine that is part of an ACE environment is referred to as a client. The clients discover the addresses of persistent store servers from configuration files. The requests processed by a server are as follows:

a) *store_object* – store a named object in a namespace.

b) *retrieve_object* – read a named object from a namespace.

c) *list_objects* – list all objects in a namespace.

d) *delete_object* – delete a named object from a namespace.

e) *store_unique_object* – choose a unique name for the object and store that in a namespace.

f) *create_namespace* – create a namespace in the store.

g) list_namespace – list all namespaces in the store.

h) *delete_namespace* – delete a namespace from the store.

i) *clear_namespace* – delete all objects in a namespace.

The client selects any server and sends the request. The server processes the request and sends the result back to the client

3.2 Server

The main function of the persistent store server is to provide a mechanism for storing and retrieving objects. The persistent store server is fault-tolerant and highly available. It provides a consistent view of the stored objects to the clients. The server is randomly selected by the client to increase the likelihood of a balanced load.

The servers are peer servers. The servers are multithreaded in order to process requests from clients and peer servers concurrently. The server has the following threads:

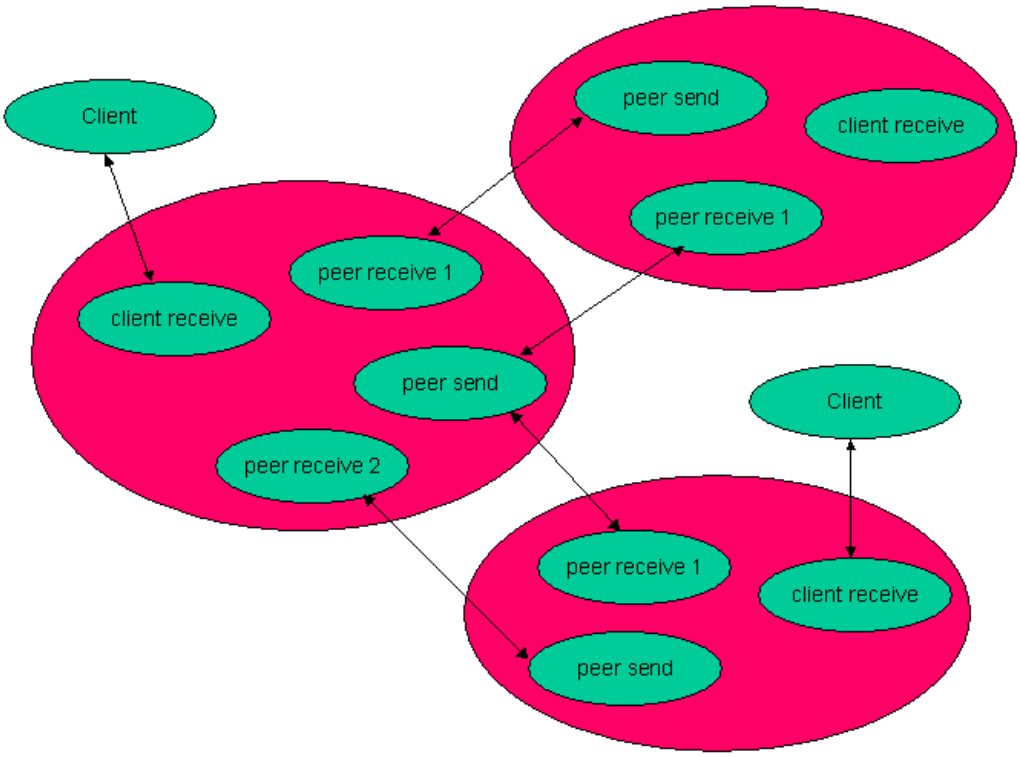
a) 'main' thread - initializes data structures, creates other threads and updates data structures.

b) ‘client_receive’ thread – creates multiple child threads that process requests from multiple clients concurrently. There is one child thread for every client.

c) ‘peer_send’ thread – sends updates to peer servers.

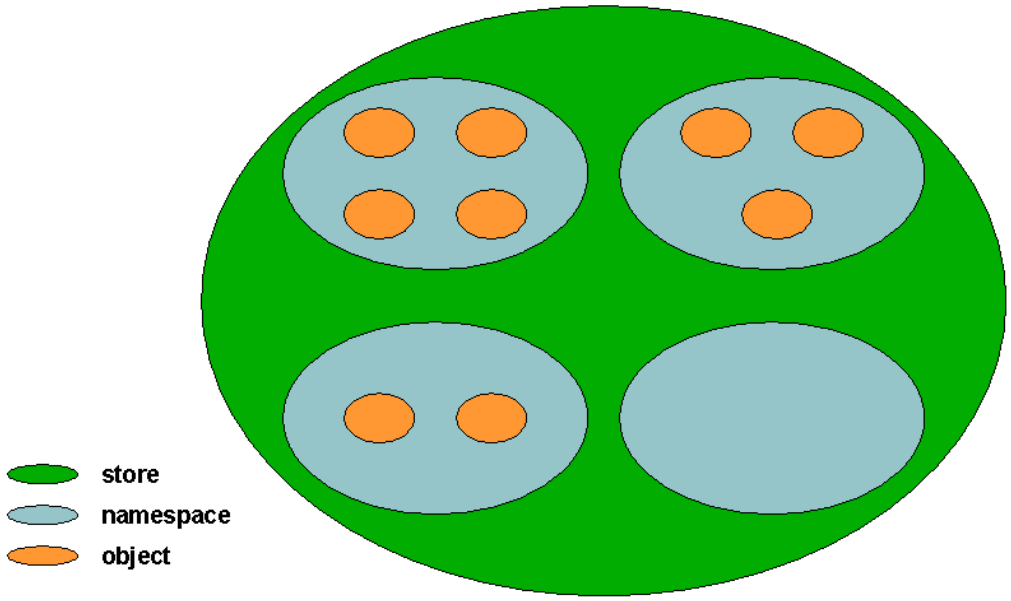
d) ‘peer_receive’ thread – creates multiple child threads that process requests from multiple peer servers concurrently. There is one child thread for every peer server.

The interaction among multiple threads in the persistent store server is illustrated in the diagram given below.



3.3 Store

The store is a collection of objects and namespaces. Any non-volatile storage device can be used as the store. Every namespace has a unique name. Namespaces contain named objects. The relationship among the various entities in the store is as shown below.



3.4 Object Commands

The persistent store supports the following object commands, i.e. the operations that can be performed on objects.

a) store_object

The ‘store_object’ command is issued by the client to store an object in a specified namespace. The object may be replicated if desired by the client. The *replication flag* indicates whether the object has to be replicated or not. The default value for replication flag is ‘true’ which indicates full replication.

Command Name: *store_object*

Arguments:

Namespace - The namespace in which the given object is to be stored.
Value Data Type: string

Name - The name of the object to be stored.
Value Data Type: string

Object - The object to be stored.
Value Data Type: uninterpreted bytes

Replication flag - The flag that indicates whether replication is desired or not.
Value Data Type: boolean

Response: The server stores the given object in the specified namespace. The object is replicated if desired by the client. The client is notified of success. Otherwise, an error message is returned.

b) retrieve_object

The ‘retrieve_object’ command is issued by the client to retrieve an object specified by the given name from the specified namespace.

Command Name: *retrieve_object*

Arguments:

Namespace - The namespace from which the object is to be retrieved.
Value Data Type: string

Name - The name of the object to be retrieved
Value Data Type: string

Response: The server retrieves the object from the specified namespace and returns it to the client, if the namespace and the object exist. Otherwise, an error message is returned.

c) list_objects

The 'list_objects' command is issued by the client to list all objects within the specified namespace.

Command Name: *list_objects*

Arguments:

Namespace – The namespace to list.
Value Data Type: string

Response: The server returns a list of all objects within the specified namespace.

d) delete_object

The 'delete_object' command is issued by the client to delete an object with a specified name in the specified namespace.

Command Name: *delete_object*

Arguments:

Namespace - The namespace from which the specified object is to be deleted.
Value Data Type: string

Name – The name of the object to delete.
Value Data Type: string

Response: The server deletes the object specified by the name and namespace, if it exists. The client is notified of success. Otherwise, an error message is returned.

e) store_unique_object

The ‘store_unique_object’ command is issued by the client to store an object with a name distinct from all the existing names in the specified namespace. This command can be used when the client is not interested in storing the object with a specific name. Any distinct name is enough. The server chooses and returns the name. The default value for replication flag is ‘true’ which indicates full replication.

Command Name: *store_unique_object*

Arguments:

Namespace - The namespace in which the object is to be stored.
Value Data Type: string

Object - The object to be stored.
Value Data Type: uninterpreted bytes

Replication flag - The flag that indicates whether replication is desired or not.
Value Data Type: boolean

Response: The server responds with the name selected for the object in the specified namespace. The object is replicated if desired by the client. An error message is returned if the server is unable to satisfy this request.

3.5 Namespace Commands

The persistent store supports the following namespace commands, i.e. the operations that can be performed on namespaces

a) create_namespace

The ‘create_namespace’ command is issued by the client to create a namespace.

Command Name: *create_namespace*

Arguments:

Namespace - The identifier of the namespace to be created by the server.
Value Data Type: string

Response: The server creates the namespace if there is no existing namespace with the same identifier. The client is notified of success. Otherwise, an error message is returned.

b) list_namespaces

The ‘list_namespaces’ command is issued by the client to list all namespaces in the server.

Command Name: *list_namespaces*

Arguments:
None.

Response: A list of all namespaces existing in the server.

c) delete_namespace

The ‘delete_namespace’ command is issued by the client to delete the specified namespace. All objects in the specified namespace shall be deleted as well.

Command Name: *delete_namespace*

Arguments:

Namespace – The namespace to be deleted.
Value Data Type: string

Response: The server deletes all objects in the specified namespace and the namespace, if it exists. The client is notified of success. Otherwise, an error message is returned.

d) clear_namespace

The ‘clear_namespace’ command is issued by the client to clear the specified namespace. All objects in the specified namespace shall be deleted, but the namespace continues to exist.

Command Name: *clear_namespace*

Arguments:

Namespace – The namespace to be cleared.
Value Data Type: string

Response: The server deletes all objects of the specified namespace, if it exists. The client is notified of success. Otherwise, an error message is returned.

3.6 Consistency model

The consistency model provided by a persistent store determines the set of guarantees that the client can expect when a read or write operation is performed on the store. Choosing a consistency model for a concurrent system is based on the tradeoff between the degree of correctness and the performance of the system when it performs read or write operations.

A concurrent system is sequentially consistent if the results of any execution are the same as if the operations of all the machines were executed in some sequential order and the operations of each individual machine appear in this sequence in the order specified by its program. If this sequential order is in accordance with the real time order for non-overlapping operations, the system is said to be linearizable.

The view provided by the persistent store to the clients is linearizable. The only consistency model that is stronger than linearizability is strict consistency, which is practically impossible in any distributed system. To implement linearizability, the two-phase commit algorithm is used.

3.7 Restart mechanism

The restart mechanism defines actions to be taken upon recovery after a machine or network failure. It works as follows.

The incarnation file to be used at the time of restart is kept in a specific location and the incarnation number is stored in the file.

An incarnation number indicates the incarnation of the server. When a server recovers from a fault, the incarnation number is modified to differentiate between the messages sent before the fault and after the fault. The incarnation number of the server is included in every message sent to peer servers.

The server checks for the existence of the incarnation file when it restarts. If the file already exists, the incarnation number is incremented and stored. Otherwise, the file is created and the incarnation number is set to 0. Whenever the server starts, its incarnation number is sent to all peer servers. The file is deleted when the server undergoes a normal shutdown.

3.8 Server joining

A new server joining the set of servers that are alive has to be atomic. Every server that is alive should be aware of the new server joining. When a new server comes alive, the server that receives a connection request from the new server initiates a two-phase commit. Other peer servers that participate in two-phase commit will not initiate another two-phase commit when they receive connection requests from the new server. Only after the two-phase commit succeeds, the new server can send messages to the other servers. The actions taken upon receiving connection requests from the new server are explained in Chapter 4.

Processing client requests when the new server is joining may introduce inconsistencies in the persistent store. So, we ensure that no new server joins the group when client request processing involving many servers is in effect. This property of the persistent store is explained in Chapter 5.

3.9 Server leaving

A crash failure occurring in a server is detected by SIGPIPE handler. When a crash failure of a server is detected, all peer servers that are alive are informed about the failure. Two phase commit is not necessary in this case. If a server is not aware of a peer server crashing, the crash will be detected when trying to send a message. So, there will not be any inconsistency among the servers because of a server leaving and other servers not agreeing with one another about that. A server may opt to terminate because of disk failures. In that case, it sends a message to all peer servers indicating failure.

Chapter 4

Implementation details

The persistent store server is multithreaded. Multithreading is implemented using the 'pthreads' library in Linux. TCP/IP sockets are used for communication among servers and between client and server.

4.1 Two phase commit

The operations on the persistent store are linearizable. To implement linearizability, the two-phase commit algorithm is used. The two-phase commit algorithm works as follows.

Before doing an operation involving all servers in the system, the server that is processing the client request sends a "Ready" message to all peer servers that are alive. The servers, which are ready to perform the requested operation, send a "Yes" message. The servers that are unable to perform the requested operation send a "No" message. When two peer servers request operations to be performed on the same object, the server sends a 'Yes' message to one peer server and a 'No' message to another peer server. Such ties are resolved based on the IP address of the server that has sent the request. The server that initiated the two-phase commit waits for a finite amount of time to get responses from its peers. The waiting time can be varied by the administrator before compilation. The server should either get "Yes" messages from its peers or detect a machine fault of its peers to proceed with committing. If this

condition is satisfied, the server sends a “Commit” message and its peers should commit the operation. Otherwise the server sends an “Abort” message and the operation will be aborted. The operation will not be performed in any of the servers and an error message will be returned to the client.

In the next section, we explain how the two-phase commit algorithm guarantees the linearizability property of the persistent store.

4.2 Failure Detection

A network fault or a machine fault may occur in any part of the server environment. The following faults can be detected.

4.2.1 Crash failures

When a server tries to send a message using a TCP/IP socket to another server that is dead, a SIGPIPE signal is triggered. A SIGPIPE handler that handles server crashes is implemented. Upon detecting a server crash, no further messages are sent to the server till it recovers. All peer servers that are alive are informed about the server crash.

4.2.2 Disk failures

If a server is unable to do local I/O successfully, it will inform its peers about the failure. Messages will not be sent to the server till it recovers. When the server restarts after recovering from failure, peer servers are informed.

4.3 Important data structures

The following are the important data structures used in the server.

1) namespace_hash_table

The hash table stores the names of namespaces in the persistent store. Each entry in the table consists of the following fields.

name – name of the namespace

lock_holder – identifier of the server that is performing an operation on this namespace

object – pointer to a linked list containing the attributes of the objects belonging to this namespace.

2) list of objects

Each entry in the list of objects contains the following fields.

name – name of the object

lock_holder – identifier of the server that is performing an operation on this object.

3) peer_attributes

This is an array containing the attributes of the peer servers. It is initialized after discovering the addresses of peer servers from the configuration file. Initialization takes place when the server starts. It is updated when server crashes are detected or when servers come alive.

Each element in the array has

- *peer identifier*

  IP address of the server is used as the identifier

- *state of the peer server*

  The state of the peer server can be any of the following

  ALIVE – peer server is sending and receiving messages

  DEAD – peer server is dead

  JOINING – peer server wants to join. A two-phase commit that has been initiated by another peer server is going on. When the two-phase commit succeeds, the state of the peer server will be ALIVE.

  TO_JOIN – a request has been received from the peer server for joining. A two-phase commit has to be initiated. When the two-phase commit succeeds, the state of the peer server will be ALIVE.

- *socket id* that will be used to send messages to the peer server
- *thread id* of the peer_receive thread that receives updates from the peer server.
- *incarnation number*

  The incarnation number indicates current incarnation of the peer server.

4) client_request_list

This is a linked list containing requests that have arrived from clients. Only those requests that need to be known by other peer servers are stored here. Those requests that can be processed without contacting other peer servers are not updated in this list.

The client_receive thread adds requests from clients to the tail of the list. The peer_send thread processes requests from the head of the list.

Each element in the list has

- *request to be processed*

  This may be the operation requested by the client or the information that a new server is joining. These operations will be performed after doing a two-phase commit.

- *request parameters*

  These are the parameters for the request to be processed. The list of parameters is in accordance with the syntax of the commands as specified in the previous chapter.

- *object*

  For requests involving objects, this field contains the object to be stored in servers. This field is valid only for the requests ‘store_object’ and ‘store_unique_object’.

- *incarnation number of the server*

  The incarnation number is included with every message so that peer servers can distinguish between messages from different incarnations.

- *index in the array of condition variables*

  The ‘peer_send’ thread notifies the condition variable as identified by this index so that ‘client_receive’ thread knows the status of operation in other peer servers and takes appropriate action.

5) cond_var_array

Each client request is updated in the client_request_list. The request will be performed only when it can be performed in all peer servers. So, each thread processing a client request sets a condition variable and waits for that variable to be signaled by the ‘peer_send’ thread. cond_var_array is a variable sized array that contains condition variables and associated mutex variables.

Each element in the array has

- *condition variable*
- *associated mutex variable*
- *flag to indicate if the condition variable has been used or not*
- *flag to indicate the status of two-phase commit*

4.4 Mutex variables

The data structures that are accessed by multiple threads are protected by mutex variables. The following are the mutex variables used.

1) mutex_peer_attributes

This mutex protects the ‘peer_attributes’ data structure.

2) mutex_client_request_list

This mutex variable protects the linked list ‘client_request_list’.

3) mutex_cond_var_array

‘cond_var_array’ is accessed by multiple threads processing client requests. This mutex variable protects ‘cond_var_array’.

4) mutex_hash_table

The hash table that stores the attributes of namespaces is accessed by multiple threads processing client requests. This mutex variable protects ‘hash_table’.

4.5 Condition variables

1) cond_peer_join

This condition variable is set when a peer server asks to join and two phase commit has already been started by a peer server that is alive. The variable is signaled when two phase commit succeeds.

2) cond_var_array

This is the variable sized array of condition variables as described earlier.

4.6 Thread Structure

The server process has the following threads.

1) ‘main’ thread
2) ‘client_receive’ thread
3) ‘peer_send’ thread
4) ‘peer_receive’ thread

The functions performed by various threads are explained as follows.

4.6.1 'main' thread

The functions of this thread are to initialize many data structures and to create other threads. From the configuration file, it discovers the addresses of the peer servers and initializes 'peer_attributes'. It updates incarnation files and sets the 'incarnation_number' value. The SIGPIPE handler, which helps detect server crashes, is installed. Mutex variables and condition variables are initialized. This thread creates the client_receive, peer_send and peer_receive threads.

Pseudocode:

initialize peer_attributes
install sigpipe handler
initialize thread_sig_mask
update incarnation file
initialize mutex variables
initialize array of condition variables
initialize hash table
create client_receive, peer_send and peer_receive threads.
wait for other threads to terminate
exit

4.6.2 'client_receive' thread

The 'client_receive' thread waits for connection requests from clients. Client requests are sent to a specified port. On receiving a connection request, a child thread is created. It is the child thread that actually processes the requests and returns the status of the operation to the client. Client requests are processed concurrently by

multiple child threads. Each child thread processes requests from a particular client in sequential order till the client notifies the server that no more requests need to be processed.

The child thread receives the request, i.e., the operation to be performed on the persistent store. The request is parsed and parameters are identified. For operations that involve only reading the contents of the store, other peer servers need not be contacted. Those requests are performed immediately and the result is returned to the client. Those operations that involve writing to the persistent store are performed after agreeing with other peer servers. The incarnation number of the server is included in the entry to be added to the client_request_list. The index in the cond_var_array that has to be updated by peer_send thread is calculated after searching cond_var_array. This thread waits for notification of the specified condition variable. After discovering the status of two-phase commit, either the operation is performed or the client is informed that it failed.

Pseudocode:

block signals (that are in thread_sig_mask).
listen for connection requests from client receive port
accept connection and create a child thread that will receive requests.

(the child thread is created in the detached mode)

The child thread does the following.

forever
do
get requests from clients

```
    parse the request
    if request is end_of_requests, terminate the thread.
    receive the object (for store_object and store_unique_object commands)
    if server need not inform peers
      do local i/o and respond to the client
   else
    begin
      get index of cond_var_array
      create a client_request
      add  the request to client_request_list along with incarnation number
      timed wait for the result of two-phase commit
      if sigalled
         do local i/o and inform the client of success
      else (if timeout)
         inform the client of failure.
      update cond_var_array
    end
 done
```

4.6.3 'peer_send' thread

The peer_send thread reads client_request_list and sends the requests to peer servers. The two-phase commit algorithm is applied before operations are committed. The status of two-phase commit is updated in cond_var_array and the corresponding condition variable is signaled.

Pseudocode:

```
forever
 do
  read client_request_list

  for each entry in client_request_list
   do
      call two_phase_commit
      wait for responses from peers (finite wait using select)
      receive two_phase_commit_yes or two_phase_commit_no
       if it returns commit, send commit message and object (if necessary)
       signal condition variable (read client_request_list and signal the
```

condition variable corresponding to the client_receive_thread).
else send abort message to peers
if EPIPE is returned for all peers
(other servers think i'm dead. so, die and come back alive)
begin
reset peer_attributes
init_peer_attributes
end
delete request from client_request_list
done

done

4.6.4 ‘peer_receive’ thread

The ‘peer_receive’ thread waits for connection requests from peer servers. On receiving a connection request, coordination with other peer servers takes place to ensure that the operation of a new server joining is atomic. After two phase commit succeeds, a child thread is created. It is the child thread that actually processes the requests from the peer server. There are *n* child threads for *n* peer servers. Requests from peer servers are processed concurrently by these child threads.

When a connection request is received from a peer server, the status of the other peer servers is discovered from ‘peer_attributes’. The following cases may occur.

1) All peer servers are dead

In this case, this is the only server that is alive. When a connection request arrives from another server, server joining is atomic. So, the two-phase commit protocol is unnecessary. A child thread is created and ‘peer_attributes’ is updated.

2) Peer server is already alive

In this case, the server is getting a connection request from a server that is already alive. This server is joining an existing group of servers. A child thread is created and ‘peer_attributes’ is updated.

3) Peer server is joining

The server is getting a connection request that wants to join the group. Another server, that is alive, has received a connection request from the same server and has initiated two phase commit. This server is waiting for the two-phase commit protocol to complete. This thread waits on a condition variable. The variable is signaled by the child thread, which receives requests from a peer server. A child thread is created and ‘peer_attributes’ is updated.

4) Peer server is yet to join

The server is getting a connection request from a server that wants to join the group. The two-phase commit protocol is initiated. After the successful completion of two-phase commit, a child thread is created and ‘peer_attributes’ is updated.

The child thread receives updates from peer servers and processes them. The following types of requests are processed.

1) Incarnation number

The incarnation number of the peer server is updated in peer_attributes.

2) Peer server dead

The state of the peer server is changed in peer_attributes. The child thread receiving updates from the server is terminated.

3) I am dead

This message is received when the peer server is facing some problems with disk operations. The state of the peer server is changed in peer_attributes and this thread is terminated.

4) New server joining

When a message is received from a peer server regarding a new server joining, the status of the server in peer_attributes is modified. A condition variable is set. The variable is signaled when all servers agree on the joining of a new server.

5) Two-phase commit ready

This message is sent when two phase commit is initiated. The type of request and parameters are sent along with this message. The list of parameters is parsed to get namespace name and object name. The current lock holder for the specified namespace or object is found. If no server holds the lock, the lock holder is set to the IP address of this server and two_phase_commit_yes message is sent.

When the lock is already held by some other server, the tie is broken using the server's IP address. The server with the smaller IP address value gets higher priority. Consider the following scenario. Servers A and B try to update the same object. A sends the request to B, C and D. B sends the request to A, C and D. Let us assume C gets the request from A first, locks the object and sets the lock holder to A. D gets the request from B first, locks the object and sets the lock holder to B. C gets B's request and D gets A's request. Let us assume A's IP address is smaller than B's IP address. After getting B's request, C sends two_phase_commit_no. D waits for the release of lock for the object. The transaction initiated by B is aborted and the one initiated by A is committed. Thus, contentions are overcome and consistency is maintained in the persistent store.

6) Two-phase commit abort

When an abort message is received, the namespace and the object are unlocked. The request initiated by the peer server fails to take effect.

7) Two-phase commit commit

When a commit message arrives, the request initiated by the server has to take effect. An object is received for the commands store_object and store_unique_object. If the request is to accept a new server into the group, the condition variable meant for server joining is signaled. Object and namespace commands work as described earlier. After successfully performing the operations, the object and the namespace are unlocked.

Along with the request, the incarnation number of the server is also sent. If the request is from a previous incarnation, it is not honored. Otherwise, the requested operation takes place.

Pseudocode:

block signals (that are in thread_sig_mask).
listen for connection requests from peer receive port
accept connection
check peer_attributes

case 'status of peer_attributes'

all peers are dead : create a child receive thread
(this is the first to join,
so no need for two phase commit)

server already set alive : create a child receive thread
(joining other servers that are already alive)

server is yet to join : call two_phase_commit
create a child receive thread

server joining : wait for peer_join condition variable to be signalled

create a child receive thread

The child thread does the following.

update peer_attributes

forever
do
receive messages from other servers

case request_type:

incarnation_number : update peer attributes with incarnation number

i_am_dead : update_peer_attributes
terminate this thread

peer_server_dead : update peer_attributes
terminate receive thread corresponding to the dead peer.

two_phase_commit_ready :
- parse the request
- get the object name and namespace name
- try to set lock for namespace and object
- if it is already locked and
server_id is greater than that of the lock holder
send two_phase_commit_no.
else if not locked
send two_phase_commit_yes
else
wait to acquire the lock and then send two_phase_commit_yes

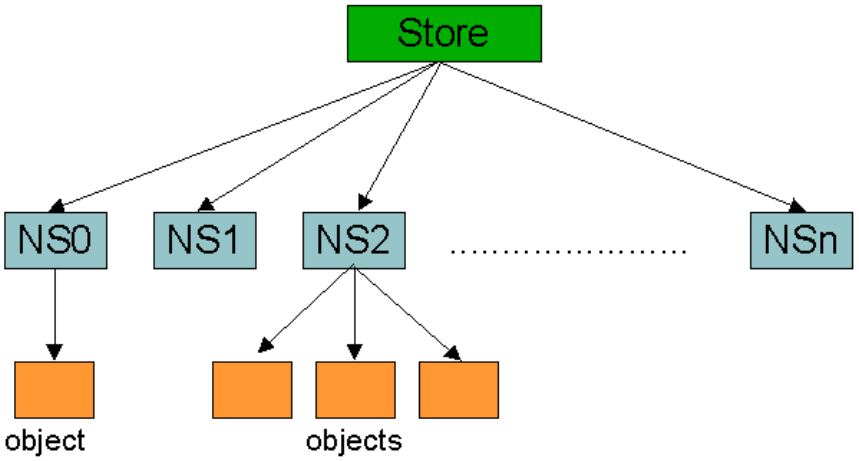
two_phase_commit_abort : reset lock for namespace and object.

two_phase_commit_commit :
- receive the object
- if request is new server joining, signal peer_join condition variable
- else
- begin
- do local i/o
- if disk failure, send i_am_dead message.
- end

done

4.7 Directory structure of the persistent store

The persistent store is a specific directory in the machine that acts as the server. It contains the subdirectories that are namespaces. Objects belonging to a particular namespace are stored as files in that directory. The type of the file can be anything, e.g., text file, binary file, C program, etc.



Chapter 5

Properties of the system

The properties of the persistent store implementation are discussed in this chapter. The assumptions regarding the thread package and communication mechanism are as follows.

5.1 Assumptions

5.1.1 Thread package

The ‘pthreads’ library in Linux is used for implementing multithreading.

Proposition 1: The thread scheduler is starvation free.

Proposition 2: Creating a child thread does not block.

Proposition 3: Terminating a child thread does not block.

5.1.2 Communication mechanism

TCP/IP sockets on Linux are used for communication between the client and the server and among the servers.

Proposition 1: All messages that are sent are eventually delivered when there is no crash. Messages are not lost, corrupted or misdirected.

Proposition 2: Every crash is eventually detected.

When the ‘send’ or ‘receive’ end of the connection is broken, the SIGPIPE signal is triggered in the other end. An EPIPE error is returned for any system call that uses the socket.

Proposition 3: We have a perfect failure detector. So, all detected crashes are crashes.

5.2 Invariants

Invariants are the conditions that are always true, except when an atomic action is being performed. The following invariants are true in our implementation.

Invariant 1: All shared data structures are protected by locks.

This is a ‘safety’ property. All shared data structures and their associated mutex variables are explained in the previous chapter. This guarantees mutual exclusion in the multithreaded server.

Invariant 2: Deadlock does not occur.

Deadlock is a common problem that is possible in multithreaded programs where locks are used to guarantee mutual exclusion. The necessary conditions for a deadlock to occur are: mutual exclusion, no preemption, hold and wait and circular wait.

Here, we explain how circular wait does not occur in the program. In peer_send thread, mutex_client_request_list is locked and then we try to acquire a lock for mutex_peer_attributes. This is the only instance of hold and wait. There is no instance where we acquire mutex_peer_attributes and try to acquire client_request_list. So, circular wait is avoided. Hence, deadlock is avoided.

Invariant 3: The number of ‘peer_receive’ threads will eventually be the same as the number of servers set 'alive' in peer_attributes.

We create a peer_receive child thread only after receiving a connection request from a peer server. The child thread updates peer_attributes once it is created. During this update, the peer server is set ‘alive’ in peer_attributes.

A server tries to connect to all peer servers when it starts up. During the initialization of peer_attributes, the peer server is set ‘alive’ if it is able to connect. The peer_receive thread receives a request from the peer server and creates a child thread.

Whenever a peer server is detected as dead, peer_attributes is updated with the status of the peer server. The thread that is receiving requests from the respective peer server is terminated.

So, the number of peer receive threads will eventually be equal to the number of peer servers set ‘alive’ in peer_attributes.

Invariant 4: No server joins the group when a two-phase commit that has been initiated by a server for serving client request is in effect.

When a new server that wants to join the group sends a connection request, two phase commit is initiated. The new server joins the group after two phase commit succeeds. When this two-phase commit is going on, no other client request is sent to other servers. New server joining request and client request are processed in a sequential fashion by the peer_send thread. So, when a client request is served, no new server joins the group.

When no other peer servers are alive, requests are served immediately without consulting peer servers. When a new server joins, two phase commit is not necessary. In this case, two phase commit is not necessary for both new server joining and serving client requests.

Invariant 5: Any thread that holds the lock does not block.

No thread does an infinite wait after acquiring a lock. Also, no thread cancels itself while holding a lock. So, threads do not block while holding the lock.

5.3 Properties

Theorem 1: Client requests are eventually served if the mutexes are starvation free and at least one server is alive and no server crashes.

Proof: If no peer server is alive or a request can be served without consulting peers, the client request is immediately served.

If peer servers are alive, they have to be consulted before a request is served. The client_receive thread that processes the client request adds the request to the client_request_list. A condition variable that waits on the status of two-phase commit is set. This is a timed wait.

The implementation of two-phase commit involves a 'select' system call with a timeout. If it does not get positive responses from the peer servers before it times out, the transaction is aborted. So, two-phase commit returns either commit or abort. The condition variable corresponding to the client request is signaled and the status of two-phase commit is also updated.

The client_receive thread does not block. So, each request is served or an error message is sent to the client.

Q.E.D.

Theorem 2: When there is a perfect failure detector and there are no network failures, the state of the persistent store including current state and pending commits, will be the same in all servers that are alive.

Proof: The current state of the server reflects the contents of the store with existing namespaces and objects. Pending commits refers to the namespaces and objects that are locked for committing an operation, but operations are not actually performed.

Operations on the persistent store, that modify the state of the store, are performed after a successful two-phase commit. During the first phase of two-phase commit, namespaces and objects on which operations will be performed are locked. During the second phase, if the operation is to be committed, the operation is performed and locks are released.

Q.E.D.

Theorem 3: Consistency is guaranteed by the two-phase commit protocol. Operations on the persistent store are linearizable.

Proof: The two-phase commit algorithm guarantees linearizability. Linearizability is a local property. If each individual object in a system is linearizable, then the entire system is linearizable. The operations on a single object replicated in many servers should follow some sequential order. The order should be the same in every server.

Before performing any write operation on an object, a lock should be obtained. After completing the operation, the lock is released. So, there is no possibility for concurrent writes in a server. This makes sure that every server views sequences of 'write' operations on the objects. When a server receives a request for locking an object that has already been locked, ties are broken on the basis of IP address of the server that initiated the transaction. Since this is done in all the servers, every server performs 'writes' on objects in the same order even if the order in which a server

receives ‘write’ requests is different from a peer server. So, the sequence of ‘write’ operations performed is the same in all servers. This guarantees that the system is linearizable.

Q.E.D.

5.4 Limitations

Limitation 1: Network partitions

Network partitions occur when servers that are considered to be ‘dead’ are actually ‘alive’. Servers in one partition are not aware of the events in another partition. Client requests are processed in a manner such that inconsistencies may arise in the system.

Limitation 2: Denial of service attacks.

Denial of service attacks occur when servers are overloaded with requests from clients. Clients are compromised and requests are sent with the sole objective of impairing the performance of servers.

Limitation 3: Two-phase commit protocol may block.

The two-phase commit protocol may block when server crashes happen during inopportune moments. If the server initiating two-phase commit crashes after getting replies from peer servers, objects and namespaces that would have been locked during the first phase will remain so. If other servers crash, it will be detected by the

server that initiates the requests. This is a limitation of the two-phase commit protocol that may affect the system.

Chapter 6

Conclusions and Future work

6.1 Conclusions

A persistent storage architecture for the ACE environment has been designed and implemented. The services offered by the servers to the clients are explained. The properties that are guaranteed by the servers are listed and explained. The assumptions regarding software packages used and the mode of communication over the network that are necessary for these properties to be true are explained.

6.2 Contributions

This work forms part of the ACE project. The persistent storage architecture that we have designed and implemented will meet the long-term storage requirements of the ACE environment. We have proved the properties of the system that we have implemented. The experiments that we have done by varying the sizes of the objects and the number of objects in the namespaces indicate that the time taken for servers to respond to various client requests is reasonable.

6.3 Future work

The persistent storage architecture can be implemented with different network protocols and different consistency models. Also, features that provide encryption and

decryption of data transmitted and user authentication features to access services can be added.

6.3.1 Different network protocols

The implementation for this work has been done using the TCP/IP protocol. User-level network protocols are based on transferring functionalities provided by traditional protocols from kernel level to user level. Latency in communication between the two endpoints is the sum of system latency and wire latency. Of these, latency in the sender and receiver machines proves to be a limiting factor in performance. User level network protocols provide better performance when compared to traditional protocols. M-VIA [8], a user-level network protocol based on 'Virtual Interface Architecture' [9] model, gives better performance than TCP and UDP. The experiments we conducted substantiate this claim [10].

6.3.2 Different consistency models

The consistency model that we have implemented is linearizability. Weaker consistency models [11] generally give better performance, though the perceived notion of correctness of operations is blurred. A weaker consistency model such as 'causal consistency' can be implemented for the same architecture [12].

6.3.3 Security issues

Communication between the client and the server and among the servers is based on insecure TCP/IP. Also, there is no mechanism to authenticate the clients before accessing services.

A secure transmission protocol can be used instead of TCP/IP. Encryption and decryption can be performed for every data transfer. Encrypted data can be stored in persistent store. For encryption and decryption, Diffie and Hellman algorithm can be used. Authentication mechanisms can be used for clients before accessing services from the servers. The hierarchy of persistent store consisting of namespaces and objects can include different security and authentication mechanisms for different types of namespaces.

References

[1] Maurice P. Herlihy and Jeannette M. Wing, “Linearizability: A Correctness condition for Concurrent Objects”, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 12, no. 3, July 1990, pp. 463 – 492.

[2] Leslie Lamport, “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”, IEEE Transactions on Computers vol. 28, no. 9, Sept. 1979, pp. 690-691.

[3] Andrew S. Tanenbaum, “Distributed Operating Systems”, pp. 169-178.

[4] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao, “The Ninja Architecture for Robust Internet-Scale Systems and Services”, Computer Networks: the International Journal of Distributed Informatique, vol. 35, no. 4, March 2001, pp.473-497. (*Special Issue on Pervasive Computing*).

[5] M. Ogg, G. Obertelli, F. Handfield, and A. Ricciardi, **“**Nile: Large-scale distributed processing and job control”. Proc. Int'l Conf on Computing in High Energy Physics, Chicago, IL, September 1998.

[6] Mustaque Ahamad, Rammohan Kordale, “Scalable Consistency Protocols for Distributed Services”, IEEE Transactions on Parallel and Distributed Systems vol. 10, no. 9, 1999, pp. 888-903.

[7] Marios Mavronicolas, Dan Roth, “Linearizable Read/Write Objects”, Theoretical Computer Science, 220(1), 267-319 (1999).

[8] “M-VIA, a modular high-performance implementation of the Virtual Interface Architecture for Linux”, http://www.nersc.gov/research/FTG/via/.

[9] Don Cameron and Greg Regnier, “The Virtual Interface Architecture”, Intel Press.

[10] “Performance comparison of TCP, UDP and M-VIA”, ITTC Technical Report.

[11] Raynal, M. and Mizuno, M. “How to find his way in the jungle of consistency criteria for distributed shared memories (or how to escape from Minos' labyrinth)”, Proc. of the 4th IEEE Workshop on Future Trends of Distributed Computing Systems, pp. 340-346, 1993.

[12] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, Phillip W. Hutto, “Causal Memory: Definitions, Implementation, and Programming. Distributed Computing”, 9(1), pp. 37-49, 1995.

Appendix

Test results

The experiments were conducted with machines in the ITTC network. The machines that act as servers are

1. alnitak (800 MHz dual processor, 256 MB RAM, SCSI disk)
2. tomato (800 MHz, 256 MB RAM)
3. bubblegum (800 MHz, 256 MB RAM)

The response time for the client from the machine ‘tomato’ is calculated for various object and namespace commands. The requests are processed by the server ‘alnitak’. For operations that involve writing to the persistent store, ‘alnitak’ sends the requests to the other two peer servers before responding to the client.

The operations that are performed without consulting peer servers are Retrieve Object and List Namespaces.

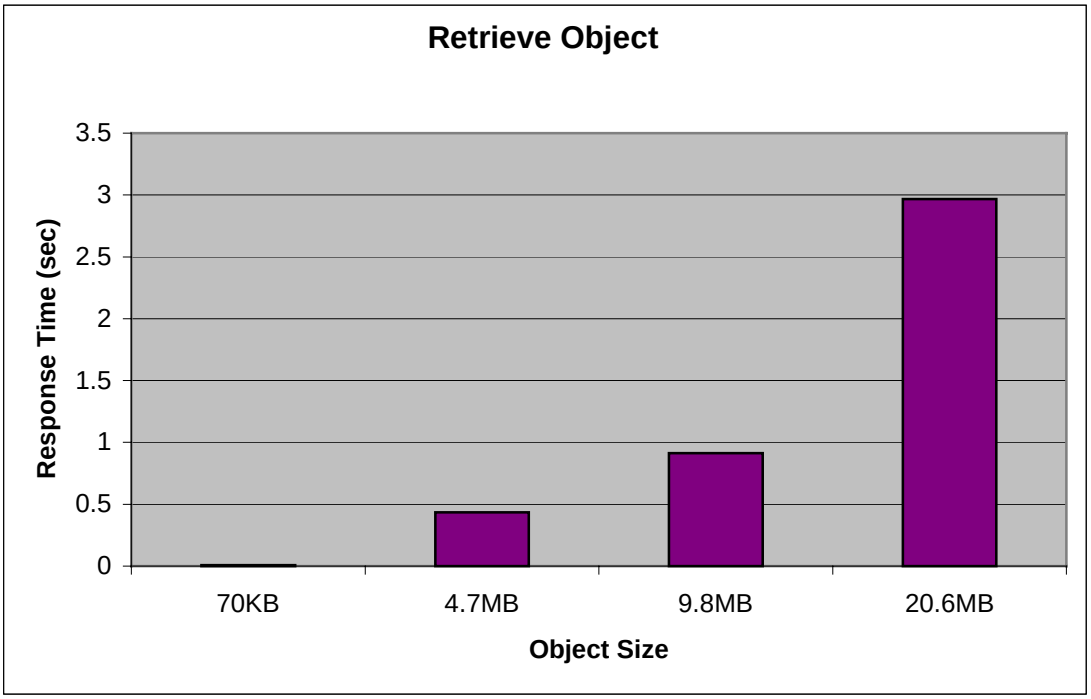
The operations that are performed after consulting peer servers are Store Object, Create Namespace, Clear Namespace and Delete Namespace.

The client requests processed and the corresponding response time are listed below.

i) Retrieve Object

The objects that are retrieved from the store include an image file of size 70KB and mp3 files of sizes 4.7MB, 9.8MB and 20.6MB. The response time for retrieving objects of different sizes are as follows.

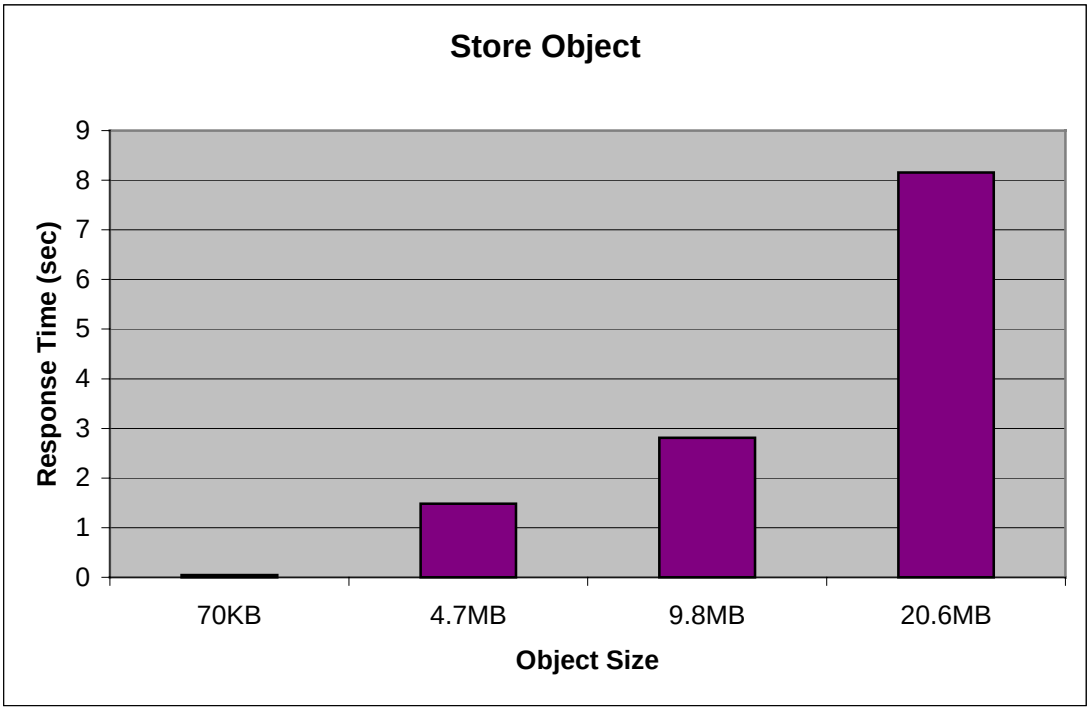
Object size	70KB	4.7MB	9.8MB	20.6MB
Response time (sec)	0.00898	0.43588	0.914212	2.967225



ii) Store Object

The objects that are stored are the same as the ones listed above. The response time for storing objects of different sizes are as follows.

Object size	70KB	4.7MB	9.8MB	20.6MB
Response time (sec)	0.042842	1.480014	2.81178	8.155889



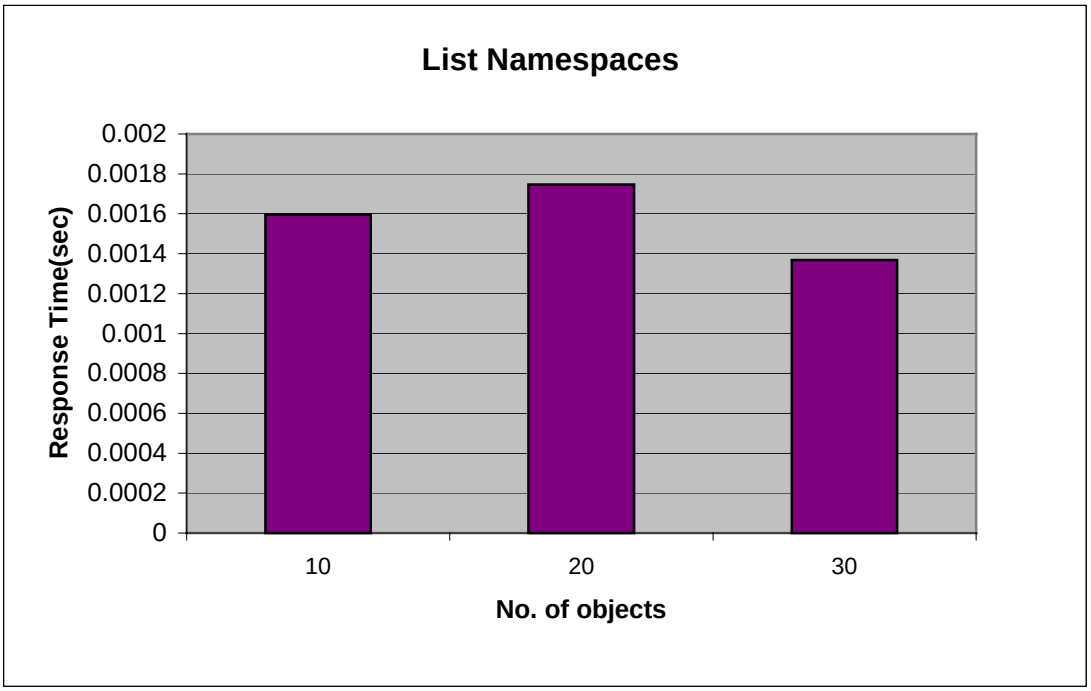
iii) Create Namespace

The average time taken for creating a namespace in this experimental setup is 0.018212 sec.

iv) List Namespaces

The time taken for listing different number of namespaces is shown here.

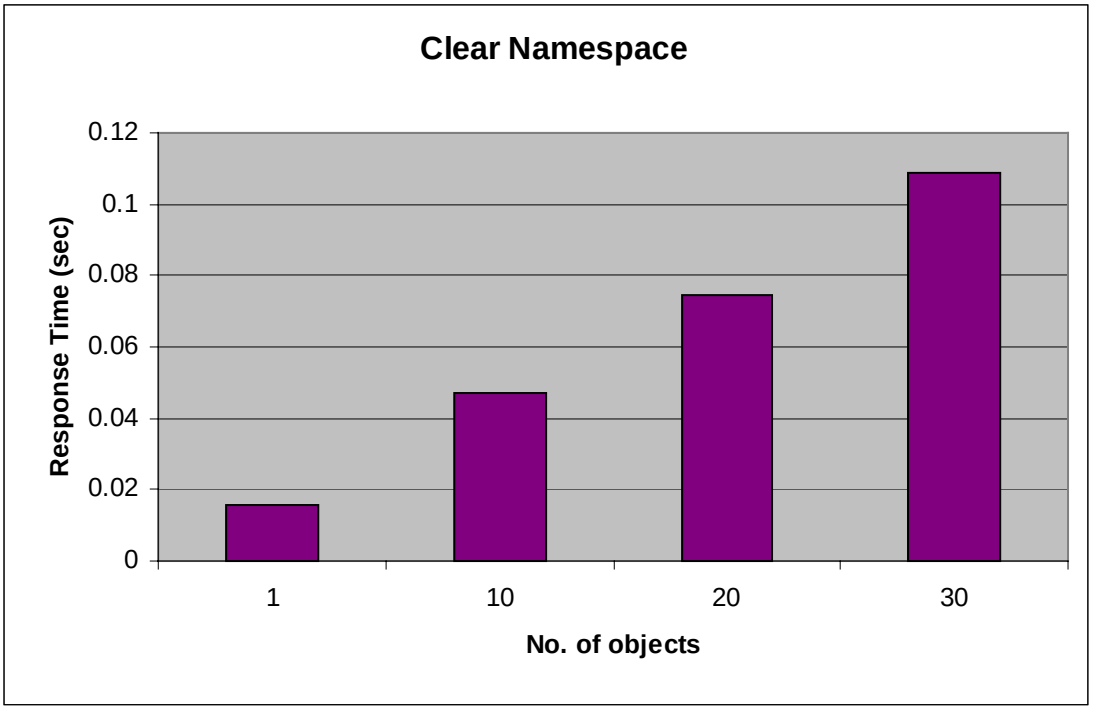
No. of objects	10	20	30
Response Time (sec)	0.001596	0.001747	0.001368



v) Clear Namespace

Clearing a namespace is tested with varying number of objects in the specified namespace. The number of objects in the namespace and the combined sizes of all objects are listed below.

No. of objects	1	10	20	30
Combined size of objects	32KB	44.976MB	85.840MB	134.920MB
Response time (sec)	0.015803	0.046947	0.074643	0.108779



vi) Delete Namespace

Deleting a namespace is tested with varying number of objects in the specified namespace. The parameters used for this test are the same as the ones used for clearing a namespace. The number of objects in the namespace and the combined sizes of all objects are listed below.

No. of objects	1	10	20	30
Combined size of objects	32KB	44.976MB	85.840MB	134.920MB
Response time (sec)	0.0169539	0.0454645	0.0730274	0.1026457