M.I.T. Laboratory for Computer Science

Computer Systems Research Division

June 9, 1980

Request for Comments No. 188

Recovery of a Repository in a Distributed Data Storage System

Gail C. Arens

This document is my Master's thesis proposal.

Brief Statement of the Problem:

Repositories provide reliable data storage for the nodes participating in the Distributed Data Storage System (DDSS) that is being built based upon Reed's object model. In order for DDSS as a whole to be able to guarantee a high level of reliability it must require that its individual repositories behave correctly at all times, even after a crash. This thesis will define correct behavior and then describe how a repository recovers its data . It will also describe how a repository's processes and communications interface are designed so that their state prior to a crash doesn't have to be recovered in order for the repository to resume normal activity.

How about undo/appl. recovery / checkpoint (appl. makes an error this is not [illegible] [illegible] [illegible])

1. Introduction

As network communications become faster and cheaper it becomes more practical for a single node in a distributed computing network to maintain only the resources that it can afford. It can get all other resources that it may need through the network from other nodes that provide them. Thus the network provides the benefit of economy of scale through sharing. Storage is one type of resource that should be shared throughout the network. These nodes exist for the sole purpose of providing high speed, high volume storage that has a high level of reliability. This gives the user nodes the option of maintaining only a small amount of inexpensive, less reliable storage to serve as a cache for the long term storage residing in the external storage server nodes [Isra78]. These nodes which are dedicated to the storage functions we call repositories. Together, the collection of repositories in the network must provide a coherent storage system regardless of whether or not a user's data is stored entirely on one repository or distributed throughout a random number of them. The distributed data storage system that this thesis is concerned with is called DDSS.

DDSS uses Reed's model for its storage objects and uses a mechanism that he developed for synchronizing all accesses to those objects [Reed78]. This mechanism allow DDSS to guarantee a high degree of reliability in maintaining the integrity of user data. In order for DDSS as a whole to be able to guarantee this reliability it must require that its individual repositories meet up to certain system standards at all times. A repository must always provide a uniform interface to the other nodes of DDSS even after it crashes. The purpose of this thesis is to describe how the components of the repository are designed so that even in the event of their failure the repository will always meet DDSS's requirements.

The components of the repository are its storage, its communications interface, and its processes. The storage is the most important component of the repository. The processes and communications exist solely for the purpose of transmitting the data between the user and the storage medium. Therefore the user data must always appear consistent at the repository - DDSS interface. When the repository fails it must be able to recover its components to a state in which they will still meet DDSS standards. The storage mechanisms are designed so that the user data can always be restored to a consistent state. The communications and most of the processes are designed so that they can simply be restarted with no memory of the state that they may have had when the repository failed.

In my thesis I will first outline the requirements that the repository must meet when it recovers from a crash and then I will show how its components meet those requirements. I will describe how the repository recovers the user data in storage. Then I will have to describe how the processes are designed and how the communications interface works in order to be able to justify throwing away their state after a crash.

2. DDSS

DDSS is supported on a local network of microprocessors. Some of these microprocessors function as personal computers and support the applications that utilize DDSS. Others function as servers that provide a particular resource to the entire network [Swin79]. A repository is a server. It is a microprocessor that is connected to a configuration of storage devices. The applications don't directly communicate with the repository. They communicate their storage needs to a lower level abstraction called a broker which resides in the personal computer. The broker decomposes these needs into requests that are comprehensible to the repositories.

The structure of and access to user data is based on the mechanisms described in [Reed78]. I will summarize these mechanisms with respect to DDSS for those readers who are not familiar with them. The functional unit of data in storage is called an object. Every time an object is updated a new version of that object is created. The collection of versions for a single object is called an object history. DDSS maintains the entire history of all its objects. Every version of an object is valid from its time of creation to the time of the next version's creation. There can only be one version in an object history for any given time. In order to access an object an application must specify a time and then it will be given the version that is valid at that time. There are never any holes left in an object history because when a new version is created at a specified time, the previously current version's time of validity is extended to fill up any vacant space.

In DDSS, time is a relative term and does not directly correspond to real time. It is used to order the events occuring in DDSS. There is a global clock mechanism that supplies the present value of DDSS time to any application that request it. The global clock is an ever increasing counter that is incremented after each time it hands out a timestamp. Therefore timestamps are always unique and non-decreasing, yet correlated. Whenever an application wishes to create a new version in an object history, i.e. update the object, it must specify a timestamp which it obtains from the global clock. This timestamp designates the time at which the new version should be created. If a version already exists for that time then the application can't update the object at that time and must try again at another time. This timestamping mechanism provides the basis for synchronizing applications that run concurrently.

DDSS provides applications with the ability to perform a series of updates as a single atomic operation [Lamp79] [Lind79] [Reed78] [Svob79]. This means that the intermediate state of the system (when only some but not all of the updates have been done) should never be seen by a process performing the operation. If there is a failure and not all of the updates can be done then the system's state should be backed up to the state it had before any of the updates were done. Another implication of performing an operation atomically is that it should not be affected by any other atomic operations that are being done in concurrently; that is, all other atomic operations behave as if they precede or follow it.

In DDSS, the atomic operations done at the user application level are called atomic actions. Every atomic action is given a globally unique timestamp and all updates within that transaction create versions at that given time. If a atomic action is unable to update any of its objects then it is aborted and must obtain a new timestamp and try the entire atomic action over again. This mechanism prevents applications from interfering with each other and essentially serializes any atomic actions that involve concurrent accesses to the same objects. It also guarantees that if the atomic action is aborted then none of the objects will be updated. It can guarantee this because it only makes the updates tentatively until it knows whether or not the atomic action completed. A tentative version in an object history is called a token. A record of these tokens is kept for each atomic action so that they can be deleted if the atomic action aborts. These records are called commit records. Commit records are stored in the repositories.

DDSS does not care whether the objects updated by an atomic action are entirely contained in a single repository or distributed throughout a random number of them. The brokers manage the atomic actions and make sure that the requests are directed to the proper repositories. Once the repositories receive the requests, they know how to carry them out in synchronization with each other when handling requests for the same atomic action. They have the ability to communicate with each other if it is necessary in order to coordinate their efforts.

3. Repository Requirements

In order for DDSS to guarantee a high level of reliability it requires that its repositories always conform to two specifications regardless of any failures. They must perform all requests atomically and they must protect the integrity of the user data that they maintain in storage. Both of these requirements are closely related because if the requests are not performed atomically then the data integrity cannot be preserved. In the event of a failure the recovery process must mask the effects of partially completed requests and must repair any damaged data.

The repository communicates with the other nodes in DDSS by sending and receiving messages. These messages contain either requests for the repository or responses to requests that the repository sent. The repository is a passive node in the network and only generates requests if they are relevent to the fulfillment of requests directed to it.

In order to fulfill the requests sent to it, a repository usually performs a series of tasks. In order for DDSS to function properly these tasks must be done atomically; either all or none of them must be done. If the repository crashes in the middle of satisifying a request then it must never let the requestor see its intermediate state. It must make this state consistent from the requestor's perspective before resuming its normal activities.

Since the local network supports limited sized packets, messages may be too large to be sent within a single packet. For efficiency reasons, the repository begins to process the request contained in a message before it receives all of the packets of the message. Therefore it may write the data for a single object version into storage in bits and pieces. If the repository crashes before receiving the all of the data then it must either delete the partially written version or finish writing the remaining fragments out to the disk.

In the process of fulfilling a single request, modifications may be made to more than one object. These modifications must also be done atomically. If a repository crashes in the middle of processing a request it must either delete all modifications that have already been made as a result of the partial processing or make the remaining modifications necessary for the complete processing of the request.

The repository protects the integrity of the actual data bits by keeping the data in *stable storage* [Lamp79]. This means that, from the users' perspectives, the data kept in stable storage will never be lost or damaged. In actuality, if a crash occurs it can cause some of the data to be damaged but there is enough redundancy and relevent information maintained in the stable storage that enables the repository to recover it before the users access it.

A requestor can never be sure that its request was attended to unless it receives an acknowledgement from the repository. Once it receives the acknowledgement, though, it has the repository's guarantee that the request was handled correctly and all of its data is in stable storage. The repository must never acknowledge any request before it correctly completes all of the necessary tasks and puts the user data in its stable storage.

4. Data Recovery

The repository has two types of data storage, volatile and stable. Physically, stable storage consists of 2 sets of append only optical disks. Of course only the most current disk in each set is kept on line. Each set contains a complete copy of all the object histories stored in the repository. All stable storage data is stored in the form of versions within object histories.

There are three categories of data that must never be lost. The repository exists for the sole purpose of reliably storing user data, so of course user data must be kept in stable storage. Commit records must never be lost so they too are treated as objects and kept in stable storage. Finally, the repository itself keeps some of its own information that it may need for recovery in stable storage.

Volatile storage consists of a reusable magnetic disk. It serves two purposes. For one, it would be silly to store short term information redundantly in stable storage since optical disks are write once only and have slow access times. Secondly, since optical disks are so slow the volatile disk storage can serve as a cache for the data in stable storage that is frequently accessed. For example, for each object history, an aggregate of all data relevent to that object history as a whole is maintained in volatile storage. Then whenever a modification is made to an object history the repository doesn't have to search through the object's versions to get this data. This aggregate is called an object header. Object headers are only hints because the correct functioning of the repository is not dependent upon them [Verh77]. If a crash occurs and damages the object headers they can be reconstructed from stable storage.

After a crash, the data in storage must be recovered at a number of different levels. First of all if the bits are damaged then they must be restored. Second, the versions in stable storage should be formed into coherent object histories. Third, the repository must restore its volatile storage to the most current consistent state that can be extracted from stable storage. Finally, the objects and commit records must look consistent from the atomic actions' perspectives.

Recovery is not as straight forward as it may seem. Complications arise due to the fact that the optical disks are write once. Versions in stable storage are supposedly ordered and have various pointers to one another. Any time something has to be restored it has to be copied into a new address and since pointers can't be changed, the versions that point to it may also have to be copied. But how do you know which versions point to the one which has just been copied? Also, does a version take on a new end time of validity if it is rewritten during the recovery process? What other complications arise due to various unique characteristics of the repository and how are they handled?

Another factor that must be taken into consideration is performance. It is important to minimize the down time of the repository since external nodes' activities may be dependent upon the information stored in the repository. Therefore the recovery process should take as little time as possible. This will influence the design of the recovery process.

5. Communications Interface

Brokers and repositories communicate their requests through messages. All requests that are sent are acknowledged with a response from the receiving node. A response is the requestor's guarantee that its request was satisfied, (unless it contains an error message). Each request - response pair of message is given a unique id so that nodes can easily determine which request is being acknowledged.

Messages are sent through the network in the form of packets. The low level processes that send and receive these packets use the following protocol. The sending process can only send the first packet of any message until it receives an acknowledgement from the receiving process. This acknowledgement contains a number indicating how many more packets the sending node is allowed to send. After sending this next number of packets, the sender must again wait to find out how many packets the receiver will accept in the next sequence. The sender can never send more packets of a message until the receiving node gives the go ahead. This gives the receiving node some control over the number of packets sent to it versus the amount of free space it has available to buffer the packets.

When a repository crashes it can throw away all messages that it is in the middle of processing as long as it does not leave its long term data in an inconsistent state and does not cause errors in the requestors' routines. The stable and volatile storage recovery process returns the data to a state that is consistent from the requestors' perspectives. Also, because the messages are thrown away and never acknowledged they appear to the requestors as lost messages. Provided that no effects of the partial processing of the disposed messages show through the repository interface, the continuity of the requestors' routines should not be destroyed since they are designed to tolerate lost messages. They use time outs to decide when to give up on a transmitted message and either retransmit it or take alternative measures.

There are two pathological situations that may arise and create problems. First, a requestor can not tell whether or not a repository has crashed or is overloaded and therefore responding very slowly. In either case the requestor may retransmit after its time out period has expired. The repository must guarantee that it won't process the same request more than once.

Also, some requestors may not notice that a repository has crashed before that repository resumes its activity. They may still be sending packets that belong to a message that was thrown away during recovery from the crash. In order for a repository to function properly and efficiently it must be able to distinguish between those packets and the packets that are part of new messages. Otherwise it will waste some of its limited buffer space by holding these packets while awaiting the arrival of the packets of the message that are missing. These packets may never be sent since they were already sent prior to the crash (but were thrown away during recovery).

In my thesis I will explain how the combined use of the message id's and the packet protocol takes both of the above situations into account even though all partially processed messages are thrown away after the repository crashes.

6. Repository Processes

There are four types of processes that run concurrently within the repository. There is a single master process that oversees all activities going on in the repository. It spawns off all of the other processes and delegates certain responsibilities to each of them. There is a single process that interfaces directly with the network and participates in all low level network protocols. This process is called the netdriver. There are a number of processes, called workers, that actually carry out the requests that are sent to the repository. When the master receives a request it hands it off to one of the workers which will do whatever repository level tasks are necessary in order to satisfy the request. Finally, there is at least one daemon process which periodically modifies parts of the data in storage.

In my thesis I will explain how these processes are designed so that if they can't complete the task that they are performing (because of a crash) they can be restarted rather than restored to the state they were in before the crash. Any user data that they affected will be cleaned up by the recovery mechanisms and any messages that they were handling will be treated as if they were never received.

7. Related Work

There are various other systems comparable to DDSS that have different underlying mechanisms for providing a coherent and reliable distributed data storage system [Bern79] [Gray79] [Lamp79] [Paxt79] [Swin79]. At a glance they all look very similar but in actuality they provide varying levels of reliability, have different models of user data and implement atomic actions differently.

SDD-1 [Bern79] provides more reliability than DDSS by maintaining more than one copy of all user data at more than one node. WFS [Swin79] doesn't provide the applications with the ability to perform atomic actions but it does guarantee the atomicity of its own operations. The system described in [Paxt79] differs from DDSS in that it does not recover all of the data immediately after restarting but instead, it recovers each atomic action when the first access (after recovering from the crash) is made to some of its affected data.

With regards to the synchronization of atomic actions, each of the mechanisms used in these systems are different from the one used in DDSS. SDD-1 statically categorizes the atomic actions as to their interference characteristics and provides different levels of synchronization for each category at run time. System R [Gray79] and the systems described in [Lamp79] and [Paxt79] all use locks to prevent atomic actions from interfering.

There are many other shades of differences which exist between these systems and DDSS and also some similarities, but they will not be discussed any further in this proposal. The systems all have to provide different types of recovery mechanisms due to their differences in goals, models and implementations. The most unique feature of DDSS that is relevent to the repository's recovery scheme is the way it models user data. The object history structure must always be restored after a repository fails. The final implication is that my thesis will combine both new and old ideas in order to provide a single coherent recovery procedure for repositories within DDSS.

8. Schedule of Tasks

April: Design DDSS-repository communications interface and repository processes.

May-June: Write proposal; develop data recovery mechanisms; do most of the programming.

July-August: Finish up the programming; write thesis.

9. References

[Bern79] Bernstein, P.A.; Shipman, D.W.; Rothnie, J.B., "Concurrency Control in SDD-1: A System for Distributed Databases; Part 1: Description," Computer Corp. of America, Report CCA-03-79, Cambridge, Ma., January, 1979.

[Gray79] Gray, J., et al., "The Recovery Manager of a Data Management System," IBM Research Laboratory, Research Report RJ2623 (33801), San Jose, Ca., August, 1979.

[Isra78] Israel, J.E.; Mitchell, J.G. and Sturgis, H.E., "Separating Data from Function in a Distributed File System," *Proceedings of the Second International Symposium on Operating Systems*, IRIA, October, 1978.

[Lamp79] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Reasearch Center, Ca. April, 1979. To appear in *CACM*.

[Lind79] Lindsay, B.G., et. al., "Notes on Distributed Databases," IBM Research Laboratory, Research Report RJ2571 (33471), San Jose, Ca., July, 1979.

[Paxt79] Paxton, W.H., "A Client-Based Transaction System to Maintain Data Integrity," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December, 1979, pp. 18-23.

[Reed78] Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," M.I.T. Laboratory for Computer Science Technical Report TR-205, September, 1978. (Also Ph.D. thesis, Department of Electrical Engineering and Computer Science, M.I.T., September, 1978.)

[Svob79] Svobodova, L., "Reliability Issues in Distributed Information Processing Systems," *Proceedings of the Ninth IEE Fault Tolerant Computing Symposium*, June, 1979, pp. 9-16.

[Swin79] Swinehart, D.; McDaniel, G.; Boggs, D., "WFS: A Simple Shared File System for a Distributed Environment," *Proceedings of the Seventh Symposium on Operating Systems Principles.* December, 1979, pp. 9-17.

[Verh77] Verhofstad, J., "Recovery and Crash Resistance in a Filing System," *Proceedings of the ACM-SIGMOD Conference on Management of Data*, August, 1977, pp. 158-167.