

Recovery in the Calypso File System

Murthy Devarakonda, Bill Kish, and Ajay Mohindra

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

Abstract: This paper presents the design and implementation of the recovery scheme in Calypso. Calypso is a cluster-optimized, distributed file system for UNIX clusters. As in Sprite and AFS, Calypso servers are stateful and scale well to a large number of clients. The recovery scheme in Calypso is non-disruptive, meaning that open files remain open, client modified data is saved, and in-flight operations are properly handled across server recovery. The scheme uses distributed state among the clients to reconstruct the server state on a backup node if disks are multi-ported or on the rebooted server node. It guarantees data consistency during recovery and provides congestion control. Measurements show that the state reconstruction can be quite fast: for example, in a 32-node cluster, when an average node contains state for about 420 files, the reconstruction time is about 3.3 seconds. However, the time to update a file system after a failure can be a major factor in the overall recovery time, even when using journaling techniques.

1 Introduction

This paper describes the design and implementation of the recovery scheme in Calypso. Calypso is a distributed UNIX file system implemented on IBM RISC clusters. One potential use of cluster systems is as scalable, fault-tolerant network servers providing mail, bulletin-boards, Web, and Telnet services. Such clusters require an internal file system that is in itself recoverable in addition to providing scalability and standards compliance.

The Calypso recovery scheme uses a protocol between the clients and servers, and it relies on the state distributed among clients. In the event of a server failure, multi-ported disks allow another processor to take over its disks and function as the file server. The recovery is non-disruptive meaning that the file open state and client modified data survive across server recovery, and in-flight client-server operations are transparently completed after recovery. The advantages of the Calypso recovery scheme are that there is no need to explicitly replicate server state, and that the multi-ported disks shorten recovery times and offer cluster reorganization capability.

Among existing commercial file systems, NFS [23] supports a limited variation of non-disruptive recovery through hard mounts, but suffers from lack of scalability due to statelessness [14]. Highly Available Network File Server (HA-NFS) [5, 6] provides non-disruptive NFS server recovery but does not address the scalability problem. AFS [14] offers scalability but does not provide non-disruptive recovery. Several research distributed file systems demonstrated fault-tolerance using redundancy provided by replication and RAID techniques [12, 13, 19, 24]. Recovery using distributed state avoids the expense of maintaining replicas.

Two other distributed file systems, Sprite [3, 4, 11] and Spritely NFS [20], have used distributed state for server recovery. Calypso development took place in parallel and independent of these efforts. Calypso differs from Sprite and Spritely NFS in state representation and internal design. Calypso implements failure detection, recovery management, and congestion control quite differently, and in separate subsystems with well-defined interfaces. The Calypso design also

separates cache consistency state from file data, enhancing the modularity and further simplifying the recovery system. Only Calypso makes use of multi-ported disks to handle permanent processor failures or scheduled shutdown.

In the past four years, we have developed and tested the Calypso file system at product quality for a possible use in RISC System/6000 clusters such as the Scalable POWERParallel (SP) [16] and HACMP [15]. SP is a multicomputer, consisting of independent RISC System/6000 units and an optional high speed switch interconnect. Individual processing units are off-the-shelf systems. Each one has local memory, disks, and other adapter cards. Rack-based packaging provides redundant power supply and monitoring capability. Large SP systems (of 512 nodes) are mainly used in scientific computing. In the commercial area, where Calypso is targeted, more modest sized systems (of up to 32 nodes) are common. HACMP is a set of software services that turns a network of standard RISC System/6000s into a highly-available server complex.

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. It offers: 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. Because of the peer relationship and single administrative domain aspects, Calypso can trust its clients to execute a distributed protocol for server recovery.

In Calypso, as in other distributed file systems, there is one server for each file system and several clients. On the server, Calypso uses the AIX Journaled File System (JFS) through the vnode interface. The server has a physical attachment to disks containing file data. Although 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 Calypso maintains cache consistency 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. The server maintains the state of all tokens, and each client maintains information about tokens it is presently holding. Server state recovery mainly consists of reconstructing the state of tokens on a backup node (or at the same server node after reboot). In addition, it requires reconstruction of a small amount of information related to file locks and disk space guarantees. To a user, server reconstruction is completely transparent. Therefore, except for a small response time delay, a user is unaware of a server failure and its recovery. Recovery from disk failures is beyond the scope of Calypso. RAID devices or disk mirroring (replication) techniques are expected to provide disk fault tolerance.

A node status service [18], 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 consensus algorithm. When this service detects failure of a server, it instantiates the Calypso recovery controller on a leader node, which manages necessary recovery. In the first phase, the controller informs all clients about the server failure and prompts a pre-determined backup server to take over the disk(s) through a secondary port, assure consistency of the file system(s), and perform file system mounts. In the second phase, the controller prompts all clients to send in their state information. It notifies recovery completion in the third phase so that the clients may now access the server as needed. Calypso handles client failures in the same way with several simplifications.

To understand bottlenecks and significant performance factors, we conducted measurements using two workloads. One key result is that the server state reconstruction time is relatively small overall, even though it increases linearly with the number of clients and tokens per client. Batching and congestion control help to keep the reconstruction cost low. We developed a statistical model for estimating the reconstruction time based on the number of clients and tokens. The second result is that if the workload modifies a large amount of file meta-data

shortly before a failure, the log redo time of the AIX Journaled File System becomes the predominant factor of the recovery time. Fortunately, the log redo time is bounded by the log size. In general, however, the activity represented by this time is an important factor in the overall recovery time of a distributed file system.

The rest of the paper is organized as follows. The next section discusses related work. Section 3 presents the background on Calypso relevant to recovery management. Section 4 describes the recovery protocol. Section 5 discusses recovery management issues such as congestion control and handling of in-flight operations. Section 6 presents measurements of the recovery system. Finally, Section 7 presents a summary and conclusions.

2 Related Work

Traditionally, distributed file systems relied on *redundancy* for high availability. File systems replicate server state and file data to deal with processor, disk, and network failures. Server replication in Coda [19], Ficus [12], and Deceit [24] are a few examples of this approach. Redundancy allows these systems to operate *continuously* despite partial failures, at the cost of maintaining replicas. Zebra-style data striping with parity [13] reduces the cost of data redundancy (while providing high bandwidth data access), but would still require replication or reconstruction for the dynamic server state. Server state *reconstruction* using client states can be an inexpensive alternative, since there is little overhead during normal operation. Sprite and Sritely NFS use this technique exclusively.

Calypso uses state reconstruction as well as redundancy. Calypso relies on device level redundancy for data fault-tolerance, and state reconstruction for recovery of dynamic server state. With the present disk attachment technology (e.g., SCSI-2), physically connecting a disk to multiple nodes in a cluster is easy and costs little. At the file system level, the simplicity and inexpensive nature of the state reconstruction technique is a good match for Calypso goals.

HA-NFS, a commercial product, uses this hybrid approach to support NFS. HA-NFS uses multiparted disks, but only one processor actively uses a disk at any time. The backup server takes over the disk if the primary server fails. The backup also takes over the IP address of the primary so that the NFS client code need not be changed to use an HA-NFS server. HA-NFS supports only the NFS protocol and since NFS is stateless, HA-NFS needs to recover relatively small amount of state information consisting of file locks and the most recent *dupcache* entries. The NFS server uses the dupcache to detect duplicate (non-idempotent) UDP messages for the past few seconds. HA-NFS stores this limited state information in JFS's write-ahead log. After the disk takeover, the backup server replays the log to reconstruct the NFS server 'state'. Clearly, Calypso (which is, for example, stateful) differs from HA-NFS a great deal and addresses different requirements.

In state reconstruction, Sprite and Sritely NFS address the same design problems as Calypso. Initially, Sprite used a client-driven recovery scheme [11], but later, Baker improved the client-driven scheme and implemented two more recovery techniques, i.e. server-driven and transparent techniques [3]. (In the rest of this section, Sprite recovery refers to Baker's server-driven technique.) At about the same time, Mogul designed and implemented a server-driven recovery scheme in Sritely NFS [20]. Calypso development took place in parallel and independent of these efforts. All three systems deal with failure detection, quorum consensus, data consistency during recovery, batching of state transfer, and congestion control. However, there are some important differences in the way these file systems implement them:

- In Sprite and Spritely NFS, the server detects failures and drives recovery management, but in Calypso, a third party carries out these functions;
- Sprite and Spritely NFS use RPC timeouts for failure detection and stable storage for client lists; Calypso uses a node status service [18] for maintaining cluster membership and a global mount service (discussed later) for maintaining client lists;
- During recovery, a busy Sprite server sends negative acknowledgements to slow down its clients; Spritely NFS uses quotas to limit RPCs from clients; and Calypso clients use exponentially increasing RPC timeouts as the congestion control mechanism;
- In all three file systems, clients pack as much state as possible in each RPC during state transfer, however, details of maximum packet size and amount of state per packet vary;
- All three file systems suspend non-recovery related RPCs at the clients until server recovery is complete and then re-enable them; this ensures data consistency even during recovery;
- Spritely NFS allows clients to rejoin after a short-term network partition and detects any conflict that may arise as a result; Calypso forcibly revokes tokens held by a partitioned client, and refuses service until the client formally rejoins the cluster through the node status service (at which time the client is reinitialized).

In Sprite’s transparent recovery method, the server keeps its dynamic state in the stable storage, preserving it across failures. After reboot, the server retrieves the state from the stable storage avoiding the need to communicate with clients to recover. For Calypso, this is impractical: to support server state reconstruction on a backup node, the primary and backup nodes must have shared access to the stable storage. In an SP system, a disk is the only way to implement such stable storage and a disk is too slow to store server dynamic state.

In summary, Calypso combines the low cost advantages of dynamic state reconstruction with hardware-level redundancy already available in cluster systems. This matches well with cluster requirements and hardware features. Modularized design of the Calypso recovery system (as group services, recovery control, state transfer engine, and so on) enhances software maintenance.

3 Calypso Overview

Figure 1 shows the architecture of the Calypso file system [10]. Calypso contains three distinct subsystems: (1) the token manager consisting of token clients and token servers, which are shown as the shaded areas in Figure 1; (2) the virtual file system implementation consisting of data clients and servers, which are shown as CFS clients and CFS servers; and (3) the communication subsystem that provides remote services using customized RPC built, at the present, on UDP/sockets (Figure 1 does not show the communication subsystem).

3.1 Calypso Data Client

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

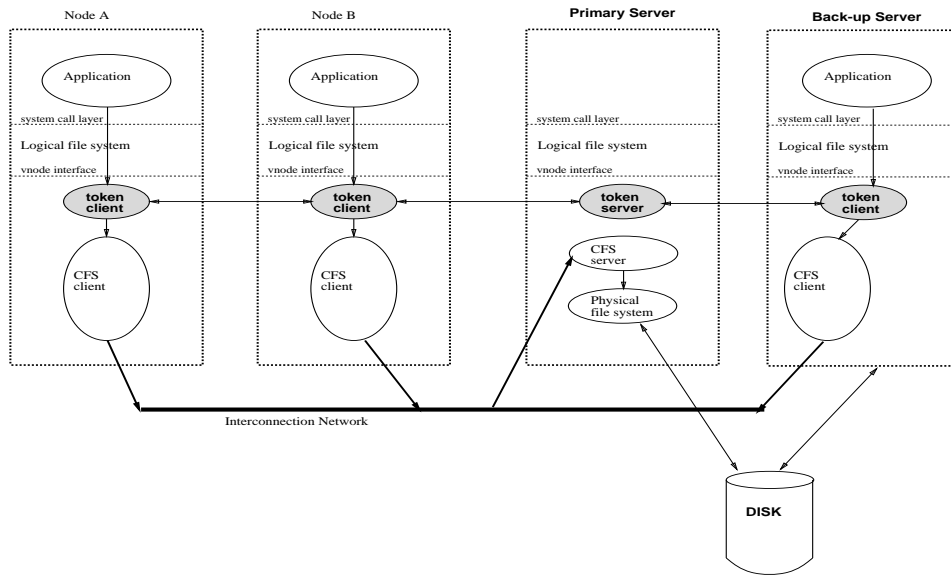


Figure 1: Organization of the Calypso File System. The figure shows how Calypso client, server, and token management components are related to each other. It also shows that Calypso is implemented as a virtual file system on the clients and uses the AIX physical file system (JFS) on the server.

the required mode, the token client part of Calypso uses an arbitration protocol to acquire the tokens.

Calypso client caches data blocks using the AIX *client storage segments*. AIX manages client storage segments and carries out page-ins and page-outs using remote I/O routines provided by Calypso. These routines use RPCs to transfer data. Calypso caches file attributes and name to vnode mappings in the pageable kernel memory, and directly manages them. Since data blocks are page aligned but the other information is in smaller units, this is the way virtual file systems are typically implemented in AIX. Most Calypso vnode operations use cached data; however, some vnode operations, especially those involving directory changes such as file rename, use remote operations directly on the server. These operations are usually non-idempotent requiring careful handling during server recovery.

3.2 Token Management

The rationale for Calypso token definitions is to allow extensive caching while supporting a high level of concurrency and single system UNIX semantics [21]. Calypso uses the following six token types to denote access to various parts of a file and to authorize certain operations:

- Data (either whole file or page ranges);
- Basic Attributes (ownership, access rights, and so on);
- Size (including the number of disk blocks);
- Modification Time;
- Exist Token (indicating existence of a file to retain it until all references are gone);
- Open (to support all POSIX-style open flags).

Tokens have allowable modes and compatibility semantics associated with them. For example, a Data token can be in the read or write mode, and single reader, multiple writer semantics apply. Some token types, such as the Open type, have complex modes and compatibility semantics.

Token Arbitration When a Calypso client needs to acquire a token with a certain mode, its token client contacts the token server which is also the data server. The token server grants the token immediately if there are no conflicting tokens outstanding and records the fact in its internal table. This table is the *server token state*. If conflicting tokens are outstanding, the token server returns a *copyset* (a list of present token holders) to the requester, and sets an *in-transition* flag (and the id of the revoking client) in its token table entry. While the flag is set, all other requests for the token are queued until the flag is cleared. Upon receiving the copyset, the requester 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’. After revoking the tokens, the requester informs the token server about the revocations and clears the in-transition flag. This completes the token acquisition protocol. Figures 2 and 3 show the token arbitration for read-to-write and write-to-read transitions respectively. Note that the dirty data is flushed directly from a client to the server and similarly a client reads data directly from the server. 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 token state*.

As seen above, Calypso uses an unconventional, client-centric approach to token arbitration. This approach has the advantage of limiting the number of messages at the server as well as reducing the server complexity. No matter how many token holders exist, Calypso token revocation requires processing of only two RPCs at the server. More importantly, this design allows a *passive* server design where a server never *initiates* any communication with the clients; it only services client-initiated RPCs. In this design, there is no need to maintain revoker threads or to manage synchronization among different thread types. The only potential disadvantage of this approach is that the recovery system must deal with the possibility of a revoker client failing during token revocation. For simplicity, Calypso reconstructs the server state from scratch even for client failures therefore, failure of a revoking client does not require special handling.

What happens if a client isn’t willing to give up its tokens? First, such a scenario cannot occur in the absence of failures and software errors. The token client state and the logic for marking a token ‘busy’ and ‘not busy’ are in the Calypso kernel code. Applications do not directly access or manage the token state. Second, if a token-holding client fails in a manner detectable by the node status service, the recovery scheme handles the situation as described in Section 5.3. In the case of undetectable failures and software errors, the revoking client initiates unilateral actions to fence off the non-responsive client as a server might take in a server-centric protocol.¹

3.3 Communication Subsystem

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 file locking operations use the asynchronous mode. This service is presently built on sockets using unreliable UDP. It sends a UDP message and waits for a response, and if a response is not received in a

¹This is an intrinsically difficult problem in any distributed system. Calypso tries to limit the impact of such errors to a minimum number of clients and file systems. Calypso also writes error messages to a log and on the console for operator intervention.

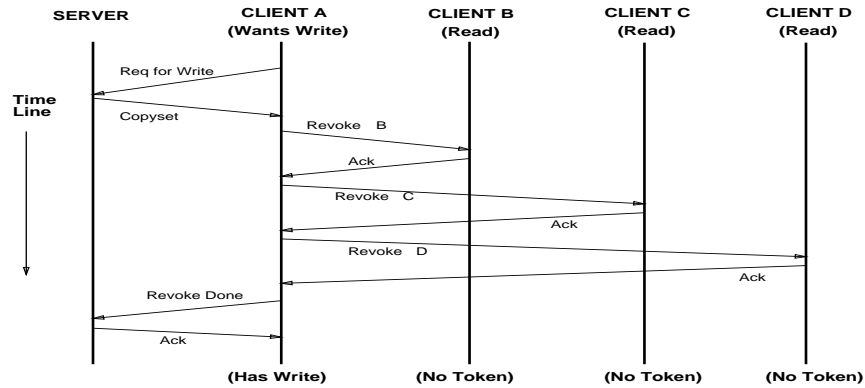


Figure 2: Read to Write Token Arbitration. To obtain a write token while read tokens are outstanding for some file data, a client first gets a list of current holders, then revokes read tokens from them, and finally informs the server about the change.

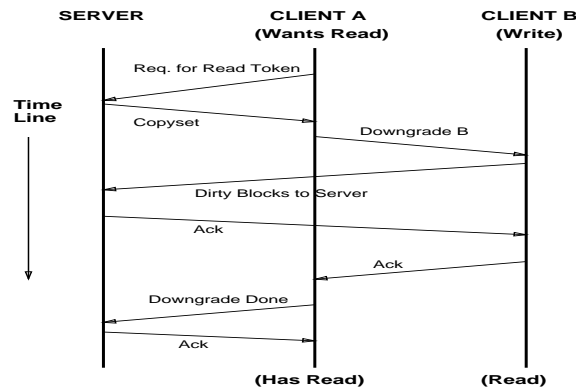


Figure 3: Write to Read Token Arbitration. To obtain a read token, while a write token is outstanding for some file data, a client first finds the current holder, downgrades the token to the read mode (this forces the current holder to write changes to the server), and finally informs the server about the change.

certain time, it re-sends the message. The messages are retried with exponentially increasing waiting periods (up to a limit) until a failure notification (from the node status service) or an interrupt from the user.

3.4 Calypso Server

The server part of the communication subsystem is fairly conventional. It consists of AIX kernel threads that receive a message for a service and invoke a 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 the previous results without invoking the service routine. 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 Journaled File System (JFS) to access file data. The only state maintained in the data server is the file lock state and disk space guarantees. Typically, this is a very small amount of state. The rest of the state information, which is the largest amount, is maintained by the token server. In summary, when a server fails, the recovery system reconstructs the state of the tokens, file locks, and disk space guarantees on a back-up server or on the same server after reboot.

3.5 Group Services

The overall operation of the Calypso recovery system relies on two key services, which are a part of the *group services* for the IBM SP system. The first service maintains the up or down status (by a certain criteria) of the cluster nodes. The second service maintains where and how file systems are mounted in the cluster.

Node Status Service The node status service used with Calypso is based on the protocols described in [18] [17]. Other researchers have also worked on such protocols [2]. In its simple form, the internal operation of the service is as follows: the node status service consists of a daemon process and some interrupt-level code on each node. The nodes form a logical ring and exchange heart-beat messages among the neighbors. The heart-beats are processed at interrupt level. 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 is communicated to a *leader* node. The leader then confirms the failure and propagates this information to the other nodes. The surviving nodes reconfigure into a new logical ring. Re-joins are similarly handled. If the leader node fails, another one is elected dynamically. Network partitions are handled through majority voting. When a membership change is detected, the leader node initiates Calypso recovery through a recovery controller process, which in turn invokes Calypso interfaces to carry out the recovery.

Global Mount Service For each Calypso file system, this service maintains the required and actual status of file system mounts. It keeps the list of clients for a file system as well as the server and backup nodes. The service uses stable storage or replication to deal with its own internal failures. This service can be queried at any time to determine the clients and server for a Calypso file system and to determine a backup server. In addition, this service also performs atomic mount or unmount of a Calypso file system using a two-phase protocol.

4 Recovery Protocol

Once a node failure is detected, the leader node of the node status service instantiates a recovery controller process. The recovery controller, using the global mount service, determines the file systems effected by the failure, and the necessary recovery. If the failed node is a server for a file system, the controller then determines the list of live clients and a backup server for it. The controller's job is to sequence through recovery phases, and for each phase invoke an appropriate interface in the Calypso file system on each surviving node. To do this concurrently, the recovery controller starts each phase by spawning one thread for every live client. Each thread sends an RPC to its client for a recovery subtask, and when the RPC is complete the thread terminates. Termination of all threads signals the end of a phase.

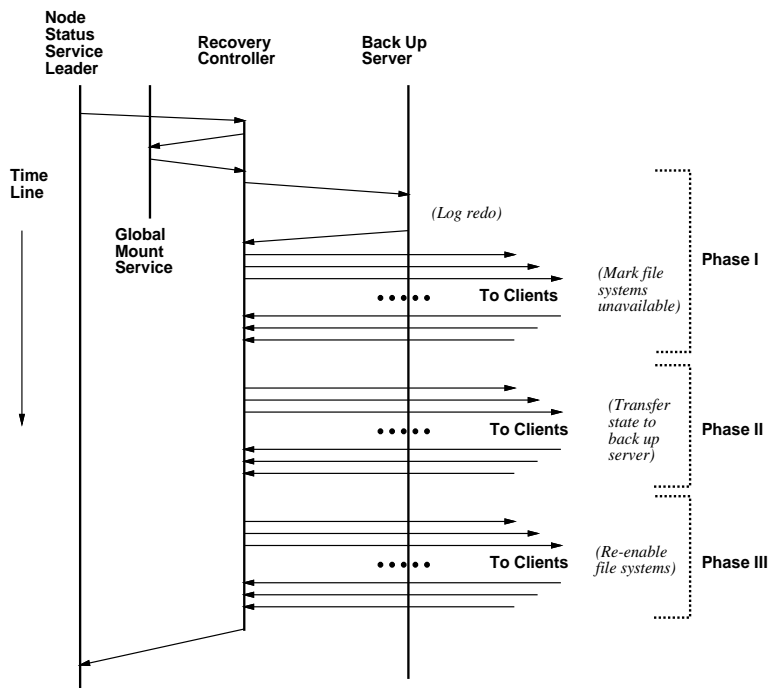


Figure 4: The Calypso Recovery Protocol. When a failure is detected, the node status service instantiates a recovery controller process. The controller obtains the list of nodes involved and sequences through the three phase recovery. For simplicity, the figure does not show messages for state transfer. The node status service, global mount service, and recovery controller are processes that run on a cluster node.

In the first of the three phases, the recovery controller informs all cluster nodes about the failure. Upon receiving a server failure notification, each client node marks relevant file systems UNAVAILABLE. This action suspends any new attempts to contact the server for those file systems. However, a client does not block accesses to cached data and attributes if it holds relevant tokens. A client checks if a process can contact a server or not at the point of initiating communication for remote service. If the file system is UNAVAILABLE, the client queues up requesting processes at the file system VFS structure. If a process is already in the communication subsystem attempting to contact the failed server, Calypso forces it to release communication resources and then queues it up at the VFS structure. Once recovery is complete, Calypso restarts all suspended processes at the point of the remote service request. However, users have the option of issuing normal UNIX signals to such waiting processes to terminate them prematurely.

Also in the first phase, the recovery controller informs the backup server to get ready to receive state reconstruction messages from the client nodes. First, the backup server takes over the disk through the secondary port of the disk, and then it performs consistency checks on the file systems on the disk.² Because of write-ahead journaling of metadata changes, JFS consistency checking is quicker than in BSD-style file systems.

²Note that throughout the paper when we say consistency check for JFS, we mean replaying redo-able log records; JFS does not require BSD file system type consistency check.

In phase 2, the recovery controller informs all client nodes to send state reconstruction messages to the backup/rebooted 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 the backup/rebooted server receives these messages, it reconstructs the dynamic state. The reason for the separation of phases 1 and 2 is to make sure the backup/rebooted server is fully prepared to receive the reconstruction messages. This prevents unproductive or dropped messages during recovery. After all clients have successfully transferred 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 the remote service request. Because of the clear separation between phases 2 and 3, a client does not reference a file until reconstruction is complete. Thus, Calypso ensures cache consistency even during recovery.

5 Recovery Management

Besides the state reconstruction protocol, a recovery system must deal with several other design issues. For example, what happens to RPCs in progress at the time of a server failure? Are there any measures to avoid congestion at the server during the reconstruction? How does the recovery system cope with transient network partitions? This section discusses how Calypso addresses these issues.

5.1 In-Flight Operations

If in-flight RPCs are idempotent, clients simply retry them after server recovery. Non-idempotent operations require special care. For instance, if a client sent a file create RPC and has not received a reply, how can the backup/rebooted server determine its progress prior to the server failure? Note that the file may have existed even before the client request. Therefore, merely examining the directory contents after the disk take over (or reboot) would not suffice. Normally, to deal with this problem, one would use write-ahead logging or depend on the internal state maintained by the JFS. Write-ahead logging is complex. For example, in a stateful server such as Calypso it is not obvious when and how to truncate the write-ahead log. We did not want to make Calypso design depend on the way JFS maintains its internal state since that would mean loss of portability and would require changes in Calypso whenever the JFS changes. Since all non-idempotent RPCs in Calypso are directory operations, we decided to use a check-before-request technique.

In this technique, prior to issuing a remote directory operation, the client determines if the intended operation would succeed or not by reading the contents of relevant directory or directories while holding write token(s) on the directory or directories (which is required under Calypso token management in any case). If the examination reveals that the intended operation would fail, then an appropriate error code is returned to the caller without invoking a remote operation. However, if the operation would succeed (in the absence of failures) 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 has been completed by the file system. This scheme might create a bottleneck for directories like /tmp as users from different nodes frequently create and delete temporary files. But, in SP

systems, such temporary directories are always on a local disk (and not in Calypso). Each node on an SP system has at least one 1G byte disk and most AIX files, temporary files, and swap areas are on the local disk.

5.2 Congestion Control

Baker has quantified congestion problems that can arise during server state reconstruction [3]. Even in normal operation, a server can easily experience peak periods of bursty activity. State reconstruction is particularly stressful as the protocol allows clients to transfer their state in parallel to minimize overall reconstruction time. Calypso uses an exponential back-off scheme to reduce congestion. It is used during recovery as well as in normal operation. A client initially waits for 1 second for a reply to an RPC and then retries. If it still does not receive a reply, the client waits for 2 seconds before retrying again. The delay increases to 4 seconds, 8 seconds, and so on until a limit is reached (e.g. 90 seconds), which can be set by an administrator. Of course, the failure detection in the node status service and subsequent recovery actions break this retry sequence and put the RPC attempt in a suspended state until recovery is completed (or until it is determined that the recovery is not possible). The idea behind this scheme is that in the absence of any recovery actions from the node status service, the only reason for a lack of response is that either the server is too busy, or the message packet is lost. Repetitive losses indicate the former and hence the RPC backs off to reduce congestion. Duplicate RPC requests are filtered by the next higher level in Calypso, which handles idempotent and non-idempotent RPCs. Calypso also has other means of avoiding communication and server congestion — for example, each client uses only a fixed number of communication ports (sockets). Therefore, a client can not generate an unbounded number of RPC messages simultaneously for another node. An administrator can change the number of ports to suit a specific configuration.

5.3 Client Failures

Calypso handles client failures as a simple case of server failures. The recovery controller reconstructs the server state from scratch by going through the three phases of the protocol. One major simplification is that it skips the JFS log redo step. One might consider an even simpler approach of just going through the server token table and releasing tokens held by the failed client. In Calypso, such a simple technique is not adequate because of the client-centric token arbitration. If the failed node is in the middle of revoking tokens it is not possible to determine how far it has progressed before the failure. The three phase protocol, however, can deal with such a scenario. In addition, since the three phase protocol informs the surviving clients about a client failure, if any one of them is trying revoke a token held by the failed node, it would stop the revocation attempt and continue. As measurements indicate, server state reconstruction alone takes relatively small amount of time.

5.4 Client Modified Data

Modified data on the client is lost when a client fails. It is not our goal to address such a loss as it is somewhat similar to a single system failure scenario. However, we do provide a way to handle a subtle implication of a server failure related to client modified data. In the normal operation, when a client flushes its dirty data to the server it is written to the JFS file cache on the server. Eventually, the periodic 30-second sync operation writes the data to the disk. This scheme can cause data loss if the server fails before the sync operation is complete. In

the current implementation, Calypso offers a mount time option for a file system to avoid this secondary data loss possibility. With this option, when a client flushes its data to the server, it is written to the server disk synchronously. The measurement section quantifies the overhead of this option. An improved method based on the NFS Version 3 [22] approach can avoid these synchronous writes. In NFS-3, clients write modified data to the server file cache 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 operation, the server notifies the clients that they may mark the data pages clean. Calypso (as well as NFS version 3) require new interfaces in JFS to implement this feature.

5.5 Network Partitions

Because of a network partition, a client might get disconnected from the rest of the cluster for a long, but finite, amount of time. Calypso must deal with two problems here. First, what to do with the tokens held by the client? Second, what to do when the client rejoins the cluster? When a client gets disconnected in this manner, the node status service declares that the client is 'dead' and forces Calypso recovery for this failure as described in Section 5.3. As a result, tokens held by the client are released and any modified data on the client is lost. When the client reconnects, none of the cluster nodes honor messages from the client until it goes through the cluster rejoin process (i.e. it is fenced off). During the rejoin process, the client reinitializes the Calypso state.

What if a pair of nodes becomes disconnected from the cluster, and can talk to each other but not with the server? One possibility is that the node status service recognizes the minority partition and initiates recovery on the rest of the cluster as described in Section 4. The other possibility is that the node status service does not detect a network partition because of its own internal errors or because the failure is limited to Calypso. If the node status service does not recognize a network partition, when one of the disconnected nodes cannot communicate with server even after a very large number of retries, it unmounts relevant file systems and returns an error code to applications. In a well tested system, this scenario is unlikely. Normally, a Calypso node treats inability to communicate as a sign of congestion, but if communication fails even after a very large number of retries, it takes the drastic measure of unmounting relevant file systems. These design decisions are made with simplicity in mind.

5.6 Multiple Failures and Failures During Recovery

What if multiple failures occur? For example, what happens if a client and a server fail at the same time? In the Calypso recovery scheme, the node status service serializes all failure events (perhaps, in an arbitrary order). Also, the node status service handles them immediately and asynchronously with respect to any ongoing activity. If the second failure occurs after recovering from the first, then it is handled as usual. But, if the second failure occurs during the recovery of the first, it requires special care.

The Calypso design for handling failures during recovery consists of simply starting over from the beginning. The node status service informs the recovery controller about the failure. (If the recovery controller dies, the node status service starts another.) The recovery controller first switches to a clean up phase and then to the first recovery phase. Since clients do not destroy their local state during recovery, one can always restart state reconstruction. However, any ongoing communication and internal progress indicators have to be cleaned up first.

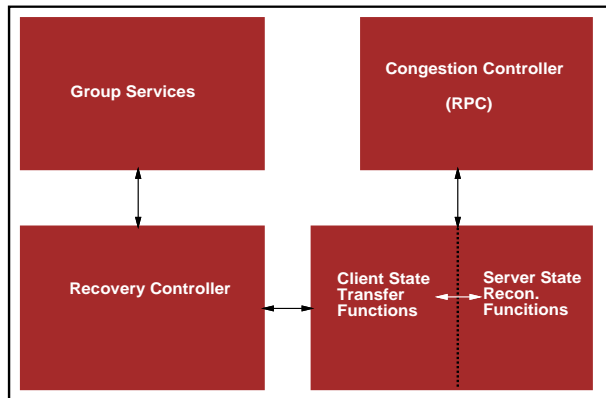


Figure 5: Functional Modules of the Calypso Recovery System. As shown in the figure, the group services, recovery controller, state transfer functions, and congestion controller are in distinctly different modules with well defined interfaces and interactions.

5.7 File Locks and Disk Space Reservations

Although tokens represent most of Calypso client state, a small amount of state is maintained in the virtual file system for file locks and disk space reservations. File locks are used infrequently in UNIX environments, so Calypso implements them by function shipping the operations to the server. However, each client maintains a list of file lock requests that are outstanding or have been granted. If a server fails, Calypso recovery uses this list to reconstruct the state of file locks on the server.

Calypso guarantees disk space for newly created pages without contacting the server on every write or at every file close. Calypso uses a reservation scheme, where a client obtains a reserved amount of (logical) space and locally allocates estimated amounts out of this reservation [9]. Both clients and the server maintain appropriate counters to manage these reservations. Calypso recovery uses the contents of the client counters to reconstruct the server counters. Each client maintains three counters, so the transferred state is quite small.

5.8 Modularity of the Recovery System

From the preceding description of the Calypso recovery system, we see that it consists four separate subsystems as shown in Figure 5. The group services include the node status service and the global mount service; the recovery controller sequences through the recovery phases; the congestion controller is a part of the RPC subsystem; and the client state transfer functions and the server state reconstruction functions form the fourth subsystem. The separation is so distinct that each subsystem has been completely replaced several times during the development. This design not only separates the distinct parts of a recovery system but also enhances software maintenance.

6 Measurements of Recovery Times

To understand the performance characteristics of the recovery system, we conducted a series of measurements on a 32-node IBM SP1 system. The key results from the measurements are as follows.

- State reconstruction time, even though it increases linearly with the number of clients and files, is relatively small due to batching and congestion control;
- The amount of meta-data modified shortly before a failure has a large impact on overall recovery time, because JFS log-redo time, although fast relative to BSD-style file system checks, increases with the number of log records that require processing.

6.1 Hardware

The SP1 system used in the measurements consists of 32 RISC System/6000 processors of Model 370 class. SPECint92 rating is 70.3 for these processors. Each processing unit has 128 Mbytes memory and 1 Gbytes disk on a SCSI-2 bus. The processors are on a non-blocking proprietary switch that has a nominal bandwidth of 20 Mbits/second. Out of 32 nodes, we used one node as the Calypso server, another one as the recovery controller and group services engine, and the rest as Calypso clients. This SP1 system does not have a multi-ported disk, therefore we reconstructed server state on the same server after rebooting it. In a preliminary study, we reported results from a configuration consisting of eleven RISC System/6000 workstations (approximate SPECint92 rating of 28.0) and a dual-ported disk attached to two of these workstations [8].

6.2 Workloads

We made measurements using two workloads, the Andrew Benchmark [14] and a synthetic mix of file references using Baker's approach [3]. The reader should see relevance of these workloads in their limited role in our experiments. Our goal is not necessarily to *benchmark* the recovery time, but to understand the performance characteristics and bottlenecks of the recovery system. Although, a more representative workload is desirable, it is not necessary for our measurements. We need workloads that create a *predictable amount* of state in a *predictable manner*. The emphasis here is on predictability. The Andrew and synthetic mix workloads offered these properties plus a reasonable variation from one another.

Andrew Benchmark It consists of five phases: subdirectory creation, source files copying, file attribute search, file content search, and source compilation. The pattern is typical of interactive technical computing in research and software development. Previous reports [14] indicate that each run generates a load equal to five average users, which is generally unimportant for our experiments, except that it reflects the amount of meta-data modification in the last (compilation) phase just before the simulated server failure. File data working set of a single run is about 4 Mbytes, about 3 Mbytes of it is the new file data. This workload references approximately 113 different files. About half of them are created in its last phase. Calypso clients maintain six different tokens for an opened file, and since the workload effectively opens and closes all of its files, the total state is about 678 tokens.

Each client runs one instance of the Andrew Benchmark in a separate directory. The server is rebooted after the benchmark completes on all clients. The node status service detects the

Table 1: Recovery Times for the Andrew Benchmark. This table shows elapsed times for the three recovery phases from the experiments using the Andrew Benchmark. The log redo and the rest of Phase 1 times are shown separately. The key result here is that the log redo time is a predominant part of the recovery followed by the state reconstruction time.

Number of Client Nodes	Recovery Times (Seconds)				
	Phase 1		Phase 2	Phase 3	Total
	Log-Redo	Rest of Ph#1	State Recon.	Restart Time	
1	2.59	0.031	0.198	0.039	2.86
8	14.55	0.030	0.531	0.044	15.16
16	17.00	0.031	0.852	0.048	17.93
24	17.77	0.031	1.113	0.049	18.96
30	18.89	0.032	1.504	0.047	20.47

failure and after server reboot, instantiates recovery controller to reconstruct the server state. Recovery timing starts *after* failure detection because relevant instrumentation is in the recovery controller and Calypso. The experiment is repeated for a varying number of clients in the range of 1 through 30.

Synthetic Workload Following Baker’s methodology [3], we created a synthetic workload that references a set of shared and private files in the following manner: 200 private files in unopened mode, 60 private files in read mode, 30 private files in write mode, 100 shared files in unopened mode, 30 shared files each in read and write modes. Each client runs this benchmark accessing the private files from its own, separate directory. Shared files are common to all clients. In addition, as a part of the workload, all clients write a few bytes to a log file, and one client creates a new file by copying contents from one privately written file from each of the clients (simulating the `lib` file creation in a parallel make). With this workload, each client accrues state information for 420-452 files. Since most files are unopened the actual state is about 1880 tokens for an average client. Unlike the Andrew Benchmark, this workload modifies small amount of meta-data (relative to its working set) and spreads the update activity throughout its running time. This behavior is an interesting variation from the Andrew Benchmark. The rest of the experiment and measurement procedures are similar to Andrew.

6.3 Results and Discussion

The time to reboot a RISC System/6000 varies considerably based on the number of adapter cards and so on. However, measurements on one of the experimental SP nodes shows that the reboot time is about 189 seconds. If a server node crashes, this is the minimum time before Calypso can start its recovery using the same server node. In comparison, measurements on the workstations configuration with dual-ported (external) disk show that it takes 5 seconds for the backup server to take over the disk and become ready for Calypso recovery. Clearly, dual-ported external disks reduce overall recovery time.

Table 1 shows elapsed times for each of the three recovery phases from the experiment using the Andrew Benchmark. The key result from this table is that the JFS log redo and state reconstruction times are the main components of the total recovery time. In fact, for the Andrew Benchmark most of the recovery time is in JFS log redo. The initial failure notification

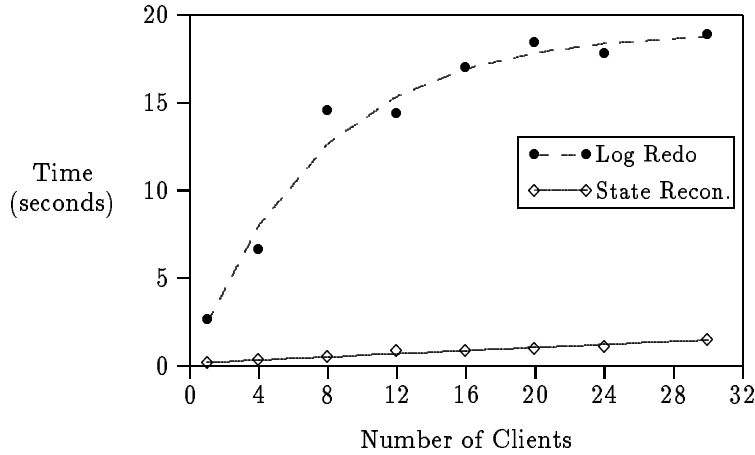


Figure 6: Log-redo and state reconstruction times for the Andrew Benchmark. Markers correspond to measurements and connected lines correspond to the analytical models described in the text.

and the final restart notification take little time. The rest of this section further analyzes the main components of the recovery time.

JFS log redo AIX Journaled File System (JFS) writes meta-data changes to a write-ahead log to optimize disk utilization as well as to reduce the time to check consistency of the file system [7]. For example, when JFS creates or removes a file, or when makes certain inode changes, it creates log records reflecting these changes. When the UNIX sync operation executes, JFS writes actual data and meta-data to permanent locations on the disk and then truncates the write-ahead log accordingly. After a crash (or after taking control of the disk in a multi-ported configuration), JFS replays relevant log records and applies appropriate changes to the on-disk file system before it allows a mount of the file system. JFS maintains a *sync point* in the log indicating the end of the replayable log records. This sync point is updated during the periodic or forced sync operation. In the normal operation, the number of replayable log records is proportional to meta-data changes since the last sync operation. However, the write-ahead log has a finite size, the default size being 4 Mbytes. Therefore, when the log is nearly filled up, JFS forces an effective sync operation on itself,³ thus truncating the log by force. Consequently, even though the log redo time increases with the amount of meta-data modified since previous sync operation, it is bounded by the log size.

The asymptotic nature of the log redo times can be seen in Figure 6, which presents the log redo and state reconstruction times for the Andrew Benchmark. The size of the log may have performance implications in the normal operation, but the decision to increase or decrease the log size is completely orthogonal to Calypso recovery. Indeed, we always used the default log size in our experiments. If the log were longer, the log redo time may be larger for high-end measurements with a corresponding reduction in Andrew elapsed time. Measured log redo times, for the Andrew, fit an exponential model, $L = 19086 - e^{-0.14C}$, where L is the log redo time in milliseconds, and C is the number of clients. For all models presented here, we used the linear and nonlinear regression procedures of the SAS statistical package [1]. We always obtained a statistical significance level (the probability of rejecting the model even if it is true) of 0.0001.

³JFS performs this as a low priority operation so as to avoid a large spike in user response time.

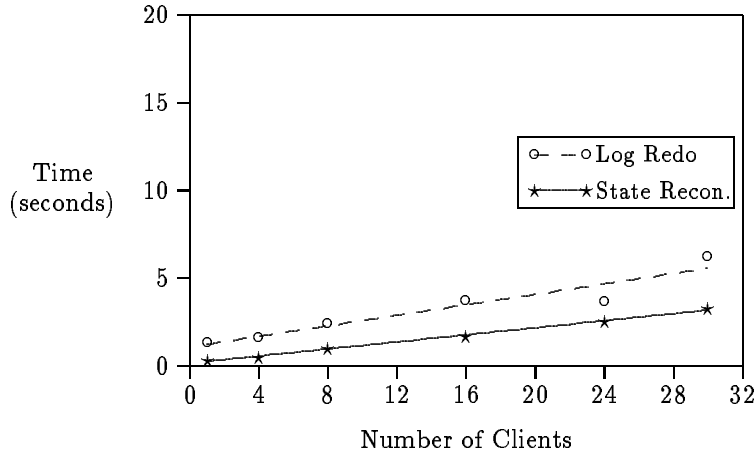


Figure 7: Log-redo and state reconstruction times for the Synthetic Benchmark. For direct comparison, this figure uses the same scale as Figure 6. As in Figure 6, markers correspond to measurements and connected lines are derived from analytical models.

As seen in Figure 7, log redo times for the synthetic workload show quite a different behavior. They are significantly smaller (although not smaller than the reconstruction time) and increase linearly with the number of clients. Note that the scale on Figures 6 and 7 is the same to facilitate a direct comparison. Even though the synthetic workload creates more new files than the Andrew, it does not have a bursty creation mode just before the failure. Therefore, the log contains fewer redo-able records after the reboot. The log redo times, for the synthetic mix, follows a linear model, $L = 1049.3 + 149.7C$, where L is the log redo time in milliseconds, and C is the number of clients. The two sets of measurements demonstrate how application behaviors can effect recovery times.

Note that the activity represented by the log redo time is not specific to JFS only. It might take some other form in other file systems, but it will always be there. For the BSD-style file systems, such a component (in the form of consistency check) is likely to be even larger and proportional to the file system size. Baker’s measurements indicate shorter times for the log structured file system [3].

Figures 6 and 7 also show the state reconstruction times for the two workloads. The state reconstruction time, although, relatively less prominent compared to the log redo time, is a direct result of using distributed state to recover from failures. Therefore, as one might suspect, it directly depends on the number of clients as well as on the amount of state among the clients at the time of recovery. For better clarity, Figure 8 shows only the reconstruction times using an appropriate scale. Because the reconstruction time, unlike the log redo time, is independent of *how* the state is created, a single linear model can represent the measured times for both workloads:

$$R = 146.4 + (12.6 + 0.047 T) C, \quad (1)$$

where R is the reconstruction time in milliseconds, C is the number of clients, and T is the number of tokens per client at the time of recovery. Note that there is a practical limit on the amount of state a Calypso client is willing to keep. It keeps necessary state for all open files, but exercises a limit on the number of unopened files. The default settings correspond to about 512 such files.

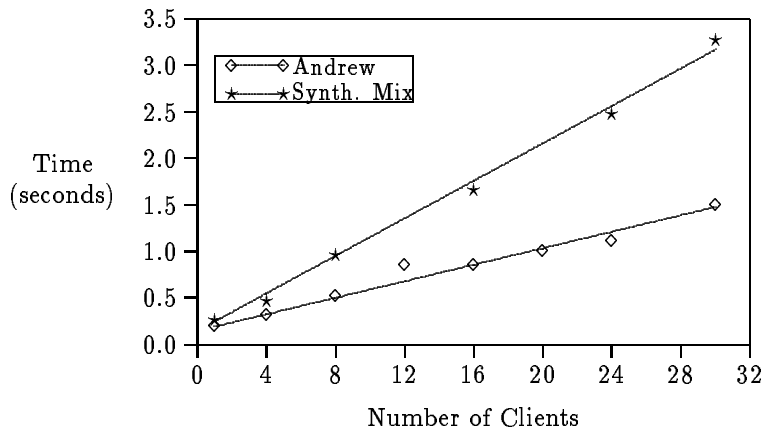


Figure 8: State reconstruction times only. This figure shows server reconstruction times for the Andrew and the synthetic mix together. Markers indicate measurements and connected lines correspond to the linear model discussed in the text.

For 30 clients, the measurements show reconstruction times of 1.5 and 3.3 seconds for the two workloads (678 and 1880 tokens per clients respectively) indicating fairly good scalability. Transferring about 200 tokens in each RPC helps to amortize the communication cost. Because of the congestion control, there are no recovery storms or exponential behavior for the measured range of clients. Therefore, the key result from these measurements is that batching and congestion control help to keep the reconstruction cost low, even though, in principle, it does increase linearly with the number of clients and tokens.

6.4 Overhead of Write-Through on Client Cache Flush

As stated in Section 5.4, in normal operation, modified data is written to the server cache when it is flushed out of a client cache. However, a failure at the server can then cause data loss. To avoid such a data loss, Calypso offers an option on the mount command that forces cache flushes directly to the server disk. Writes are still asynchronous with respect to user applications unless applications use `O_SYNC` flag in opening a file or use the `fsync` system call. With this option, once data is forced out of a client cache it is safely on the disk at the server. However, this option increases disk utilization at the server. As a result, the Andrew Benchmark elapsed time increases by as much as 24% for 16 clients (see Table 2). We plan to minimize this performance

Table 2: Performance penalty due to synchronous writes to the server disk. To avoid data loss, Calypso offers an option to write data directly to the server disk when flushing modified data out of client caches. This table shows its impact on Andrew benchmark elapsed times. Increased disk utilization reduces overall performance for a large number of clients.

Number of Clients	Andrew Benchmark Elapsed Time (Seconds)		
	Without Recov.	With Recov.	Perf. Penalty (%)
1	45.5	47.0	3%
16	114.6	142.1	24%

concern using the technique employed in NFS Version 3 [22]. It requires support in AIX virtual memory management and in JFS, which is likely to become available once NFS Version 3 is ported to AIX.

7 Conclusion

In this paper, we described the design and implementation of a non-disruptive recovery scheme for the Calypso file system. Non-disruptive recovery means that open files remain open, client modified data is saved, and in-flight remote operations are handled properly across server recovery. We showed that such a non-disruptive recovery can be accomplished efficiently even in a cache consistent, stateful file system.

The recovery scheme further demonstrates that using distributed state among clients is a viable alternative to explicit replication. Maintaining client (cache consistency) state separately in the form of tokens made it particularly easy to implement this scheme. A three-phase protocol guaranteed data consistency even during server recovery while allowing concurrent transfer of state from clients. Batching of state transfer and congestion control are instrumental in keeping the cost of state reconstruction low.

While Calypso uses reconstruction for dynamic state recovery, it also exploits inexpensive hardware redundancy (in the form of multi-ported disks) for data availability. For cluster systems, the combination offers a low overhead solution with short recovery times.

Acknowledgements

Ashutosh Tripathi made the measurements reported in Section 6; He found and fixed many subtle errors in the implementation of the Calypso recovery system. John Robinson provided valuable comments on the presentation. Andy Zlotek, Anand Rao Ladi, Bob Curran, and Bill Tetzlaff have contributed a great deal to the Calypso project. Anonymous referees helped to improve the paper significantly. We thank them all for their invaluable contributions.

References

- [1] Anonymous. *SAS User's Guide: Statistics, Version 5 Edition*. SAS Institute, Inc., 1985.
- [2] O. Babaoglu, R. Davoli, L. Giachini, and M. Baker. RELACS: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed System. In *Proc. of the Twenty-Eighth Hawaii International Conference on System Sciences*, Jan 1995.
- [3] M. Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California, Berkeley, CA 94720, January 1994. Technical Report UCB/CSD 94/787.
- [4] M. Baker and J. K. Ousterhout. Availability in the Sprite Distributed File System. *ACM Operating Systems Review*, 25(2), April 1991.
- [5] A. Bhide, E. N. Elnozahy, and S. P. Morgan. A Highly Available Network File Server. *Winter USENIX Conference*, pages 199–205, January 1991.

- [6] A. Bhide and S. Shepler. A Highly Available Lock Manager for HA-NFS. *Summer 92 USENIX*, June 1992.
- [7] 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.
- [8] M. Devarakonda, B. Kish, and A. Mohindra. Server Recovery Using Naturally Replicated State: A Case Study. In *Proc. of IEEE Conf. on Distributed Computing Systems*, May 1995.
- [9] M. Devarakonda, A. R. Ladi, A. Zlotek, and A. Mohindra. Disk Space Guarantees as a Distributed Resource Management Problem: A Case Study. In *Proc. of IEEE Symp. on Parallel and Distributed Processing*, Oct 1995.
- [10] M. Devarakonda, A. Mohindra, J. Simoneaux, and W. H. Tetzlaff. Evaluation of Design Alternatives for a Cluster File System. In *Proc. of the USENIX Technical Conference*, Jan 1995.
- [11] 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.
- [12] 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.
- [13] J. H. Hartman and J. K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th Symposium on Operating System Principles*, 1993.
- [14] 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.
- [15] IBM Corporation. *AIX HACMP/6000 System Overview Ver. 2.1.0; Order Number: SC23-2595-00*, December 1993. Available from IBM Branch Offices.
- [16] IBM Corporation. *9076 Scalable POWERparallel 2: Technical Brochure; Order Number: GH23-2485-00*, May 1994. Information available from <http://ibm.tc.cornell.edu> Web page.
- [17] F. Jahanian, R. Rajkumar, and S. Fakhouri. Processor group membership protocols: Specification, design and implementation. In *Symposium on Reliable Distributed Systems*, October 1993.
- [18] 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.
- [19] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.
- [20] J. C. Mogul. Recovery in Spritely NFS. *Computing Systems, the Journal of the Usenix Association*, 7(2), Spring 1994.
- [21] A. Mohindra and M. Devarakonda. Distributed Token Management in Calypso File System. In *Proc. of IEEE Symp. on Parallel and Distributed Processing*, Oct 1994.

- [22] 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.
- [23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proc. of Summer USENIX Conference*, 1985.
- [24] 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.