The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

# Fault Tolerance in a Distributed CHORUS/MiX System

Sunil Kittur, Online Media
Francois Armand, Chorus Systemes
Douglas Steel, ICL High Performance Systems
Jim Lipkis, Chorus Systemes

# Fault Tolerance in a Distributed CHORUS/MiX System

Sunil Kittur†
Douglas Steel
*ICL High Performance Systems, Manchester, UK*

François Armand
Jim Lipkis
*Chorus Systems, Saint-Quentin-En-Yvelines, France*

**ABSTRACT**

Within a distributed system, resources may be shared between nodes. The system should continue to operate even if individual nodes fail due to hardware or software errors. This may result in the loss of resources that were hosted on the failed node, but it may be possible to continue to provide access to some resources by hosting them on another node.

This paper describes mechanisms that allow the failover of resources from failed nodes. Failover is currently restricted to disk volumes and file systems. The failover mechanisms maintain the correct semantics at the UNIX system call level for operations from surviving nodes that were in progress at the time of the failure, including non-idempotent operations.

Minimal resource and performance overheads are imposed for the normal running case, and in contrast to replication techniques, state is recovered and rebuilt at the time of a failover.

## 1. Introduction

The GOLDRUSH system, developed at ICL, is a distributed memory multi-computer consisting of up to 64 nodes connected by a high-speed interconnect. Each node contains up to 12 SCSI disks that are only physically accessible by that node, and some nodes contain FDDI couplers which provide external access to the machine (Figure 1).

Each node is a self-contained UNIX system running a version of the CHORUS/MiX V.4 operating system, with some devices and file systems accessed from remote nodes using CHORUS IPC protocols [Rozier88, Batlivala92]. CHORUS/MiX CHORUS/MiX implements SVR4 UNIX as a ‘‘personality’’ on top of the CHORUS microkernel. It consists of a set of independent actors (a process-like abstraction) which run in supervisor mode. These actors include, the Process Manager (PM) which receives system calls from UNIX processes and acts as a client on behalf of these processes; and the Object Manager (OM) which provides file and device access.

The main application of the machine is to provide a parallel database server, and a number of commercial databases such as Oracle, Ingres and Informix have been ported to it. Administrative software is used to provide a consistent view of shared operating system resources across nodes.

To provide a highly available database service, the system must continue to function in the event of the failure of disks or nodes.

To provide resilience to individual disk failures, disk volumes are mirrored, so that if one of the mirrors fails, the volume is still available from the remaining mirror. The mirror may be a remote device; this functionality is provided by the Distributed Plex Manager (DPM) as shown in Figure 2. As mirroring techniques are fairly standard, the rest of this paper will discuss the handling of node failures.

Node failure can cause the loss of applications and disks/file systems that were hosted on that node. In the case of software errors, such as a kernel panic, it may be possible to reboot the node, and resume communication with the node, possibly performing some recovery actions. This is the approach used in Sprite [Baker94], NFS [Sandberg85] and Spritely NFS [Mogul92]. However, this can result in a considerable delay, since a reboot involves much more than the recovery of just the shared devices and file systems. This approach will also fail if the node cannot be rebooted, for example, if there was some permanent hardware failure.

† Author’s current affiliation is at Online Media, Cambridge, UK.

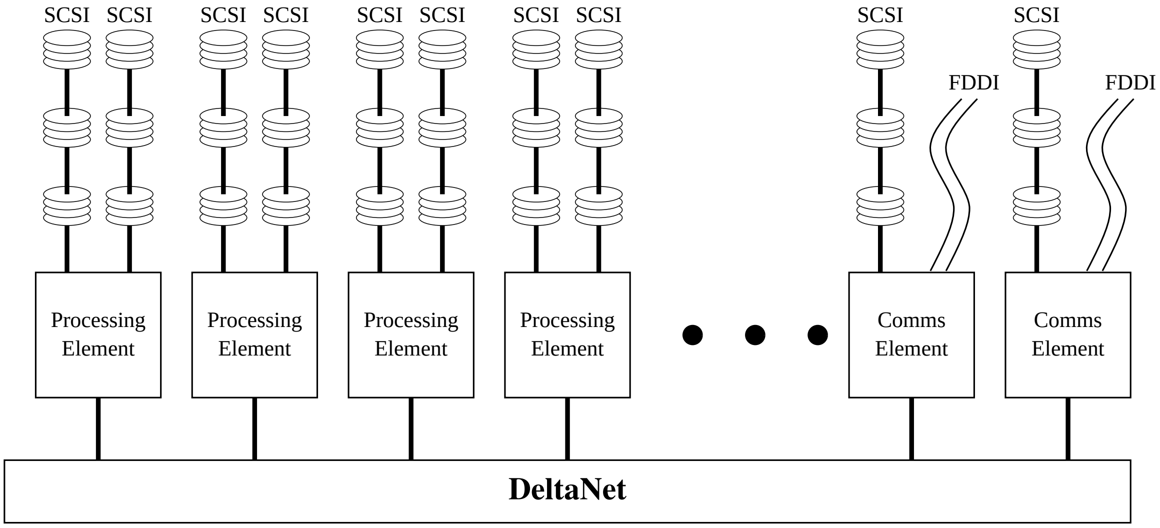


**Figure 1.** GOLDRUSH system architecture

Instead, we make the device or file system available from another node by a failover mechanism which allows access to the resource as soon as the appropriate recovery actions have been completed (Figure 3). This is similar in some respects to the HA_NFS work [Bhide91], but our approach covers more than just the NFS protocol.

It is important that this failover be transparent to the users of the resource, and in particular, the expected semantics of any operations that were in progress at the time of the failure must be preserved. This is not a problem for idempotent operations, since they can simply be retried once the failover has completed. However, non-idempotent operations, such as `unlink(2)` or `write(2)` on a file opened in O_APPEND mode should be retried only if the operation had not completed before the failure occurred.

## 2. Goals and Tradeoffs

We identified a number of key goals for GOLDRUSH:

- There should be no single point of failure that causes the entire system to be shut down.
- Applications not using resources on a failed node should continue without disruption.
- Performance and resource overheads should be minimised.
- The performance of normal operation is more important than the speed of recovery.
- The semantics of all UNIX APIs should be preserved during and after recovery.
- If a resource cannot be recovered, it should be cleanly removed, including cleaning up any state on surviving nodes.

Whilst satisfying these goals, our initial implementation makes a number of tradeoffs:

- The system does not tolerate double failures; if a second node fails before the first failure is recovered, the entire system may be shut down.
- The mechanisms are intended to cover only operating system resources; applications are responsible for handling the failure of application components that were on failed nodes.
- There is no need for instantaneous recovery; some short delay is acceptable. Providing instantaneous failover would require some form of replication, which would add considerable overheads to the normal running situation.
- The initial implementation covers disk volumes and file systems, but the mechanisms should be flexible enough to accommodate other types of resource.
- Data to remote file systems is written data-synchronously. This avoids having to recover from the loss of nodes containing dirty cached data. Synchronous writes are ideal in any case for database servers, which perform their own caching and have no reason to incur the cost of disk block copying required for buffered I/O. However, if general time sharing had been a goal, there would be a performance hit.

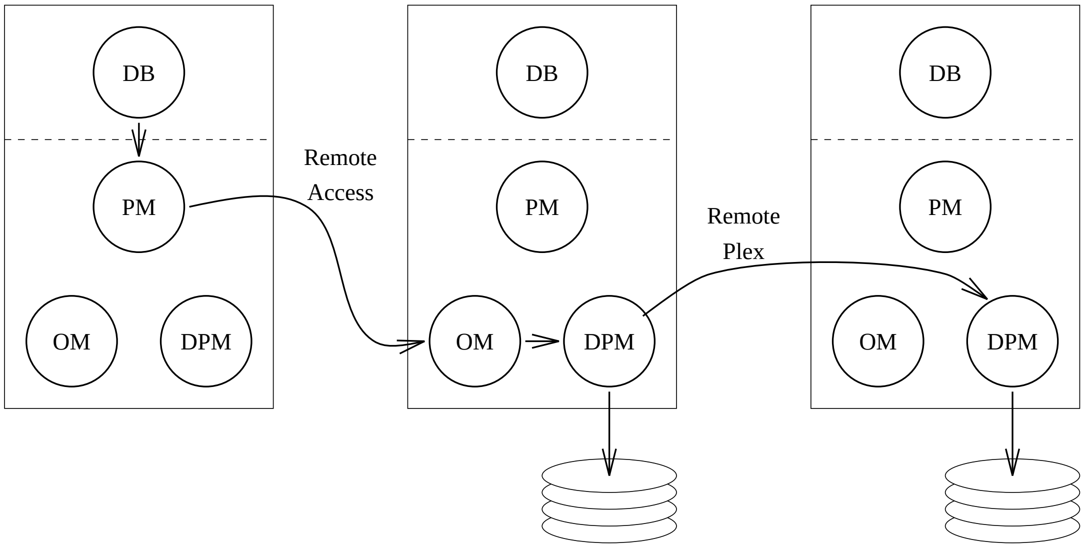


**Figure 2.** Access to a remote volume/filesystem

## 3. Architecture

The GOLDRUSH system can be viewed as a collection of homogeneous UNIX machines connected by a high-speed private network. The memory and disks on each node are only physically accessible by that node. Remote access is provided by software using CHORUS IPC. This is feasible because of the high bandwidth available.

Resilience to disk failure is provided using the VERITAS VxVM volume manager to provide mirrored disk volumes [vxvm93a, vxvm93b], and these volumes are made resilient to node failure by providing remote mirrors, using a remote driver access mechanism [Armand91][1]. In the event of a node failure, the volume is restarted on the node containing the remote mirror.

In order to provide a uniform device name space for volumes shared by nodes (global volumes), we have implemented a global device numbering scheme which allows a volume to be known by a globally persistent device number. After recovery by the backup node, the device is still accessible using the same device number.

Each node contains its own root file system, and hence its own independent file name space. By convention, we mount shared file systems on the same mount point name on each node, to present a shared global file name space. Files outside this shared name space are private to the node.

1. Although we provide this remote mirroring by software, the architecture can accommodate the use of multi-ported disks.

File systems that are created on global volumes can be accessed by all nodes by mounting the block device on a directory in the standard manner. Remote requests to the file system are forwarded to the server node using CHORUS IPC protocols.

A remote file system is mounted using the same command line arguments as if it were on a local device, and the kernel performs the appropriate internal remote connection protocols. If we used a scheme like NFS which specifies the host and pathname on the host, we would have to update user visible data such as `/etc/mnttab` to reflect the new host after failover. With our scheme, the user visible state remains the same, namely the block device name, mount point name and mount options.

Failover of a file system from a failed node requires the volume to be started on the backup node, and the file system to be recovered. We use the VERITAS VxFS file system [vxfs92] to provide fast recovery using its intent log. The recovery required for failover is slightly different to the normal recovery that fsck performs, since it must preserve state held by active requests that fsck could throw away, such as unlinked files that are still open.

A number of components are used to provide these mechanisms:

- A distributed failover manager (FOM), which maintains the status of resources, whether they are active, failed, or in the process of being recovered. The FOM also maintains a list of clients who are using the resource.

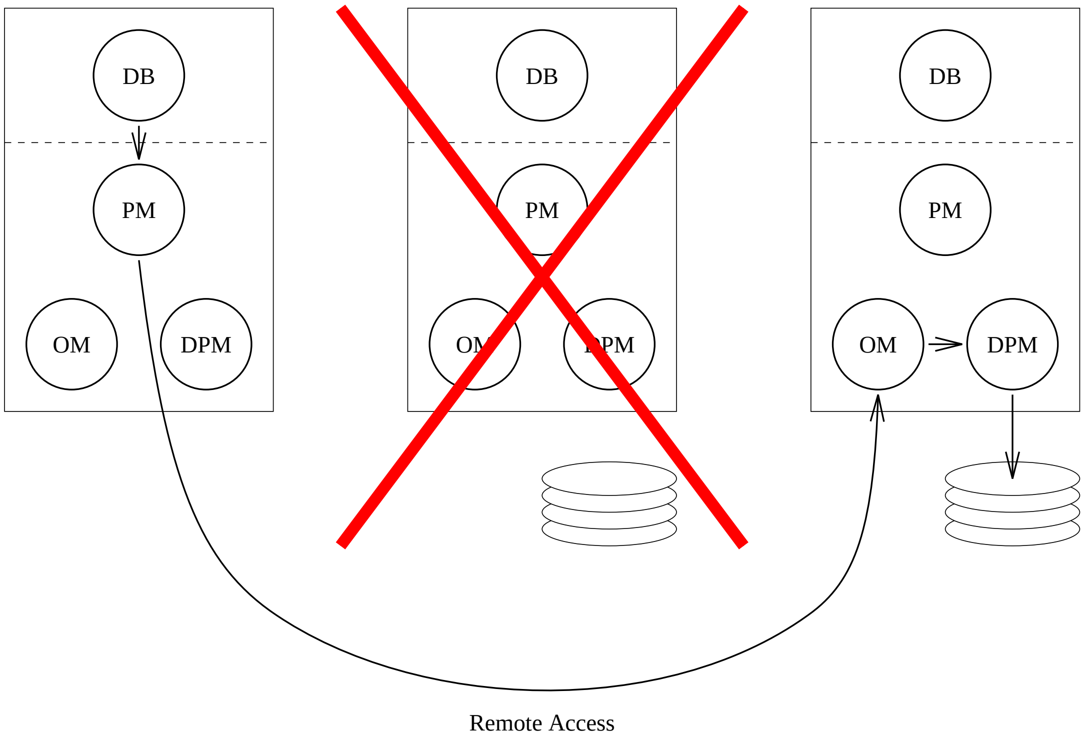


**Figure 3.** Access to a remote volume/filesystem after a failure

- A library linked with clients which handles the binding between clients and servers. The library is used to monitor IPC requests and detects failures using timeouts or by detecting link failures. After a failover, the library transparently rebinds the client to the new server.
- A state recovery protocol which allows servers to reconstruct state from information provided by clients.
- Modifications to server modules for server specific recovery actions.
- A mechanism for handling non-idempotent operations, to allow a backup server to determine if the operation was already completed before the failure.

## 4. Failure Detection and Handling

A set of resources and associated interfaces can be described as a *service*. At any given time a service is hosted by a single server, running on a particular node. An example of a service is a file system or raw disk device.

In general the servers in our system are the file Object Managers on each node, and clients are the Process Managers on each node. Object Managers may also be clients of a Process Manager, for example, to obtain process credentials, or to handle the `/proc` file system operations [Killian84].

A service can be fault-tolerant or not; a fault-tolerant service is available (after some recovery action) even if the server currently hosting it fails. A service is the unit of failover, and if a particular server hosting multiple services fails, the individual services can be failed over to different backup servers. This can provide a better load balance between remaining nodes which pick up only a part of the activity of the failed node.

Services are identified by a unique identifier or name, in our case, a CHORUS unique identifier (UI) that is unambiguous across all nodes. This service UI is persistent during the lifetime of the system, and is persistent across a failover in the case of a fault-tolerant service.

The FOM is responsible for tracking the status of services, and maintains a list of clients using each service. The FOM is a global entity, distributed across all nodes. It is informed by servers when they first create a service, and by clients when they first access a service.

We provide a library, linked with each client, that monitors IPC requests. If these time out, or receive an error indicating the destination has failed, the FOM is notified.

The FOM determines whether the service has failed, and if it has, it informs all clients of the service, who then suspend all communication with the service. This is handled within the client-side library, and does not require any special action within the client.

It is important that a central agent determines the failure, as it is possible for conflicting error reports to be generated. For example, two nodes may be unable to communicate with each other, and each reports that the other has failed.

If the service is fault-tolerant, the FOM elects a suitable node to host the service, and instructs a server on that node to recover it. Once the recovery is complete, the clients are notified, and resume communication with the new server, including any requests that were blocked during the recovery.

If the service is not fault-tolerant, or the recovery failed, the clients are informed, and they mark the service as unavailable. This results in all requests returning with an error until the service is removed, for example, by unmounting a failed file system.

Servers are also prepared to handle the failure of their clients, detected by the fault detection library, and if necessary, clean up any state held on behalf of the client, for example, `vnode` references.

## 5. The Failover Manager

The Failover Manager (FOM) is responsible for detecting failures and initiating the failover procedure. It needs to be available at the time of a failure and hence needs to be resilient to node failure. The FOM is implemented as a set of daemons and actors. Some of these daemons run on an external host system.

The main components of the FOM are:

- *Error Recovery Manager* The ERM is the main interface for the kernel components and runs on the nodes. It is resilient to failure by the use of a number of hot backup ERM's running on other nodes. A master ERM is elected and serves as the consensus point for determining node failure and the coordinator of the failover process.
- *Disk Management Daemon* The DMD component knows which disk is connected to which node and where any second plex may be found. The DMD runs on the host.
- *Configuration Server* Conserv knows what the desired system should look like and what resources are critical to the users of the system (e.g. the system should not continue if it loses both plexes of a specified volume). Conserv runs on the host.

Failure reports are sent by kernel components to the ERM, for example, a PM noticing an IPC timeout, the communications mechanism noticing a communications failure. When a failure has been established the ERM informs all clients of services on that node that the services have temporarily failed. Next, the ERM checks with Conserv if the system is still viable, (e.g. it hasn't lost any critical components). If the system is viable failover can proceed, otherwise the whole system needs to be shutdown and restarted. The ERM then informs the DMD of the node failure which launches the backup services on the site of the second plexes.

The host is involved with startup, shutdown and failure and is hence a critical component of the GOLDRUSH system; if it fails, no other node failure will not be recovered until the host has recovered. The host is a standard SVR4 machine that does not run any user services and can therefore be recovered by rebooting it[2].

## 6. Distributed State and Recovery

Each type of service imposes its own requirements for recovery. In general, the service must be recoverable to the state it was in before the failure occurred. This means that all volatile state maintained by the service must be identified, and somehow recreated in the backup server. Cached information does not need to be recovered, since this affects only the performance of the service.

Our strategy is to move as much of this volatile state as possible to the clients, and use some persistent storage for the rest. In the case of file systems, this includes such things as file and `vnode` reference counts, file seek offsets and directory blocks and so on. Apart from the file system specific data (incore inodes and so on), much of this state can be moved to the clients [Welch90].

During recovery, the file system can be recovered to a consistent state using fsck, and the reference counts and incore inodes can be recreated by recovering using client information [Baker94]. Once the volatile state has been reconstructed, the server can begin to receive new requests and retries of requests that were pending at the time of the failure.

2. The host is not a single point of failure. However permanent loss leaves the system unable to recover from node failure. A future goal is to fix this.

We use the VxFS file system, which uses transactions in an intent log to record all updates to file system structures. This provides very fast recovery, since completing the transactions recorded in the log will bring the file system to a consistent state. However, not all file system operations are atomic, and some minor changes were required to fsck to support failover. An example is unlinking an open file; the directory entry is removed, but the freeing of the inode is deferred until the last reference is released. During failover, this inode removal must be performed only once it has been determined that no remaining clients are referencing the file. The changes to fsck amounted to a new command line option, about 30 lines of code, and a new super block state that indicates that kernel level recovery is required to check reference counts of unlinked inodes.

Using a standard file system such as UFS [McKusick84] would create a number of problems. Firstly, without the atomicity of transaction-oriented disk update, it would be much harder to manage non-idempotent system calls. Secondly, the recovery would be much slower, since the fsck must scan the whole disk to repair the file system. In addition, the actions required to repair the file system can often result in the unpredictable loss of files, which complicates recovery.

Once the file system on disk has been made consistent, the kernel state must be recovered. Each client of the file system is contacted, and replies with reference counts and inode numbers of files it is using from the file system. These can be used to recreate the incore inodes.

A number of changes were made to the VFS interface [Kleiman86] to allow for file system specific actions during this recovery:

- When the file system is re-mounted, a new flag is passed indicating that this is for a failover. This allows the file system specific code to perform any special actions.
- A new VFS routine is used to recreate an incore inode and vnode with the correct reference counts and object capability.
- A new VFS routine is used to perform any file system specific activity once all the incore vnodes have been created. In the case of VxFS, this checks for deferred removals.

## 7. Non-idempotent Operations

Failed calls from a client to a server are retried when the failover service has resumed activity. Provision is made to prevent the system from executing the same non-idempotent requests twice.

Most existing systems use a server-side log which is duplicated on the backup server site. Instead we use a so-called ‘‘intent-message’’ mechanism with the support of the client side library.

When a non-idempotent request is received for the first time (ie. before a failure occurs), an upcall is performed to the client before committing the VxFS transaction on the inode. The ‘‘intent’’ message sent in this upcall identifies the request being processed, and contains a modification counter associated with the inode, as well as the expected return value of the request. The modification counter is then incremented, and the transaction is committed to disk, which also records the new counter value. The server then replies normally to the request.

If the transaction was completed before a failure, or if recovery rolls forward the transaction, the modification counter associated with the inode will have been incremented. If the client had not received the reply from the failed server, it will retransmit the request to the backup server; this retry request will contain the intent message sent by the failed server.

When the server receives this retry, it can determine whether the operation has already been completed, by comparing the modification counter of the inode with that in the message[3] ; if the counters are the same, the operation had not completed, and is retried, otherwise the return value saved in the intent message is returned to the caller.

This simple mechanism has some interesting properties:

- Unlike log mechanisms, there is no log compression or cleanup issue, since the intent message information is kept with the client's current request state. No extra allocation, deallocation, or management for the ‘‘log’’ memory is required.
- Since retry requests are modified with the contents of the intent message, there is no extra work for the server to determine whether this request is a retry of a non-idempotent request or not. There is no need to scan a server side log. All the information

3. The value of the counter at recovery time is used, since it may have evolved after recovery due to other intervening non-idempotent requests.

needed is within the message.

## 8. Comparison With Related Work

A number of other systems address the issues of high availability and fault tolerance in distributed environments. However, we believe our work differs from existing experiences in the following areas:

- Our system maintains the full UNIX semantics across distribution and recovery (stronger semantics than NFS based implementations).
- Client side caching is under control of the applications, not under control of the system, since applications are database servers.
- Replication is done at the logical volume level, not at the file level.
- We have based our implementation on commercially available software and have completed this mainly by introduction of a client side logging mechanism. The systems referenced below use mostly server side logging.
- We have introduced the notion of service and are able to deal with recovery of an individual service, using a generic failover manager which can be extended to services that are not necessarily file system based. None of the systems listed below appear to have worked in that direction.
- Our system supports (today) disk devices as well as filesystems.
- We do not rely on a new file system implementation (though VxFS was modified somewhat).

Sprite provides separates client and server state in a similar manner to our system [Welch90]. During recovery, each client must perform a re-open protocol on a per-file basis to provide the server with reference counts and so on, and to obtain a valid handle to the server object [Baker94]. However, in our system, the client handles are persistent, and the server only requires the reference counts and so on. This allows us to package the state for all files on the client into one RPC message, which reduces the amount of server congestion during recovery.

Spritely NFS [Mogul94] and Not Quite NFS [Mack94] are mostly based on NFS even though they extended NFS semantics. Anyhow, they do not make use of data mirroring, thus clients have to wait for the failed server to be up and running again before they can resume their activity.

HA_NFS [Bhide91] is closer to what we do for the failover mechanism. However it supports only NFS requests, and requires some hardware so that the backup machine appears with the same IP address as the failed node. This also implies that all resources of the failed node will be backed by the same node.

DECEIT [Siegel90] is mostly based on NFS and thus has not had to deal with some of the issues we had to face, such as non-idempotent requests. DECEIT is also based on file replication rather than disk duplexing.

HARP [Liskov91] is based on file replication with a 2 phase-commit protocol from the primary machine to the secondary machine. A log is maintained in both sites and requires a Uninterruptible Power Supply. We have not found any detail on how clients detects failures and how they switch from primary to backup.

ECHO [Birrell89] and FICUS [Guy90] fall in the same category as the HARP file systems, being based on replication of files with secondary servers being synchronized with the primary one

CALYPSO[Devara94] is probably the closest system to ours, although there are a few differences. They use a three-phase recovery mechanism similar to our FOM, except that in GOLDRUSH, the state recovery is under the control of the backup server; the server can use the most appropriate mechanism for its recovery, for example, it can collect state from clients, or it can recover from a log filled by the crashed server. CALYPSO seems to be less flexible, requiring that server state is rebuilt from client caches. CALYPSO deals ithe site crashes, and the activity of one site is taken over by another site; the notion of service as used in GOLDRUSH is a major improvement which allows the load of a crashed site to be spread among several surviving nodes.

CODA [Satya91] is mainly oriented towards disconnected operations and does not maintain UNIX semantics. Since most of the design is based on the existence of cache local to the client, the approaches are hardly comparable.

## 9. Performance Measurements

It has not been possible to completely isolate the effects of the fault tolerance mechanism as a large number of other changes have also been made over our previous system. We have been tuning the system with an emphasis on raw device access since this is the most performance critical for our applications. Our applications also mainly use read and write operations.

| Call overhead for 512 byte raw device access | | | |
|---|---|---|---|
| Call | resilient access (ms) | standard access (ms) | overhead (%) |
| write | 0.264 | 0.262 | 0.7 |
| read | 0.147 | 0.145 | 1.4 |
| open | 1.82 | 1.72 | 6.8 |

The overhead of using the fault detection library is minimal for the most frequent raw device accesses (around than 1% for read and write). However FOM interaction involves a larger overhead for open and mount (around 7%). This interaction involves communication with the ERM to declare the clients and services, and subscription to client and service failure. The ERM logs all operations to a file thus adding to this overhead.

The recovery time for a single raw volume is on the order of 2 seconds. The recovery time for VxFS is around 3 seconds for a 1GByte volume, and is proportional only to the size of the intent log. VxFS recovery is performed only after the raw volume has been sucessfully recovered.

The total recovery of a typical system (16 nodes, 10 user disks per node (duplexed), 10 volumes per duplexed disk group) after a node failure with the backup services being launched evenly across the remaining nodes is on the order of 20 seconds for raw volumes. VxFS recovery typically adds around another 10 seconds to this (typically only a few volumes have filesystems). This is single threaded on each node, but in parallel across the nodes.

## 10. Experiences and Further Work

Although our work utilised the inherently distributed nature of CHORUS/MiX we believe it can be adapted to other systems including monolithic kernels.

Although we had to port VxFS to the CHORUS/MiX environment, we found it had a number of useful properties - its transactional nature allows most operations to be treated atomically, simplifying recovery, and the intent log provides very fast recovery time, on the order of a few seconds.

The current system has been implemented and is currently undergoing system test. We are tuning the system with the empahsis on raw device access performance since this is the most critical for our applications.

We have yet to fully measure the performance of the system and are in the process of developing our measurement techniques to isolate the overhead of non-idempotent calls.

We found the performance of frequently used operations to be reasonable. It will be possible to further improve performance by moving the ERM into kernel space as a CHORUS supervisor actor, thus alleviating a large number of context switches especially on mounting a filesystem or the first open of a raw volume.

## 11. Acknowledgements

We would like to thank the efforts of the people who contributed to the design and implementation of the mechanisms described in this paper: Brian Anthony, Richard Harry, Martin Hogg, Simon McKenna, David Messham, Steve Noble and Iain Robertson at ICL, and Jean-Marc Fénart, Ruby Krishnaswamy, Pierre Lebée and Gilles Maigné at Chorus Systèmes.

## 12. Biographies

Sunil Kittur received his BSc in Computer Science in 1988 from University College London. He spent the next 4 years at the Santa Cruz Operation, working on the SCO XENIX, UNIX and MPX kernels. In 1992 he joined ICL High Performance Systems as a senior engineer on the GOLDRUSH project. He has recently joined Online Media as principal software engineer working on the OS for a distributed interactive multi-media system. His email address is skittur@omi.co.uk.

François Armand received his Engineer diploma in 1977 from ENSEEIHT in Tolosa, France. He spent a few years in a software house and joined the Sol research project at INRIA, working on Unix V7, in 1980. He then moved to the Chorus research project in 1985, and since 1987 has been at Chorus Systems working on aspects of the Unix subsystem for the CHORUS microkernel. He has been mostly involved in the design of the distributed Unix and more recently in failure resilience issues. His email address is francois@chorus.fr.

Douglas Steel completed his BSc in Computing Science in 1988 from Glasgow University. He spent 4 years as a research assistant at Queen Mary & Westfield College (University of London) investigating operating system support for distributed object oriented programming, and completed a part-time MSc. He joined Unix Systems Laboratories Europe and worked on the Esprit Ouverture project. He joined ICL High Performance Systems in late 1993 as a senior engineer on the GOLDRUSH project. His email address is doug@wg.icl.co.uk.

Jim Lipkis has been involved in operating system design for parallel, real time, and fault tolerant systems. At New York University he worked on OS and language software for scalable shared-memory multiprocessors such as the NYU Ultracomputer. Since 1989 he has been a senior engineer and architect at Chorus Systems, working on microkernel design and on application of the microkernel to systems ranging from embedded real time to supercomputing to highly available parallel database servers. His email address is lipkis@chorus.fr.

## REFERENCES


[Armand91] F. Armand, "Give a Process to your Drivers!", Proc. of the EurOpen Autumn 1991 Conference.

[Baker94] Mary Baker, "Fast Crash Recovery in Distributed File Systems", PhD Thesis, Univ. California at Berkeley, 1994.

[Batlivala92] Nariman Batlivala et al., "Experience with SVR4 Over Chorus", Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, April 1992, pp.223-242.

[Bhide91] Anupam Bhide et al., "A Highly Available Network File Server", Proc. of the Winter 1991 USENIX Conference, pp.199-218.

[Birrell89] A. Birrell et al., "Availability and Consistency Tradeoffs in the Echo Distributed File System", Proc. of Second Workshop on Workstation Operating Systems, pp.49-54.

[Batlivala92] N. Batlivala et al., "Experience with SVR4 over Chorus" Proc. of 1st Workshop on Microkernels and Other Architectures, Usenix, Seattle 1992.

[Devara94] M. Devarakonda , B. Kish, A. Mohindra "Non-Disruptive Server Recovery in Calypso File System" IBM Researh report RC 19794(87665) 10/19/94

[Guy90] Richard G. Guy et al., "Implementation of the Ficus Replicated File System", Proc. of Summer 1990 USENIX Conference, pp.63-72.

[Killian84] T.J. Killian, "Processes as Files", Proc. of the Summer 1984 USENIX Conference.

[Kleiman86] S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", Proc. of the Summer 1986 USENIX Conference, pp.238-247.

[Liskov91] Barbara Liskov et al., "Replication in the Harp File System", Proc. of the 13th ACM Symposium on Operating Systems Principles, pp.226-238.

[McKusick84] M.K. McKusick et al., "A Fast File System for UNIX", ACM Transactions on Computer Systems Vol.2 No.3, August 1984, pp.181-197.

[Mack94] Rick Macklem, "Not Quite NFS, Soft Cache Consistency for NFS", Proc. of the Winter 1994 USENIX Conference, pp.261-278.

[Mogul92] Jeffrey C Mogul, "A Recovery Protocol for Spritely NFS", Proc. of the USENIX File Systems Workshop, May 1992, pp.93-110.

[Mogul94] Jeffrey C. Mogul, "Recovery in Spritely NFS", Computing Systems, Spring 1994, pp.201-262.

[Rozier] M. Rozier et al., "Chorus Distributed Operating Systems" Computing Systems 1(4), December 1988.

[Sandberg85] R. Sandberg et al., "The Design and Implementation of the Sun Network File System", Proc. of the Summer 1985 USENIX Conference, June 1985, pp.119-130.

[Satya91] M. Satyanarayanan et al., "Disconnected Operation in the Coda File System", Operating Systems Review, vol. 5, pp.213-225.

[Siegel90] Alex Siegel, "Deceit: A Flexible Distributed File System", Proc. of Summer 1990 USENIX Conference, pp.51-62.

[vxfs92] Veritas Sofware Corporation, "VERITAS File System (VxFS) System Administrator's Guide Release 1.2.1", 1992.

[vxvm93a] Veritas Software Corporation, "VERITAS Volume Manager (VxVM) Basic User's Guide Release 1.2", 1993

[vxvm93b] Veritas Software Corporation, "VERITAS Volume Manager (VxVM) System Administrator's Guide Release 1.2", 1993

[Welch90] Brent B Welch, "Naming, State Management, and User-Level Extensions to the Sprite Distributed File System", PhD Thesis, Univ. California at Berkeley, 1990.