Non-Disruptive Server Recovery in Calypso File System

Murthy Devarakonda, Bill Kish, and Ajay Mohindra

IBM Thomas J. Watson Research Center, Hawthorne, NY.

Abstract: This paper presents design, implementation, and measurement of a recovery scheme for stateful file servers in a cluster-optimized, distributed Unix file system called Calypso. The recovery scheme is non-disruptive, meaning that the open-files remain open, un-sync'ed data is not lost, and in-flight operations are properly handled across server recovery. The scheme relies on naturally replicated file system state among the clients. It uses a three-phase protocol to reconstruct the server state either on a back up node (if disks are multi-ported) or on the rebooted server node. Measurements show that the time to reconstruct the server state, for a 10-node cluster with each node actively referencing 112 files, is about 1.5 seconds. However, the total recovery time is dominated by the log redo of the underlying journaling file system (about 15 seconds), which is bounded by the log size.

1 Introduction

This paper describes design and implementation of a recovery scheme for stateful file servers in a cluster-optimized, distributed Unix file system called Calypso. The recovery scheme is based on a recovery protocol between the clients and servers, and it relies on naturally replicated state among clients thus avoiding explicit replication. The recovery system is non-disruptive meaning that the file opens and modified file data at the clients survive across server recovery, and in-flight client-server operations are completed after recovery. Furthermore, the recovery scheme exploits multi-ported disks so that another processor can take over the disks and function as the file server resulting in short recovery times and system re-organization capability.

In previous distributed file systems, server fault tolerance is provided mostly through replication, primary/back-up, RAID, internal fault-tolerance, or fast restart. The use of naturally replicated state among clients has the advantage of simplicity and does not have the overhead of additional resources. In some of the previous systems, server recovery is disruptive of client state requiring additional handling in applications, and in some others, servers are stateless, thus simplifying server recovery at the cost of poor performance and scalability. Non-disruptive recovery of stateful servers provides scalability and high-availability.

The recovery schemes in the Sprite file system [1] [2] and Highly Available Network File Server (HA-NFS) [3] [4] are closely related to the work described here. HA-NFS supports NFS, and since NFS servers are stateless except for byte-range locks and most recent 'dupcache' entries, HA-NFS server recovery scheme involves hardening and reconstruction of this limited state using multi-ported disks. Calypso and Sprite recovery schemes differ in the recovery protocol and state representation. Furthermore, Sprite has not used multi-ported disks. In particular: Calypso recovery protocol has three phases with synchronization between phases, which guarantees the integrity of files across server recovery while allowing concurrent transfer of state from clients; Calypso's token-based state representation of client caches allows compact and rapid transfer of state from clients to the new server; Calypso's multi-ported disk support allows short recovery time and data availability in spite of a permanent processor failure or scheduled shutdown.

For the past two years Calypso file system has been developed and tested at product quality for possible use as an internal file system in RISC System/6000 clusters such as SP [5] and HACMP [6]. SP is a multicomputer, consisting of up to 512 independent RISC System/6000 processors, interconnected using an optional high speed switch. Individual processors are off-the-shelf systems with local memory, disks, and other interface cards. Rack-based packaging of SP provides redundant power supply and monitoring capability. HACMP is a relatively small (up to 8 processors) cluster of independent, off-the-shelf RISC System/6000 processors with fiber connectivity and multi-ported disk capabilities. SP is used mainly in technical computing, although it is gaining ground in the commercial area. HACMP is used highly available commercial applications.

As a cluster file system, Calypso takes advantage of homogeneity, special hardware, peer relationship among processors, and single administrative domain to provide small and efficient implementation offering strong cache consistency and other POSIX semantics, reduced false sharing in simultaneous update of large files, NFS support to external clients, and non-disruptive server recovery. The cluster aspects of Calypso relevant to server recovery are peer relationship among processors, single administrative domain, and multi-ported disk support. The peer relationship and single administrative domain allow us to trust all our nodes, and use a distributed recovery protocol without being concerned about serious security violations.

In Calypso, as in other distributed file systems such as NFS [7] and AFS [8], one node functions as the server for each file system and the rest function as clients. The server node has a physical attachment to the disks with file data. Even though multi-ported disks are physically attached to two or more nodes, only one port is active at any given time. Clients cache file data extensively and cache consistency is maintained using a token-based scheme. A client must first obtain necessary tokens before carrying out a file operation, and acquired tokens remain with the client until revoked. A token not only authorizes caching and access to a file or file attributes but also opening of a file in a certain mode or use of a cached reference to a file. The server maintains the state of all tokens, and each client maintains information about tokens it is presently holding. Server recovery consists of reconstructing the state of tokens at the back up node (or even at the same server node after reboot) using the aggregate information maintained at the clients. To a user, server reconstruction is completely transparent. Therefore, except for a small response time delay, a user is unaware of the server failure and recovery. Recovery from disk failures is beyond the scope of Calypso. We expect RAID devices or disk mirroring (replication) to provide the necessary function.

A node status service [9], which is external to Calypso, detects failures and initiates recovery. The node status service maintains 'liveness' status using selective 'Are you alive' messages and a quorum election algorithm. When this service detects failure of a server node it starts Calypso recovery protocol through a leader node. In the first phase, all cluster nodes are informed of the server failure and the alternate server (pre-determined) takes over the disk(s) through a secondary port, checks consistency of the file system(s), and performs file system mounts. In the second phase, all clients send in their state information. Clients are informed of the completion of recovery in the third phase, and they may now access the server as needed. In-flight operations are handled using retries and a check-before-request approach as described in Section 6. Measurements of reconstruction times, which are presented in Section 5, show good scalability. The major portion of the recovery time is in the log-redo of the AIX Journaling File System. Fortunately, the log redo time is proportional to the size of the log and hence, after a certain point, it does not increase with the number of clients.

The rest of the paper is organized as follows. The next section discusses the related work, especially the recovery schemes in Sprite and HA-NFS. Section 3 presents background on Calypso as relevant to this paper. Section 4 details the recovery protocol. Section 5 discusses handling of in-flight operations. Section 6 presents measurements of recovery times. Lastly, Section 7 concludes the paper.

2 Related Work

The closest related work is the Sprite's recovery scheme [1] [2]. Both Sprite and Calypso use naturally replicated state among the clients to recover the server state. However, differences in the protocol, state representation, and failure and re-join detection distinguish the Calypso design point. The Calypso design offers the following advantages: guaranteed cache consistency even during the state reconstruction, state transfer in few messages, and a consistent view of the cluster membership derived from formal membership protocols. Furthermore, Calypso supports multi-ported disks offering the ability to treat processor and disk failures separately for planned reconfiguration and handling of permanent processor failures.

In Sprite, each client detects server failures and re-joins independently, and also reconstructs its part of the server state independently. In Calypso, server failure and re-join detection is based on a rigorous, scalable protocol that maintains consistent, mutually agreed cluster membership, even if the network is partitioned. In addition, Calypso uses a multi-phase recovery protocol with synchronization between the phases. Clients independently and simultaneously transfer their state, but because of the inter-phase synchronization they do not reference the common state until reconstruction is complete. Sprite's independent failure detection and recovery protocols allow the possibility of a client accessing a file even before the file state is completely reconstructed.

In Sprite, each client maintains the state of local file opens, byte-range locks, and cached blocks, and the server maintains the global state [10]. In a larger sense, this is similar to Calypso except that in Calypso the state is represented using file-system independent tokens that allows us to transfer the client state to the new server compactly. For example, near the end of the Andrew benchmark [8], a Calypso client typically caches information about nearly 112 files. But since a token can be represented in 40 bytes, each Calypso client transfers its state in about four messages of 8K-bytes long.[1] In Sprite, server congestion and recovery storms were reported [11] because clients performed a single RPC for each open file and cached blocks. Also, by batching the token table updates from a client, Calypso reconstructs the server state quickly.

Because of the multi-ported disk usage, Highly Available Network File Server (HA-NFS) [3] [4] has relevance to the Calypso recovery scheme. However, since HA-NFS supports only the NFS protocol, HA-NFS deals with the recovery of a stateless server except for the (byte-range) lock state and 'dupcache' entries of the most recent few seconds. The dupcache is a mechanism to detect duplicate UDP messages for non-idempotent operations at the server. HA-NFS uses the JFS log, which is written to a partition on the multi-ported disk, for storing this state. HA-NFS recovery is driven by the need to handle existing NFS clients unchanged. For this reason, in part, HA-NFS also provides IP address take over. Calypso does not store any state on the disk to avoid additional disk traffic in the normal operation. Since Calypso is a stateful file system, an entry in the equivalent of the dupcache may have to be maintained for ever (or nearly so) in the worst case, and this makes JFS log truncation complex if Calypso were to use JFS log for this purpose. Therefore, we handle in-flight operations using a combination of retries and a check-before-request scheme.

The recovery scheme proposed for Spritely NFS [12] is similar to that of Sprite. The disconnected operation in Coda [13] and replication in Ficus [14] can lead to conflicts. In addition, explicit replication at another server (as in Deceit [15], for example) or Zebra-style striping [16] require extra resources and treat server nodes and disks as single entities.

3 Background on Calypso

Figure 1 shows the architecture of the Calypso file system. Calypso is made up of three distinct subsystems: (1) The token manager comprising of token clients and token servers, which are shown as the shaded areas in Figure 1; (2) The virtual file system implementation comprising of data clients and servers, which are shown as CFS clients and CFS servers; (3) The communication subsystem that provides remote services using customized RPC's built at the present on UDP/sockets. Figure 1 does not show the communication subsystem.

The Calypso data client implements *vnode* and *VFS* operations as defined in AIX. Every vnode operation first checks to see if the node is holding tokens on necessary files, and if so, marks the tokens 'busy' and proceeds to perform the function as required. The tokens are marked 'not busy' before exiting the vnode operation. If any needed tokens are unavailable on the node in the required mode, the token client part of Calypso uses an arbitration protocol to acquire the tokens as described later in this section.

[1] Because of the present UDP limitation, the maximum amount of data that can be sent in one message is 8K-bytes.

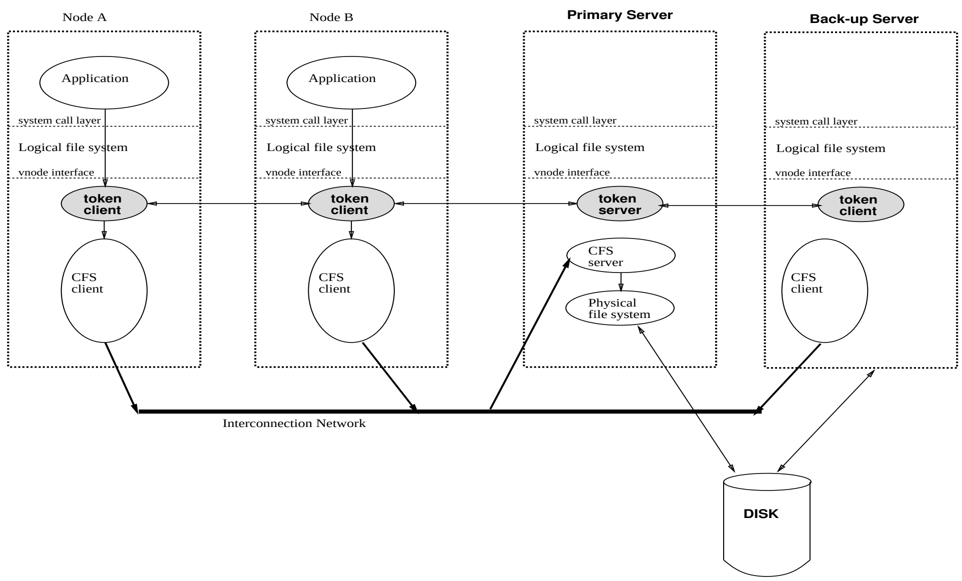

Figure 1: Organization of the Calypso Cluster File System.

In a Calypso client, file pages are cached using the AIX virtual memory. File attributes, name to vnode mappings, and so on are cached in the kernel memory. As needed, these caches are filled and flushed using remote services of the communication subsystem. Most vnode operations use cached data, however, some vnode operations, especially those involving directory changes such as file `rename`, are implemented by executing all of the function on the server. The latter method is referred to as function-shipping of an operation (as opposed to data-shipping), and these operations are usually non-idempotent requiring careful handling during server recovery. Section 6 has more details on this aspect.

In Calypso, seven token types represent various parts of a file and authorization to perform certain operations. The token types are: Data (either whole file or page ranges), Basic Attributes (ownership, access rights, and so on), Size (including the number of disk blocks information), Modification Time, Exist Token (indicating existence of a file to help in retaining a file that is unlinked on one node but may have active references on the others), Open (to support POSIX-style open flags), and Quota. Each token type also has a mode and compatibility semantics associated with it. For example, a Data token can be in the read or write mode, and single reader, multiple writer semantics apply to it. Some token types, such as the Open type, have complex modes and compatibility semantics. The rationale for the token types and details of the token management are described in another paper [17].

When a Calypso client needs to acquire a token in certain mode, its token client contacts the token server which is also the data server. The token server grants the token if there are no conflicting tokens outstanding and records the fact in its internal table. This table is the state of the token server. If conflicting tokens are outstanding, the token server returns a 'copyset' (a list of present token holders) to the requester, and the requester, in turn, sends token revocation messages to the copyset clients. The copyset clients give up their tokens by invalidating their local cache as soon as the tokens are locally marked 'not busy'. In the present implementation of the protocol, the requester then informs the token server about the revocations and thus acquires the token in the needed mode. A write mode token is downgraded instead of being revoked if the requester needs it in the read mode. The token client makes an entry in its local table for the tokens acquired and this table represents the client state.

On Calypso clients, the communication subsystem implements a simple, remote service interface which can be an RPC, or a combination of an asynchronous message and reply notification. Most uses of the remote service are in the RPC mode, but some operations such as the byte-range lock operations use the asynchronous mode. This service is presently built on sockets using unreliable UDP. Therefore, the implementation of a remote service request mainly consists of sending a UDP message and waiting for a response, and if the response is not received in a certain time, the message is resent. The retransmission is repeated with exponentially increasing (up to a limit) waiting periods until the node status service informs of a failure or an interruption from the user occurs.

The server part of the communication subsystem consists of AIX kernel processes that receive a message for a service and invoke an appropriate, pre-installed service routine. The Calypso data and token servers install appropriate service routine entry points as a part of initialization. The communication subsystem maintains a 'call cache' to detect retransmissions of a request for non-idempotent operations and when such a retransmission is detected it replies with previous results without invoking the service routine once more. Handling in-flight, non-idempotent operations is an important aspect of the server recovery.

Service functions of the Calypso data server use the vnode interface to the AIX Journaling File System to access file data. The only state maintained in the data server and in JFS is the byte-range lock state. The rest of the state is maintained by the token server. In summary, when a Calypso server fails, the state of the tokens and the byte-range lock state need to be reconstructed on a back-up server or on the same server after a reboot to maintain data availability.

Node Status Service: The node status service used with Calypso is based on the protocols described in [9]. The actual scheme is somewhat involved, but in its simple form, it can be summarized as follows: The node status service consists of a daemon process running on each node, and the nodes form a logical ring and exchange heart-beat messages among the neighbors. The time between heart beats is typically small, for example, 1 or 2 seconds. If a node fails to respond to a few consecutive heart-beats from its neighbors, the failure communicated to a 'leader' node. The leader then confirms the failure and then propagates this information to the other nodes. The surviving nodes reconfigure into a new logical ring. If the leader node fails, another one is elected dynamically. Network partitions are handled through majority voting. The leader node initiates Calypso recovery. Note that the recovery protocol is entirely built in Calypso, the node status service's function is only to initiate recovery upon detecting a failure. Re-joins are similarly handled.

4 Recovery Protocol

Once a server failure is detected, the leader node of the node status service invokes the recovery function of Calypso. In the following discussion, we refer to the leader node as the recovery controller. The recovery protocol consists of three phases.

In the first phase, the recovery controller informs all cluster clients about the server failure. Upon receiving the server failure notification, each client node marks relevant file systems as UNAVAILABLE. This action suspends any new attempts to contact the server for those file systems. However, accesses to cached data and attributes are not blocked if relevant tokens for the cached entities are held by the client. The check to see if a process can contact a server or not is made at the point of initiating communication for remote service. If the file system is UNAVAILABLE, the requesting process is queued at the file system VFS structure. If a process is already in the communication subsystem attempting to contact the failed server, it is forced to release communication resources and then queued at the VFS structure. Note that such a process may be requesting a non-idempotent operation in which case the failed server may have received the request and may have processed the request to some degree before failing. Handling of such in-flight operations is discussed in Section 5. Once recovery is complete all suspended processes are restarted at the point of remote service request. However, users have the option of issuing normal Unix signals to such a waiting process to terminate it prematurely.

Also in the first phase, the recovery controller informs the back-up server to get ready to receive state reconstruction messages from the client nodes. First, the back-up server takes over the control of the disk through the secondary port of the disk, and then performs consistency checks on the file systems on the disk. Because of write-ahead journaling of metadata changes, JFS consistency check is quicker than in non-journaling file systems.

In phase 2, the recovery controller informs all client nodes to send state reconstruction messages to the server. Each client marks the state of the affected file systems as RECOVERING, and then packages its token and lock states in as few messages as possible. As these messages are received at the back-up server, the server state is reconstructed. The reason for the separation of phases 1 and 2 is to make sure that as the reconstructions messages arrive at the back up server, the back up server should be fully prepared to receive and process them. This prevents unproductive or dropped messages during the recovery. After all clients have successfully sent in their states, the recovery controller enters phase 3.

In phase 3, the recovery controller informs all the cluster nodes that the affected file systems are now available. As a result, all nodes change the state of the file system to AVAILABLE, and resume any suspended processes at the point of remote service request. Because of the clear separation between phase 2 and 3, a client does not reference a file until the state of the file is completely reconstructed. Section 6 describes recovery time measurements.

5 In-Flight Operations and Modified Data

In-Flight Operations: At the time of a server failure, some non-idempotent operations may be in progress on the server, and recovery must handle these in-flight operations appropriately. For example, consider a file create operation. After server reboot or disk take over, how to determine whether or not the create operation succeeded. Note that the file may have existed even before the client request. HA-NFS handles this by using JFS log for write-ahead logging of non-idempotent operations. Since NFS implementations require detection of duplicate requests only for the most recent few seconds, a limited amount of log is sufficient. However, this approach requires either an unlimited amount of log or a very complex logging scheme for a state-full file system. In Calypso, all non-idempotent operations are directory operations and hence we use a technique tailored for handling directory operations without requiring write-ahead logging.

Prior to issuing a remote directory operation, a client determines if the intended operation will succeed or not by examining the contents of relevant directory or directories while holding write token(s) on the directory or directories (which is required under Calypso token management anyway). If the examination reveals that the intended operation will fail, then an appropriate error code is returned to the caller without invoking a remote operation. However, if it reveals that the operation will succeed then the remote request is issued still holding the token(s) on the directory or directories. Now if the server were to fail during the remote operation, after the server reboot or disk take over, inspection of the corresponding directory or directories is sufficient to determine whether or not the operation was completed by the file system.

Client Modified Data: Another aspect that requires careful handling is the modified data that has been flushed from the client cache but has not been written to the server disk. In Calypso, if server recovery is not needed for a file system, when a client flushes modified data to the server the data is not written to the disk immediately for the performance reasons. The usual 30-second sync operation on the server writes such data to the disk. Of course, this can cause loss of data if the server were to fail before the sync operation is complete. In the current implementation, we require client writes to the server be synchronous to the disk when non-disruptive server recovery is needed for the file system. An improved method based on the technique employed in NFS Version 3 [19] can avoid these synchronous writes. In the improved method, clients write modified data to the server memory but the data is not marked clean on the clients until the sync operation on the server writes the data to the disk. After the sync, server notifies the clients that they may mark the data pages clean.

6 Measurements of Recovery Times

To understand the scalability and the break up of the total recovery time, we conducted a series of measurements on a cluster of RISC System/6000 workstations. Most workstations in the cluster

Table 1: Recovery Times Measurements.

Number of Client Nodes	Recovery Times (Seconds)					
	Phase 1		Phase 2	Phase 3	Total
	Basic	Log-Redo	(State Recon.)	(Data Flush & Restart)	
3	0.07	7.25	0.45	3.66	11.43
4	0.05	8.66	0.58	3.85	13.14
7	0.08	15.12	0.92	4.47	20.59
10	0.06	13.16	1.40	4.75	19.37

are Model 530 class (25 MIPS) units, but some faster and slower models are also involved in the measurements. We kept the mix of clients proportionately equal at different measurement points. The workstations are interconnected using 16 Mbits/second token ring. A pair of RISC System/6000's shared a 9333 external, two-ported disk, and these systems functioned as the primary and back up servers.

In the experiment, we used the Andrew benchmark, used in AFS scalability studies [8], as the workload on the clients. The benchmark consists of five phases in which subdirectories are created, source files are copied, file attributes and contents are examined, and the source is compiled. Thus, this benchmark represents the workload of interactive computing typical in research and software development environments. It was reported [8] that each run of the benchmark generates load equal to five average users. Because of copying and compiling, the workload generates a lot of new file data. The file data working set of a single run is about 4 Mbytes, out of which, about 3 Mbytes is the new file data. Approximately, 112 different files are referenced by this benchmark.

In our measurements, we failed the primary server near the end of the compile (the last) phase of the Andrew benchmark by turning off the power to the primary. As the cluster detected the failure and reconstructed the server state on the back up, we internally measured the times for various phases. The experiment is repeated for an increasing number of clients starting from three. The results are shown in Table 1, and times for the log redo (part of Phase 1) and state reconstruction (Phase 2) parts are shown in Figure 2 also.

Clearly, the recovery time is dominated by JFS log redo for the range of clients we have measured. However, results shown here as well as further experiments showed that the log redo time is dependent

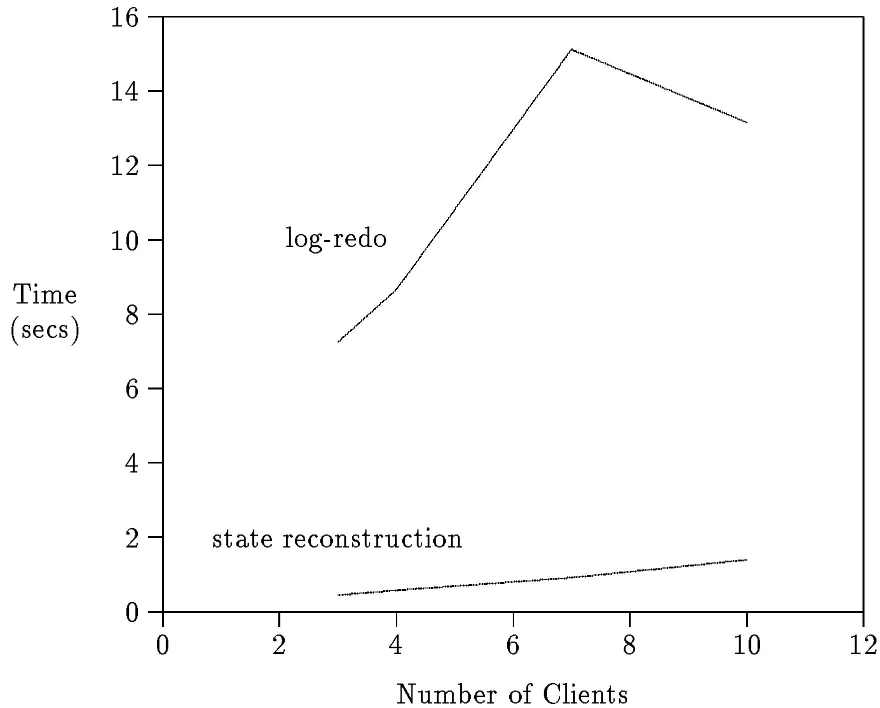

Figure 2: State Reconstruction and Log-Redo Times.

on the size of the log and not on the number of clients after a certain point. JFS maintains a *sync point* in the log indicating the point to which the log needs to be replayed [18]. This sync point is updated (i.e. the log is truncated) when the metadata changes are written to actual on-disk locations. However, to take advantage of journaling, these metadata changes are lazily written to the disk. Therefore, the log contains many redo-able records when the clients are actively creating new files as in the case of the Andrew benchmark. However, since the log size is limited (typically to 8 Mbytes), the number of redo-able records are also bounded. In Figure 2, we see that this bound is reached for about 7 clients. For the number of clients beyond 10, we expect the log redo time to remain around 12 to 16 seconds range.

The state reconstruction time (in Phase 2) is a small part of the recovery time for the range of clients shown here, although, it can be seen to increase with the number of clients. Assuming that the rate of increase remains the same, for a 64-node system the reconstruction time is likely to be about 9 or 10 seconds. This clearly demonstrates that the distributed state can be rebuilt in a reasonable time.

Table 1 also shows significant times for the Phase 3 because in the current implementation, the clients are required to flush modified data to the server before recovery can be completed. Complexity in handling data tokens, which may be in a page-range or whole file mode at the time of recovery, made us to implement a simple but somewhat inefficient solution. We are in the process of eliminating this forced flush operation and we expect Phase 3 times to reduce to the basic Phase 1 times (i.e. to about 0.04 to 0.08 seconds range).

In summary, measurement results showed that the state reconstruction time itself is in the expected range for up to 10 clients, and assuming a similar rate of increase, it should be under 10 seconds for 64 clients. The JFS log redo time has turned out to be a significant portion of the overall time, even though it is bounded by the log size, and is expected to add 12 to 16 seconds to the overall recovery time.

7 Conclusion

In this paper, we described design, implementation, and measurements of a non-disruptive server recovery scheme for stateful file servers. The scheme demonstrates that it is not only feasible but it is also efficient to take advantage of separately aggregated global state (i.e. the token manager state) and natural replication of the state (i.e. the union of client states) for recovery from the failure of a key component. The use of file-system independent state representation using tokens is particularly helpful in reconstructing the server state with only a few messages per client. The three-phase recovery protocol guarantees the integrity of files across server recovery while allowing concurrent transfer of state from clients. As applied to distributed file systems, we show that non-disruptive recovery (i.e., keeping open-files open and not loosing un-synced data) can be accomplished efficiently even in a cache consistent, stateful file system. We also show that multi-ported disks can be exploited for the purpose.

References

[1] B. Welch et al. Sprite position statement: Use distributed state for failure recovery. In *Workstation Operating Systems: Proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II)*, pages 130–133, September 1989.

[2] M. Baker and J. K. Ousterhout. Availability in the Sprite Distributed File System. *ACM Operating Systems Review*, 25(2), April 1991.

[3] A. Bhide, E. N. Elnozahy, and S. P. Morgan. A Highly Available Network File Server. *Winter USENIX Conference*, pages 199–205, January 1991.

[4] A. Bhide and S. Shepler. A Highly Available Lock Manager for HA-NFS. *Summer 92 USENIX*, June 1992.

[5] IBM Corporation. *9076 Scalable POWERparallel 2: Technical Brochure; Order Number: GH23-2485-00*, May 1994. Available from IBM Branch Offices.

[6] IBM Corporation. *AIX HACMP/6000 System Overview Ver. 2.1.0; Order Number: SC23-2595-00*, December 1993. Available from IBM Branch Offices.

[7] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of Summer USENIX Conference*, 1985.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[9] F. Jahanian and Jr. W. L. Moran. Strong, Weak, and Hybrid Group Membership. In *Proceedings of 2nd IEEE Workshop on Management of Replicated Data*, November 1992.

[10] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[11] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the Unix Environment. In *Proceedings of the Summer 1992 Usenix Conference*, June 1992.

[12] J. C. Mogul. A Recovery Protocol for Spritely NFS. In *Proceedings of the Usenix Filesystems Workshop*, Spring 1992.

[13] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[14] R. G. Guy, J. S. Heidemann, W. Mak, Jr. T. W. Pager, G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. *Summer USENIX Conference*, pages 63–70, June 1990.

[15] A. Siegel, K. Birman, and K. Marzullo. Deceit: A Flexible Distributed File System. In *Proceedings of Workshop on the Management of Replicated Data*, June 1992.

[16] J. H. Hartman and J. K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th Symposium on Operating System Principles*, 1993.

[17] A. Mohindra and M. Devarakonda. Distributed Token Management in Calypso File System. In *Proc. of IEEE Symp. on Parallel and Distributed Processing*, Oct 1994 (*To Appear*).

[18] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of Storage Facilities in AIX Version 3 for RISC System/6000 Processors. *IBM Journal Research and Development*, 34(1), January 1990.

[19] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3: Design and Implementation. In *Proceedings of the Summer 1994 Usenix Conference*, June 1992.