DRAFT: Boardwalk Design Specification: I/O System

Nick Christenson
Scott Lystig Fritchie
Jim Larson
Philip Guenther
Jason Evans

May 8, 2001

Abstract

This document no longer describes anything Sendmail, Inc. is working on, and not by modification of the document.

Contents

1 Overview **2**
1.1 Purpose of This Document . . . 2
1.2 DI/ODE Goals and Features . . . 2
1.2.1 Interfaces . . . 2
1.2.2 Data Striping . . . 3
1.2.3 Data Migration . . . 3
1.2.4 Redundancy . . . 3
1.3 Cluster Architecture . . . 3
1.3.1 Access Server . . . 4
1.3.2 Data Server . . . 4
1.3.3 Metadata Server . . . 4
1.3.4 Command Server . . . 5
1.4 Software Architecture . . . 5
1.4.1 Client Library . . . 5
1.4.2 Client Daemon . . . 6
1.4.3 Data Server Daemon . . . 6
1.4.4 Metadata Server Daemon . . . 6
1.4.5 Command Server Shell . . . 6

2 Client Protocol **7**
2.1 File Client Protocol . . . 7
2.1.1 Design Philosophy . . . 7
2.1.2 Synopsis . . . 7
2.1.3 Connection Setup . . . 11
2.1.4 Symbolic Links . . . 11
2.1.5 Status Codes . . . 12
2.2 Sleepycat RPC Protocol . . . 13

3 Application-Side Libraries **16**
3.1 File Client Library . . . 16
3.1.1 Interface Styles . . . 16
3.1.2 Supported System Calls . . . 17
3.1.3 Path Canonicalization . . . 18
3.1.4 Client Library Internal State, File API . . . 20
3.1.5 Support For Multiple Client Daemons . . . 22
3.1.6 Authentication Issues . . . 22
3.1.7 Internal Locking . . . 22
3.1.8 Unsupported Calls and Semantics . . . 23
3.1.9 Supported STDIO Calls . . . 25

3.1.10 Other Calls . . . 25
3.1.11 Current Implementation Sketch and Details . . . 25
3.2 Sleepycat DB Interface . . . 25
3.2.1 List of Supported Sleepycat Calls . . . 25
3.2.2 Client Library Internal State, DB API . . . 25
3.2.3 A Note on Transactions . . . 26
3.3 Multithreaded RPC Client Support . . . 26
3.3.1 Goal . . . 26
3.3.2 Existing work . . . 26
3.3.3 Thread and Synchronization Model . . . 26
3.3.4 Data Structures and Memory Management . . . 27
3.3.5 API . . . 28

4 Client Daemon 29
4.1 Common Overview . . . 29
4.2 Consistent Hashing . . . 29
4.3 File Architecture . . . 30
4.3.1 Session State . . . 30
4.3.2 Open File Table . . . 32
4.3.3 Dnode Table . . . 33
4.3.4 NFS Mount Table . . . 34
4.3.5 Consistent Hash Bucket Map Table . . . 34
4.3.6 Namei . . . 34
4.3.7 Locking . . . 34
4.3.8 Linking and Renaming . . . 35
4.3.9 Process Tree . . . 35
4.3.10 Migration . . . 35
4.4 Database Handling . . . 35
4.4.1 Theoretical constraints . . . 35
4.4.2 Categorization of Sleepycat DB RPC messages/responses . . . 37
4.4.3 Broadcasted DB operations . . . 39
4.4.4 Multiplexed DB transactions . . . 39

5 Data Protocol 41
5.1 File Interface . . . 41
5.1.1 Implementation . . . 41
5.2 Sleepycat DB Interface . . . 41
5.2.1 Goals . . . 41
5.2.2 The Threading Constraint . . . 42
5.2.3 Prior Work . . . 42
5.2.4 Implementation . . . 42
5.2.5 Unresolved Issues . . . 61
5.2.6 Problems . . . 61

6 Command Server 63
6.1 Configuration . . . 63
6.1.1 Access Server Information . . . 63
6.1.2 Data Server Information . . . 66
6.1.3 Metadata Server Information . . . 66
6.2 Command Line Utilities . . . 66
6.3 Interactive Command Vocabulary . . . 68

6.4 Monitoring . . . 68
6.4.1 Data Servers . . . 69
6.4.2 Metadata Servers . . . 69
6.4.3 Access Servers . . . 69
6.5 Implementation . . . 69
6.5.1 Application Structure . . . 69
6.5.2 Process Tree . . . 70
6.5.3 Global Tables . . . 70
6.5.4 Event and Alarm Handling . . . 71
6.5.5 Report Browsing . . . 71
6.5.6 Crash and Restart . . . 71

7 File Migration 72
7.1 File Migration Problems . . . 72
7.1.1 Why Directory Renaming Is Evil . . . 72
7.2 The Freezing Process . . . 73
7.2.1 The Freezing Process, RPC Credentials, and File Ownership . . . 73
7.2.2 File ownership strawman for Boardwalk 1.0 . . . 74
7.3 Migrating a directory . . . 74
7.4 Migrating a file . . . 75
7.4.1 `EPERM` note . . . 76
7.4.2 RENAME and Migration . . . 76
7.5 Migration Phase 0 . . . 76
7.6 Migration Phase 1 . . . 76
7.6.1 Phase 1 Operation Rules . . . 77
7.6.2 Directory Renaming . . . 77
7.6.3 Other Possible Solutions . . . 77
7.6.4 The Boardwalk 1.0 Solution . . . 77
7.7 Migration Phase 2 . . . 78
7.8 Variations on Migration Techniques . . . 78
7.8.1 Migration: “Directory at a Time” or “File at a Time”? . . . 79
7.9 Phase 2 Operation Rules . . . 79
7.9.1 REMOVE . . . 79
7.9.2 WRITE . . . 79
7.9.3 SETATTR . . . 79
7.9.4 CREATE . . . 79
7.9.5 MKDIR . . . 79
7.9.6 RMDIR . . . 80
7.9.7 RENAME . . . 80
7.9.8 LINK . . . 80
7.9.9 READDIR . . . 80
7.9.10 Other read-only operations . . . 80
7.9.11 Tagging “Migration Done” In Directories In OLD . . . 80
7.9.12 Tagging “Migration Done” In Directories In NEW . . . 81
7.9.13 Phase 3 Cleanup Sweep . . . 81

8 Metadata Migration 82
8.1 Phase Change Algorithms . . . 82
8.2 Phase one algorithms . . . 84
8.3 Phase two algorithms . . . 86

9 Future Work **88**
9.1 Redundancy Considerations 88
9.1.1 Types of Redundancy 88
9.1.2 Implications for Client Daemon Design 90
9.1.3 Implications for Metadata Server Design 91
9.1.4 Implications for Command Server Design 91
9.1.5 Implications for Application Design 91

A DI/ODE Client Protocol RPC Definition **92**

List of Figures

1.1 Boardwalk Components 4
1.2 DI/ODE Components 5

5.1 Database Structure Hierarchy 44
5.2 RPC Server Structure Hierarchy 45

7.1 File Attribute Encoding 75

List of Tables

2.1 DI/ODE File Protocol 8
2.2 Sleepycat RPC Protocol 14

3.1 DI/ODE File API and Protocol Mapping—OS Independent Calls 19
3.2 DI/ODE File API and Protocol Mapping—OS Dependent Calls 20

4.1 Translation of File Protocol to NFS Operations (simplified) 31

Chapter 1

Overview

1.1 Purpose of This Document

This document describes the design of the Distributed I/O Development Environment (DI/ODE) I/O system implemented for the Boardwalk project. This document is mainly meant for developers who will be working with the code, but it will also be useful for QA when doing white-box testing. It can also be used to assess the feasibility of using DI/ODE technology for other projects.

1.2 DI/ODE Goals and Features

DI/ODE is an architecture for adopting conventional applications to work in a scalable, reliable cluster setting. It provides an interface to existing applications with the following goals:

- Minimal modification to the existing application(s)
- Arbitrary horizontal scalability
- On-the-fly expansion and contraction of the storage capacity and bandwidth
- Centralized management of the distributed system
- Provide a redundant network path between the servers on which the applications run and on which the data are stored
- Provide redundant data storage

The last two goals are not met in this release of the DI/ODE software, but the present release supports a straightforward path to implement them in the future.

1.2.1 Interfaces

DI/ODE integrates into existing applications by using standard API's, although with some nonstandard restrictions and semantics.

File I/O

DI/ODE presents a set of file hierarchies to the application, much as if they were mounted into the filesystem. However, this mounting happens at the application level rather than the kernel level. DI/ODE accomplishes this by intercepting the standard file I/O API for an application that it is

linked with. It emulates most of the I/O system calls (e.g., `read()`, `write()`, etc..) and most of the system Standard I/O library (e.g., `fprintf()`, `fseek()`, `mktmp()`, etc.). If those operations are performed on a file in the emulated region of the system's file namespace, the DI/ODE system handles them. Otherwise, the underlying standard file system calls are invoked.

DI/ODE file storage resides on a set of Data Servers. This set can be expanded or contracted on-the-fly. When this is done, the data stored on these machines will be redistributed to balance the load using a process we call *Data Migration*. The Data Migration process is explained in detail in Chapter 7.

Sleepycat I/O

DI/ODE also presents a database service, using the Sleepycat DB API. Taking advantage of Sleepycat's RPC facility and a multiplexer of our own design, we allow a set of back-end DB servers to appear as a single huge database. The Sleepycat I/O system is also capable of on-the-fly data migration This process is described in Chapter 8.

1.2.2 Data Striping

For both the file and Sleepycat storage, DI/ODE transparently stripes the data across a set of back-end storage resources. This allows the illusion of a single huge storage system. Rather than using a centralized registry, the back-end location of each datum stored is determined algorithmically. We expect the Law of Large Numbers to ensure that the data distribution will be sufficiently smooth.

1.2.3 Data Migration

Unlike other transparent striping mechanisms, DI/ODE allows the administrator to add or remove back-end storage servers while the system is running. The storage load will automatically re-balance across the new set of back-end storage servers. This process is called *migration* and is detailed in Chapter 7 and Chapter 8.

1.2.4 Redundancy

In the future, we will have redundancy of system components sufficient to ensure that the system has no single point of failure.

Data Network redundancy, that is, having multiple network paths between the Access Servers and the Data and Metadata Servers, is not implemented as a feature in the Boardwalk release of DI/ODE. It would be possible to add this feature in a minor revision number of a successive release of DI/ODE if it became a priority to do so.

Data redundancy is not implemented as a feature in the Boardwalk release of DI/ODE. Adding this feature requires significant rework of key DI/ODE components. In any release where this feature is added, it will be *the* major thrust of that release, perhaps excluding all other feature additions. Therefore, a DI/ODE version with redundancy will have a new major release number associated with it. It could be implemented as the key feature for version 2.0 if that were the top priority.

1.3 Cluster Architecture

Figure 1.1 illustrates the roles of various hosts in a DI/ODE cluster. Each machine in the figure has a specific role.

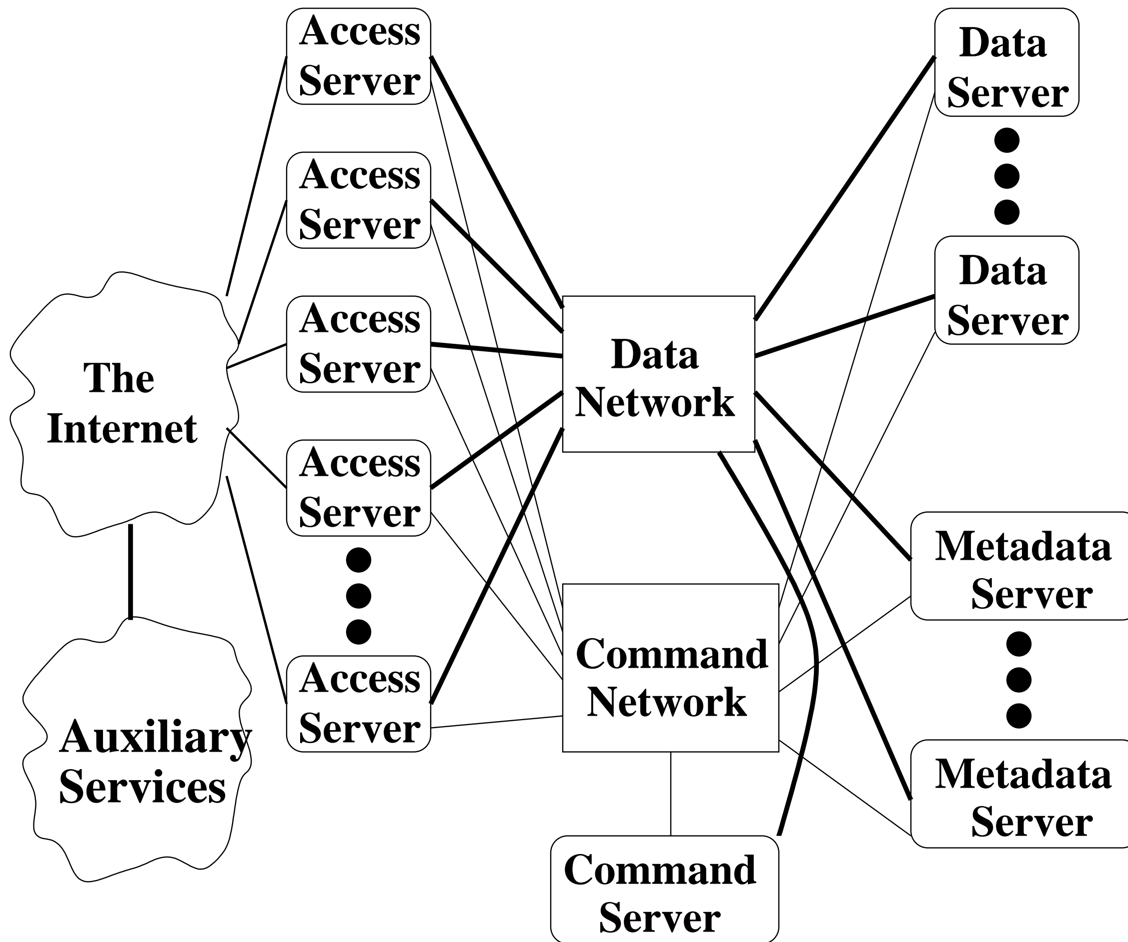

Figure 1.1: Boardwalk Components

1.3.1 Access Server

The Access Servers run applications linked with the Client Library or Sleepycat Library with the DB Environment set to connect to a remote server via RPC. Also running on the Access Server is the Client Daemon.

1.3.2 Data Server

The internal interfaces to the Data Server are NFS v3. The Client Daemons speak NFS v3 over UDP to the Data Servers and transfer file state to and from them via this protocol. The Command Server speaks NFS v3, also over UDP, to the Data Servers. The "/" partition of each Data Server is NFS mounted on the Command Server, and changes to the configuration of each Data Server are made by modifying the representation of that file on the Command Server.

1.3.3 Metadata Server

The Metadata Server is a UNIX based server with a daemon that receives requests from the Client Daemons on the Access Servers via the Sleepycat DB/RPC protocol. The Metadata Server Daemon

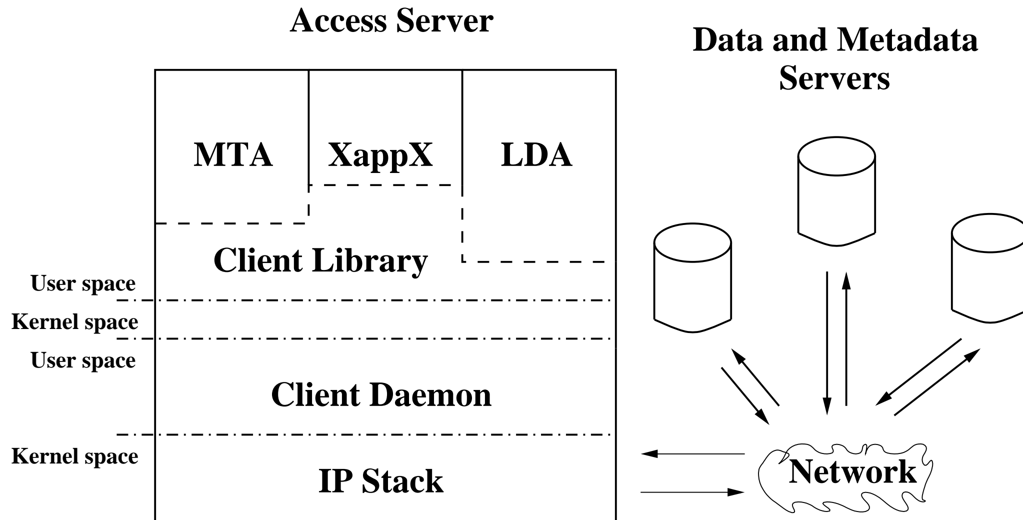

Figure 1.2: DI/ODE Components

unwraps the DB calls and applies them against the DB environments which are stored locally.

1.3.4 Command Server

A Command Server node in the system coordinates the work of all the Client Daemons and controls their state transitions as they perform migrations. It is also the monitoring and reporting center of the DI/ODE system. The Command Server is not critical for normal operation.

1.4 Software Architecture

The DI/ODE software components and their interactions are illustrated in Figure 1.2.

1.4.1 Client Library

The Client Library turns standard file I/O calls into RPC requests to the Client Daemon. The Client Library is written in C and is thread-safe.

The Client Library intercepts the standard file I/O API for an application that it is linked with. It emulates most of the I/O system calls (e.g., `read()`, `write()`, etc..) and most of the system Standard I/O library (e.g., `fprintf()`, `fseek()`, `mktmp()`, etc.). If those operations are performed on a file in the DI/ODE controlled region of the system's file namespace, then the Client Library turns these into ONC RPC requests and sends them to the Client Daemon for processing. If those operations are performed on a file in the portions of the file system the local operating system is responsible for, then the Client Library falls through to their locally defined behavior. In this way, a `read()`, operating on a file that resides on a Data Server will get turned into a Client Daemon request sent to the appropriate Data Server, while the same `read()` operating on a file local to the system (for example, `/etc/motd`) will operate directly on that file as if the Client Library were not linked in with that application.

More than one Client Daemon may run on any Access Server. The Client Library will select which one to contact, attempting to balance the load over each possible Client Daemon. The mechanism by which this happens is described in Section 3.1.5.

Similar in function to the Client Library, we will use the Sleepycat Berkeley DB library configured as an RPC client to connect to the Client Daemon as an RPC server. The Client Library will intercept calls to the Sleepycat `DBENV→set_server()` method and redirect it to create an RPC connection with one of the Access Server's Client Daemons.

1.4.2 Client Daemon

The Client Daemon more than any other piece exemplifies the DI/ODE system. One or more Client Daemons run on each Access Server. Applications linked against the Client Library will send I/O requests, both file and Sleepycat DB, through the Client Daemon(s) which will route them to the appropriate Data Server or Metadata Server.

The connections between the Client Library and the Client Daemon all carry ONC RPC, via either a TCP stream over the loopback interface or a Unix domain stream socket within the file system. The RPC protocols used for carrying file I/O and Sleepycat DB over these connections are described in Chapter 2. Separate connections are used to carry the two RPC protocols.

The connections between the Client Daemon and the Data Servers are all NFS v3 [CPS95] running over UDP. The connections between the Client Daemon and the Metadata Servers are TCP streams carrying the same Sleepycat DB/RPC protocol as above.

The Client Daemon is written entirely in Erlang [AVWkW96]. Since Erlang does not currently have SMP support, this is the reason one may wish to run several Client Daemons on a single multi-processor Access Servers. We do not know what the ideal number of Client Daemons should be run on an Access Server. The optimal number of will depend on several factors, including which application(s) the Access Server runs. Future development in the Erlang language may solve this problem and enable a single Client Daemon to take advantage of multiple processors.

A Client Daemon also runs on the Command Server. The Command Server also runs the Erlang shell which controls and monitors the Client Daemons running on the Access Servers. It uses Erlang's native inter-node message passing mechanism to remotely execute commands and pass data from the Access Servers to the Command Server. The API for this interaction is detailed in Chapter 6.

1.4.3 Data Server Daemon

The Data Server Daemon is the NFS implementation built into the kernel of the Data Server. We require no special modifications or accesses to this process.

1.4.4 Metadata Server Daemon

The Metadata Server Daemon is a piece of software written by Sendmail Engineers to receive, process, and reply to DB/RPC requests from the network with a high level of concurrency and low latency. We expect the Metadata Server Daemon to be incorporated into Sleepycat's DB product in future releases, and maintained and developed by them.

1.4.5 Command Server Shell

A Client Daemon also runs on the Command Server to process administrative requests against the Data and Metadata Servers. Administrative utilities that affect the data acted upon by applications on the Access Servers can be run here in an automated or manual fashion. A special process, called the Command Server Shell, or `csterm`, also runs on the Command Server. It receives data from and provides state updates to the Client Daemons as they run on each Access Server.

Chapter 2

Client Protocol

This chapter describes the protocols spoken between the DI/ODE Client Library, which is linked with the applications, and the DI/ODE Client Daemon While these protocols are intended to link two processes on a single host, there is nothing preventing them being used between different hosts.

2.1 File Client Protocol

2.1.1 Design Philosophy

The file protocol is designed to mimic as closely as possible the traditional UNIX file-related system calls. This is for several reasons:

- by putting most of the state on the DI/ODE Client Daemon side of the connection, we enable faithful emulation of UNIX semantics for shared files after a `fork()` and the preservation of descriptor information after an `execve()`.
- keeping state on the Client Daemon side also allows it to take advantage of its state knowledge during a migration;
- the UNIX kernel provides an architecture model with well known structure.

2.1.2 Synopsis

Table 2.1 gives a synopsis of the protocol. Though it will be defined in the ONC/RPC syntax, and the resulting description fed to code-generating tools, the RPC syntax is not the most readable format. (The full RPC definition is given in Appendix A.)

The argument and return fields may be of the following types:

void The call takes no arguments.

mntindex An unsigned integer representing a mount point. Zero represents the process's current working directory.

pathblob A structure containing a *mntindex* and a string specifying a file or directory path name relative to the `mntindex`'s location.

mountentry A structure containing the absolute path of a DI/ODE mount point and that mount point's *mntindex*.

Procedure	Arguments	Results
`open`	*pathblob, flags, mode, fid*	*status*
`close`	*fid*	*status*
`read`	*fid, nbytes*	*status, data*
`write`	*fid, data*	*status, nbytes*
`pread`	*fid, nbytes, offset*	*status, data*
`pwrite`	*fid, data, offset*	*status, nbytes*
`seek`	*fid, offset, whence*	*status, offset*
`delete`	*pathblob*	*status*
`rename`	*pathblob, pathblob*	*status*
`link`	*pathblob, pathblob*	*status*
`mkdir`	*pathblob, mode*	*status*
`rmdir`	*pathblob*	*status*
`setattr`	*pathblob, sattr*	*status, fattr*
`stat`	*pathblob*	*status, fattr*
`fstat`	*fid*	*status, fattr*
`fsetattr`	*fid, sattr*	*status, fattr*
`fstatfs`	*fid*	*status, fsstats*
`statfs`	*pathblob*	*status, fsstats*
`fdirlist`	*fid*	*status, offset, direntries...*
`flock`	*fid, flockop*	*status*
`fiddump`	*void*	*status, procflags, fids...*
`setsessid`	*sessid, opaque*	*status, sessid, oldopaque*
`forksess`	*sessid*	*status*
`listmounts`	*void*	*status, mountentry...*
`resolv`	*pathblob*	*status, pathblob*
`dup2`	*fid, fid*	*status*
`chdir`	*pathblob*	*status, mntindex*
`fchdir`	*fid*	*status, mntindex*
`pathconf`	*pathblob, pathconfop*	*status, value*
`fpathconf`	*fid, pathconfop*	*status, value*
`fsync`	*fid, offset, nbytes*	*status*
`getflags`	*fid, flagop*	*status, flags*
`setflags`	*fid, flagop, flags*	*status*
`setuid`	*uid*	*status*
`setgid`	*gid*	*status*
`setgroups`	*gid...*	*status*

Table 2.1: DI/ODE File Protocol

flags An unsigned 32-bit integer holding flags corresponding to those of the `open()` and `fcntl()` system calls.

mode An unsigned 32-bit integer holding the permission mode bits, as in the `chmod()` system call, for a newly-created file or directory.

fid An unsigned 32-bit integer holding the descriptor number of an open file or directory, presumably matching the application-level descriptor.

nbytes An unsigned 32-bit integer holding the size of a read request or returning the number of bytes written in a write request.

data A block of up to 4GB ($2^{32} - 1$ bytes) holding data to be written, or data returned from a read.

offset A signed 64-bit integer holding an offset within a file.

whence An enumeration describing whether the seek is relative the the beginning or end of the file, or the current offset.

sattr A structure holding the settable attributes of a file or directory, comprising:

- *whichset* A bitmap indicating which of the following fields are being set in this call.
- *mode* As above.
- *uid* An unsigned 32-bit integer, holding the user ID.
- *gid* An unsigned 32-bit integer, holding the group ID.
- *size* An unsigned 64-bit integer, holding the file size.
- *atime* The time of last access.
- *mtime* The time of last file modification.

sessid An structure that must be unique and deterministic for a given Client Library instance. This is opaque to the Client Daemon and is currently expected to consist of:

- *host* A string giving a hostname.
- *pid* An unsigned 32-bit integer giving a unique process ID.

This structure is used in several protocol operations in conjunction with a `fork()` or `execve()` by the application.

opaque and *oldopaque* Data to be preserved in the Client Daemon across `execve()` calls in the client. The `setsessid` call returns the data set by the previous call. Currently, this is used to pass to the new image of the Client Library the number of the file descriptor of the previous image's connection to the Client Daemon.

procflags Random bits that the Client Daemon knows about a process that the process won't know when it starts up. Right now that is a single bit, `DIODE_FIDDUMP_REMOTE_CWD`, indicating whether the process's current working directory is under a DI/ODE mount point. This bits are copied to the new session created by a `forksess` call.

status A signed 32-bit return code. If the value is nonzero, then the protocol operation may not return any other values.

fattr A structure holding file or directory attributes, comprising:

- *type* An enumeration, saying whether this is a file or directory.

mode The permissions of the file or directory, as above.

nlink The number of links to this file or directory. At this time we are not sure whether the link count for directories will follow the normal UNIX semantics of being equal to two more than the number of subdirectories.

uid User ID, as above.

gid Group ID, as above.

atime Last access time, as above.

mtime Last modification time, as above.

ctime Time of last modification of file attributes.

rdev1 The first word of file type specific info.

rdev2 The second word of file type specific info.

dev The file system ID.

ino The file inode number.

size File size, as above.

used The actual number of bytes used by the file.

blksize The suggested unit of transfer, in bytes.

The last five members are unsigned 64-bit integers. All other members are unsigned 32-bit integers.

fsstats File system statistics, comprising:

tbytes Total size, in bytes.

fbytes Free space, in bytes.

abytes Free space available to current user, in bytes.

tfiles Total number of nodes on filesystem.

ffiles Free nodes available.

afiles Available nodes.

These are all unsigned 64-bit integers.

direntry A structure describing a directory entry, comprising:

ino A 64-bit number, corresponding to the inode.

name A string giving the directory entry name.

flockop An enumeration specifying whether this is a lock, unlock, or test lock request.

pathconfop An enumeration specifying which path configuration setting is being checked.

flagop An enumeration specifying whether the file status flags (`O_RDWR`, etc) or file descriptor (close-on-exec) flag is being retrieved or set.

uid An unsigned 32-bit integer, holding the user ID.

gid An unsigned 32-bit integer, holding the group ID.

This protocol differs from the UNIX system call API in a couple ways. In particular, the client specifies the file descriptor in the `open` call, and there is no `dup` call, only `dup2`. This follows the decision, explained in the next chapter, to allow the Client Library to control descriptor assignment. Therefore, it is up to the client to specify unique descriptors in the protocol. Attempts to `open` an already open descriptor should generate a `DIODE_ERR_BADF` error.

Unlike the UNIX `readdir()` or `getdents()` functions, the `fdirlist` call returns the names of *all* the items in the open directory and not an incremental portion of them. This simplifies the Client Daemon's handling of directories that are stored across more than one backend Data Server, whether for redundancy or stripping. The `fdirlist` call also returns the current file offset, as set by the `seek` call. This lets the Client Library support calls to `seekdir()` that occur before any directory entries are read.

Also, file and directory paths are expressed in the Client Protocol by combining a mount point, indicated by index, and a path relative to that point. Paths relative to the process's current working directory are expressed by using a mount index of zero.

By default, it is an error to pass a relative path in a Client Protocol call and the `DIODE_ERR_NO_CURRENT_DIR` status should be returned. However, sending a `fchdir` request or a `chdir` request with a non-empty path sets the directory from which relative paths should be resolved and sets the `DIODE_FIDDUMP_REMOTE_CWD` bit in the *procflags* returned by the `fiddump` call. A `chdir` request with an empty path and any *mntindex* clears that bit and makes it an error again to pass a relative path.

Due to limitations in the current Erlang RPC implementation, in this iteration of the DI/ODE Client Protocol, we will use the null authentication flavor, `AUTH_NONE`, and pass information regarding user and group IDs to the Client Daemon via separate RPC procedures. The Client Daemon is expected to use the information so obtained to generate authentication credentials for its connections to the Data Servers.

2.1.3 Connection Setup

New RPC connections are opened in two cases: during a `fork()`, and during initialization of the Client Library after an `exec()`. In the former case, the first call made on the connection must be `forksess`, to form a new session and duplicate the file descriptor state. That must be followed by a `setsessid` call to 'name' the session and save the connection specific opaque data in the Client Daemon.

In the case of an `exec()`, the first call made must be `setsessid`. This either reestablishes the association of the process's RPC connection to the previous process image's session, or creates and names a new session. If there was a previous session then the next call on this connection should be `fiddump` to finish the transfer of previous state from Client Daemon to Client Library. Finally, the Client Library must initialize its table of mount points and mount indexes by making the `listmounts` call, and set the correct authorization information in the Client Daemon by making the `setuid`, `setgid`, and `setgroups` calls.

2.1.4 Symbolic Links

This revision of the Client Protocol does *not* support symbolic links. In particular, it lacks that RPC procedures to handle `readlink()` and `symlink()`, and the 'ell' calls (`lstat()`, `lchown()`, `_lxstat()`, and `lutimes()`). Note that this is a separate (and simpler) issue from that of symbolic links in the local filesystem that point 'into' diode-space. Those are an issue for the Client Library only. Limitations on them are considered in Section 3.1.3.

2.1.5 Status Codes

The following status codes will be used for the *status* field in the protocol:

`DIODE_OK` The operation was successful. If the status is *not* `DIODE_OK`, then no other arguments will be returned.

`DIODE_ERR_ROOT` One or more of the *pathblob* arguments specified in the call resolved to a location outside of the DI/ODE mount point that it started under. The Client Library should call `resolv` on each path passed in the call and restart its the processing. If this status is returned then at least one of the paths used in the call must generate a `DIODE_OK` status when passed to the `resolv` call.

`DIODE_ERR_NEW_SESSID` A `setsessid` call was made that specified an unknown *sessid* value. A new session has been created. This status may not be returned by any other call.

`DIODE_ERR_LOOP` Too many symlinks were encountered while processing a path so the Client Daemon gave up. In this revision of the Client Protocol this status will be returned when *any* symlinks are encountered by the Client Daemon, i.e., the "number considered to be too many" will be *one*.

`DIODE_ERR_STILL_REMOTE` A path was passed to the `resolv` call that when processed did not leave the mount point it started under. This status may not be returned by any other call.

`DIODE_ERR_BADF` An operation was requested on an invalid fid. The Client Library and Client Daemon are out of sync.

`DIODE_ERR_NO_CURRENT_DIR` A relative path was passed, but this session does not currently have set a current working directory.

`DIODE_ERR_AGAIN` An attempt to lock a file failed because the file was already locked. This status may only be returned by the `flock` call.

`DIODE_ERR_PERM` This error code and the following are lifted directly from the NFSv3 specification ([CPS95]) and should be interpreted as described there.

`DIODE_ERR_NOENT`

`DIODE_ERR_IO`

`DIODE_ERR_NXIO`

`DIODE_ERR_ACCES`

`DIODE_ERR_EXIST`

`DIODE_ERR_XDEV`

`DIODE_ERR_NODEV`

`DIODE_ERR_NOTDIR`

`DIODE_ERR_ISDIR`

`DIODE_ERR_INVAL`

`DIODE_ERR_FBIG`

`DIODE_ERR_NOSPC`

`DIODE_ERR_ROFS`

`DIODE_ERR_MLINK`

`DIODE_ERR_NAMETOOLONG`

`DIODE_ERR_NOTEMPTY`

`DIODE_ERR_DQUOT`

`DIODE_ERR_STALE`

`DIODE_ERR_REMOTE`

If a status is returned that is not in the above list then the behavior of the Client Library is undefined. The initial implementation will panic and call `abort()`.

2.2 Sleepycat RPC Protocol

Starting with version 3.1.14, the Sleepycat DB package has included its own RPC protocol for using a remote database server [Sof00]. We have extended the original protocol to support the direct locking calls. Like our file protocol, this protocol is designed so that the client-side does the minimal amount of work to package the function call and ship it across the wire. No significant semantic changes are done on the client-side. The protocol is summarized in Table 2.2.

The argument and return fields may be of the following types:

envid, dbid, txnid, parent-txn, and *curid* The client-side IDs for the actual server-side handles, given as 32-bit unsigned integers.

key-args and *data-args* "Database Thangs": `DBT` structures used to pass key and data values to the server. Includes flags and partial request fields.

keydata and *datadata* Variable length byte-strings, returned from the server for some, but not all *get* and *put* messages. Whether these are returned depends on the *flags* passed with that call and the type of the underlying database.

obj-args A variable length byte-string, currently passed as a `DBT` with the extra fields ignored by the server, that describes the object being locked.

lock A `DB_LOCK` structure, passed as an opaque byte string (`DB_LOCK` structures are not for interpretation outside of the Sleepycat library itself). Note the exact size of the `DB_LOCK` structure structure depends on the size of the `size_t` type and is therefore platform dependent.

mode An unsigned 32-bit integer holding the permission mode bits, as in the `chmod()` system call, for a newly-created database.

lockmode The type of lock being requested. The meaning of a given lock mode is implied by the lock conflict matrix currently in use by the environment.

lockmodes The number of different lock types that are supported by the new lock conflict matrix.

conflicts A lock conflict matrix, passed as a byte string whose length is the square of the *lockmodes* value.

req-array A variable length array of lock requests. There are four types of request:

`DB_LOCK_GET` Get a lock, as specified by included *mode* and *obj* items.

Procedure	Arguments	Results
`env_cachesize`	*envid, gbytes, bytes, ncache*	*status*
`env_close`	*envid, flags*	*status*
`env_create`	*timeout*	*status, envid*
`set_lk_conflict`	*envid, conflicts, lockmodes*	*status*
`set_lk_detect`	*envid, detect*	*status*
`set_lk_max`	*envid, max*	*status*
`env_open`	*envid, home, flags, lockmode*	*status*
`env_remove`	*envid, home, flags*	*status*
`txn_abort`	*txnid*	*status*
`txn_begin`	*envid, parent-txn, flags*	*status, txnid, real-txnid*
`txn_checkpoint`	*envid, kbyte, min, flags*	*status*
`txn_commit`	*txnid, flags*	*status*
`txn_prepare`	*txnid*	*status*
`db_bt_maxkey`	*dbid, maxkey*	*status*
`db_bt_minkey`	*dbid, minkey*	*status*
`db_close`	*dbid, flags*	*status*
`db_create`	*envid, flags*	*status, dbid*
`db_del`	*dbid, txnid, key-args, flags*	*status*
`db_flags`	*dbid, flags*	*status*
`db_get`	*dbid, txnid, key-args, data-args, flags*	*status, keydata, datadata*
`db_h_ffactor`	*dbid, ffactor*	*status*
`db_h_nelem`	*dbid, nelem*	*status*
`db_key_range`	*dbid, txnid, key-args, flags*	*status, less, equal, greater*
`db_lorder`	*dbid, lorder*	*status*
`db_open`	*dbid, name, subdb, type, flags, mode*	*status, type, dbflags*
`db_pagesize`	*dbid, pagesize*	*status*
`db_put`	*dbid, txnid, key-args, data-args, flags*	*status, keydata*
`db_re_delim`	*dbid, delim*	*status*
`db_re_len`	*dbid, len*	*status*
`db_re_pad`	*dbid, pad*	*status*
`db_remove`	*dbid, name, subdb, flags*	*status*
`db_rename`	*dbid, name, subdb, newname, flags*	*status*
`db_stat`	*dbid, flags*	*status, entlist*
`db_swapped`	*dbid*	*status*
`db_sync`	*dbid, flags*	*status*
`db_cursor`	*dbid, txnid, flags*	*status, curid*
`db_join`	*dbid, curlist, flags*	*status, curid*
`dbc_close`	*curid*	*status*
`dbc_count`	*curid, flags*	*status, dupcount*
`dbc_del`	*curid, flags*	*status*
`dbc_dup`	*curid, flags*	*status, curid*
`dbc_get`	*curid, key-args, data-args, flags*	*status, keydata, datadata*
`dbc_put`	*curid, key-args, data-args, flags*	*status, keydata*
`lock_detect`	*envid, flags, atype*	*status, aborted*
`lock_get`	*envid, locker, flags, obj-args, lockmode*	*status, lock*
`lock_id`	*envid*	*status, id*
`lock_put`	*envid, lock*	*status*
`lock_stat`	*envid*	*status, entlist*
`lock_vec`	*envid, locker, flags, req-array*	*status, locklist*

Table 2.2: Sleepycat RPC Protocol

`DB_LOCK_PUT` Release a lock, specified by an included *lock* item.

`DB_LOCK_PUT_OBJ` Release all locks on a specified object, passed via an `obj` item.

`DB_LOCK_PUT_ALL` Release all locks held by this locker.

locklist A list of *lock* items obtained by `DB_LOCK_GET` requests.

entlist A statistics array, returned as a list of unsigned 32-bit integers.

real-txnid The internal server-side ID of the transaction. This can be used to associate lock requests with particular transactions. (This capability is not used by our current expected set of applications, and is not fully supported by the Sleepycat client-side library. Further work involving this is under evaluation by Sleepycat.)

id A new unique locker ID.

status A status code, passed as a signed 32-bit integer. Zero is success, while negative values have special meaning in the Sleepycat DB context. Positive values are errno values directly from the server. Note that this is a platform specific encoding.

curslist A list of cursor ID numbers.

home, name, subdb, and *newname* Character string specifying, respectively, the last component of the home directory of an environment, the name of a database file, the name of a sub-database, and the new name for either a database file or a contained sub-database.

dbflags The internal flags associated with a newly opened database. (This is provided strictly for use by the TCL extension.)

All the other items are signed or unsigned 32-bit integers, matching exactly to the argument of the same name in the original Sleepycat call.

Chapter 3

Application-Side Libraries

3.1 File Client Library

3.1.1 Interface Styles

The Client Library is a collection of functions which make it possible for applications to perform I/O both to DI/ODE Data Servers as well as local file systems with minimal source code modification. It provides functions which mimic the standard UNIX file I/O system call functionality of `open()`, `read()`, `write()`, `lseek()`, `close()`, et al.

Ideally, an application need only be linked with the Client Library in order to access DI/ODE Data Servers. Doing so requires some platform specific code in the Client Library in order to handle a simple problem: if the Client Library contains a function called `open()`, how does one then call the underlying `open()` for local file system operations? To complicate the matter, on many platforms the Pthread library contains wrappers around many I/O system calls that implement cancellation points or otherwise make them suitable for use by multi-threaded applications. These wrappers would be inaccessible if the Client Library were to naïvely invoke system calls via `syscall()`.

Previous versions of the Client Library have attempted to solve this problem by taking advantage of a detail in the construction of of `libc` and `libpthread`. On all platforms used to date (Solaris, FreeBSD, and Linux), the functions in `libc`, such as `open()`, are actually implemented by `_open()`. `Libc` then contains a "weak symbol" which tells the linker to resolve `open()` calls to `_open()` if `open()` isn't defined "strongly" elsewhere (such as in `libpthread`. The Client Library would take advantage of this construction just as `libpthread` does, calling the internal symbol when a local operation is needed.

Unfortunately, this overriding of symbols removes the Pthread library from the loop, thereby breaking certain expected semantics. We have therefore implemented a style of symbol override that relies on the dynamic linker to provide the necessary information for completing local calls. At least on platforms that use the ELF binary format, the `dlsym()` function, when passed the constant `RTLD_NEXT` for its `handle` argument, will return the 'next' definition of the specified symbol. As part of its initialization, the Client Library calls `dlsym()` for each function that it needs to extend, caching the pointer to the underlying implementation that is returned.

Using this, we may even be able to provide wrappers for the internal symbols, such as `_open()`. Doing so would eliminate problems with using the STDIO component of `libc`—on all the platforms that we have tested on, STDIO uses the internal symbols (e.g., `_open()`) rather than their corresponding weak symbol names (e.g., `open`). However, this may prove to be unworkable if the Pthread library uses those same symbols when it intercepts system calls. For example, the implementation of `fopen()` in found in `libc` and the implementation of `open()` found in `libpthread.so` may both directly invoke the function/symbol `_open`. If so, then the Client Library implementation of `_open`

would need to meet the union of the requirements of both those calls. Specifically, it would be need to be both async-cancel-safe and async-signal-safe (see also the Solaris 2 `attributes(5)` manpage for details of those terms).

If the interception of internal symbols proves to be impractical, then the Client Library will also need to provide implementations for all functions in `libc` that use the internal symbols. In particular, the Client Library may need to include implementations of the POSIX directory functions (`opendir()`, `readdir()`, etc) and the ISO C STDIO functions (`fopen()`, `printf()`, etc). If this happens, we plan on using the STDIO implementation found in the SFIO library released by AT&T.

One 'gotcha' that we will have to check for is whether this method creates a restriction on the link order for applications. In particular, it may turn out that the Client Library must appear before `libpthread` in the link command.

Implementations of previous versions of the Client Library have been successfully linked again the following applications and thereby accessed files stored on DI/ODE Data Servers. Unless specified, the only modifications required have been to include the Client Library and SFIO library in the final executable linking phases.

GNU `tar` Version 1.13. No source code modifications are required. Limits on file descriptor duplicating renders use of the `-z` flag unusable when the source file is stored on a Data Server. It is expected that this limitation does not apply to the version of the Client Library and Client Protocol described in this document.

GNU fileutils Version 4.0. No source code modifications are required. Since the Client Protocol does not implement NFS symbolic link and special file operations, `ln -s`, `mkfifo`, and `mknod` do not currently work.

sendmail Version 8.11.1. No application source code modifications appear required, though its feature set has not been thoroughly tested yet.

qmail Version 1.03. Due to its dependence on file descriptor copying, a couple of hacks are required to copy data from Data Servers to local disk prior to certain operations. The patch affects roughly 100 lines of code. Qmail's operation has only been lightly tested: testing so far has been at a proof-of-concept level. Again, it is expected that this limitation does not apply to the version of the Client Library and Client Protocol described in this document.

3.1.2 Supported System Calls

The Client Library provides most of the semantics provided by the UNIX file I/O system calls.

When used on files in local file systems, the Client Library uses the underlying operating system's function, as provided by either `libc` or `libpthread` and accessed via `dlsym()`. When the operation is on a file stored on a Data Server, the operation is converted to a Command Protocol operation and invoked in the Client Daemon via an ONC RPC call.

The list of UNIX file I/O system calls and `libc` functions that are always intercepted by the Client Library can be found in Table 3.1.

Additional calls that may be intercepted on some platforms can be found in Table 3.2. Specifically, the following additional calls will be intercepted under Solaris: `fstatvfs()`, `getdents()`, `statvfs()`, `_fxstat()`, `_lxstat()`, and `_xstat()`. The functions `_fxstat()`, `_lxstat()`, and `_xstat()` provide binary support for previous versions of `struct stat`. We expect to only provide support for the current version of `struct stat` for files under DI/ODE mount points.

Some platforms define separate 'large file' functions that use 64-bit types. While we don't currently feel the need to support the entire large file interface, we have avoided creating any obstacles to doing so beyond the extra effort needed. The Client Protocol itself matches NFSv3 by using 64-bit types for many fields. There are three 'large file' functions that we may implement regardless:

`getdents64()`, `lstat64()`, and `stat64()`. Commands like `ls` use these to safely walk and examine file trees.

The use of other platform specific file system related calls on DI/ODE file space is not supported. For example, we do not support use of the `ustat()` or `realpath()` systems calls on DI/ODE 'devices' or filenames.

We do not currently plan to support the SPARC V9 (LP64) environment under Solaris.

We would prefer to not implement the directory handling functions, `closedir()`, `opendir()`, `readdir()`, `readdir_r()`, `rewinddir()`, `seekdir()`, and `telldir()`, and instead continue to use the versions in `libc`. As long as our implementations of the `getdents()` and `getdirentries()` functions are complete, this should Just Work. If problems arise during development then we may simply fall back to intercepting the directory handling functions in the Client Library.

While the Client Protocol at this time does not directly support symbolic links beneath DI/ODE mount points, the Client Library still intercepts all symlink related calls. How it handles each one is described below.

We do not currently plan to support applications that expect to be able to access files beneath DI/ODE mount points after calling `chroot()` or `fchroot()`

3.1.3 Path Canonicalization

The Client Library must canonicalize all paths in order to determine whether they specify objects in a local file system or on a remote DI/ODE Data Server. The canonicalization process has to handle three types of paths: absolute paths, paths relative to directories in the local file systems, and paths relative to directories beneath DI/ODE mount points.

This DI/ODE specifically *does not* support either symbolic links within the diode-space in any form, or symbolic links outside of diode-space that point into it. Supporting such accesses would require either that the Client Library perform full kernel-like path canonicalization, including calling `lstat()` on each component, or that the kernel provide better hooks for user-level path processing. The `realpath()` function, as implemented in both BSD 4.4 and Solaris, while tantalizing, is *not* sufficient. Attempts to resolve through non-existent path components, such as anything beneath a DI/ODE mount point in the local file system, generate an error without returning any information about what had so far been resolved.

On the other hand, we can support accesses through symlinks in the local file system that resolve to other locations in the local file system without going beneath a DI/ODE mount point. All it takes is for use to use the uncanonicalized path whenever we actually perform an operation on a local object. The canonicalization process tells us whether we need to redirect the operation over the Client Protocol and, if so, what path to uses there. It does not change the path used for local operations.

To canonicalize an absolute path, the Client Library considers each component in turn, comparing the path so far against the list of DI/ODE mount points listed in `/etc/diode/paths.conf`. A component of "`.`" is dropped, while a component of "`..`" is removed along with the previous component. If a "`..`" component is encountered when there is no previous component, then it is simply dropped. As soon as the Client Library finds a match, the rest of the path is passed to the Client Daemon without further changes. If the entire path is canonicalized without matching a DI/ODE mount point, then the path must be local and the original path is passed to the underlying system call.

If the process's current working directory is not beneath a DI/ODE mount point and a relative path is to be processed by the Client Library, then the Client Library will append it to the path to the process's current working directory, as obtained via the `getcwd()` function, and resolve it as an absolute path. The result of the `getcwd()` call may be cached between calls to `chdir()` or `fchdir()`. There may be opportunities for optimizing this process further.

File API Prototypes		Client Protocol call
int	access(const char *path, int mode)	stat
int	chdir(const char *path)	chdir
int	chmod(const char *path, mode_t mode)	setattr
int	chown(const char *path, uid_t owner, gid_t group)	setattr
int	close(int fd)	close
int	creat(const char *path, mode_t mode)	open
int	dup(int fd)	dup2
int	dup2(int oldd, int newd)	dup2
int	‡execve(const char *path, char *const argv[], char *const envp[])	-
int	fchdir(int fd)	fchdir
int	fchmod(int fd, mode_t mode)	fsetattr
int	fchown(int fd, uid_t owner, gid_t group)	fsetattr
int	fcntl(int fd, int cmd, ...)	getflags, setflags
int	flock(int fd, int op)	flock
pid_t	fork(void)	forksess
long	fpathconf(int fd, int name)	fpathconf
int	fstat(int fd, struct stat *sb)	fstat
int	fsync(int fd)	fsync
int	ftruncate(int fd, off_t length)	fsetattr
int	†lchown(const char *path, uid_t owner, gid_t group)	†setattr
int	link(const char *old, const char *new)	link
off_t	lseek(int fd, off_t offset, int whence)	seek
int	†lstat(const char *path, struct stat *sb)	†stat
int	mkdir(const char *path, mode_t mode)	mkdir
int	open(const char *path, int flags, ...)	open
long	pathconf(const char *path, int name)	pathconf
ssize_t	pread(int fd, void *buf, size_t nbytes, off_t offset)	pread
ssize_t	pwrite(int fd, const void *buf, size_t nbytes, off_t offset)	pwrite
ssize_t	read(int fd, void * buf, size_t bytes)	read
int	readlink(const char *path, char *buf, size_t bufsize)	returns EINVAL
ssize_t	readv(int fd, const struct iovec *iov, int iovcnt)	read
int	rename(const char *old, const char *new)	rename
int	rmdir(const char *path)	rmdir
int	setegid(gid_t gid)	setgid
int	seteuid(uid_t uid)	setuid
int	setgid(gid_t gid)	setgid
int	‡setgroups(int ngroups, const gid_t *grouplist)	setgroups
int	setuid(uid_t uid)	setuid
int	‡setregid(gid_t rgid, gid_t egid)	setgid
int	‡setreuid(uid_t ruid, uid_t euid)	setuid
int	stat(const char *path, struct stat *sb)	stat
int	symlink(const char *name1, const char *name2)	returns ENOSYS
int	truncate(const char *path, off_t length)	setattr
mode_t	umask(mode_t numask)	-
int	unlink(const char *path)	unlink
int	utime(const char *path, const struct utimbuf *timep)	setattr
int	utimes(const char *path, const struct timeval *timep)	setattr
ssize_t	write(int fd, const void *buf, size_t bytes)	write
ssize_t	writev(int fd, const struct iovec *iov, int iovcnt)	write

Table 3.1: DI/ODE File API and Protocol Mapping—OS Independent Calls
† Symbolic link semantics are disabled for these calls
‡ The signatures for these functions vary from platform to platform. The table shows the types appropriate for Solaris 7.

	File API Prototypes	Client Protocol call
int	fstatvfs(int fd, struct statvfs *buf)	fstatfs
int	‡getdents(int fd, struct dirent *buf, size_t nbytes)	fdirlist
int	statvfs(const char *path, struct statvfs *buf)	statfs
int	_fxstat(int version, int fd, struct stat *sb)	fstat
int	†_lxstat(int version, const char *path, struct stat *sb)	†stat
int	_xstat(int version, const char *path, struct stat *sb)	stat
int	fstatfs(int fd, struct statfs *buf)	fstatfs
int	futimes(int fd, const struct timeval *timep)	fsetattr
int	getdents(int fd, char *buf, int nbytes)	fdirlist
int	getdirentries(int fd, char *buf, int nbytes, long *basep)	fdirlist
int	†lutimes(const char *path, const struct timeval *timep)	†setattr
int	statfs(const char *path, struct statfs *buf)	statfs
ssize_t	preadv(int fd, const struct iovec *iov, int iovcnt, off_t offset)	pread
ssize_t	pwritev(int fd, const struct iovec *iov, int iovcnt, off_t offset)	pwrite

Table 3.2: DI/ODE File API and Protocol Mapping—OS Dependent Calls
† Symbolic link semantics are disabled for these calls ‡ The signatures for these functions vary from platform to platform. The table shows the types appropriate for Solaris 7.

If the process's current working directory is beneath a DI/ODE mount point and a relative path is to be processed by the Client Library, then the Client Library will simply pass the path through to the Client Daemon unchanged.

If a Client Protocol call returns `DIODE_ERR_ROOT` then the Client Library will use the `resolv` call to obtain an equivalent partially canonicalized path from the Client Daemon. This path will have as a prefix the directory above the involved DI/ODE mount point, followed by whatever remains of the orignal path. This path will be further canonicalized as described above and the call retried.

For the intercepted `rename()` and `link()` functions—the two functions that take two paths as arguments—the Client Library will only invoke the Client Protocol call of the same name if both paths appear to be beneath DI/ODE mount points. If the Client Protocol call returns `DIODE_ERR_ROOT` then the Client Library will attempt to resolve both paths via the `resolv` call and then restart the processing. If only one path is beneath a DI/ODE mount point then the Client Library will first pass that path to the `resolv` call. If that doesn't result in a new path above the DI/ODE mount point, then the Client Library will return the system error `EXDEV`. If both paths are local, then the Client Library will call the underlying system call.

Note that as described above, the Client Library does 'lazy' canonicalization: the Client Library stops processing a path as soon as it goes beneath a DI/ODE mount, leaving all canonicalization of the rest of the path to the Client Daemon. Similarly, it only uses the `resolv` call if it received a `DIODE_ERR_ROOT` error on a previous call or if it is logically necessary (as in the case of a `rename()` with only one local path).

3.1.4 Client Library Internal State, File API

The Client Library keeps the following state:

- list of DI/ODE mount points
- session ID (hostname and pid, cached)
- map of Client Daemon number to socket address and RPC `CLIENT` handle

- the process's umask
- whether the process's current working directory is beneath a DI/ODE mount point, and if so, which Client Daemon is handling relative paths
- if the current working directory is not beneath a DI/ODE mount point, then it may cache the absolute path to the current working directory
- map of emulated fds to Client Daemon numbers.

In addition, the first time `getdents()` or `getdirentries()` is called on an emulated directory fd, the Client Library will obtain the complete list of directory entries from the Client Daemon using the `fdirlist` call. This info will be stored by the Client Library and doled out with successive calls to `getdents()` and `getdirentries()`.

Configuration Files

The Client Library uses two configuration files located in the `/etc/diode` directory: `filesock.conf` and `dbsock.conf`. The `filesock.conf` file lists the sockets on which Client Daemons will be accepting the Client Protocol. The `dbsock.conf` file lists the sockets on which Client Daemons will be accepting Sleepycat DB/RPC calls as described in Section 3.2.

fork

A fork is handled as follows:

```
pid_t
fork()
{
        for each DI/ODE daemon connection {
                make a new, non-multithreaded RPC connection
                send the forksess() request
                destroy the CLIENT handle without closing the fd
        }
        if ((pid = syscall(FORK)) == 0) {
                /* child */
                for each DI/ODE daemon connection {
                        close parent's connection
                        create multithreaded RPC connection on child fd
                        send the setsessid() request
                }
                return 0
        }
        /* parent */
        close all child connection fd's
        return pid
}
```

exec (initialization)

After an exec, the Client Library needs to initialize itself. This will either be done via an `ELF .init` section which is automatically executed during process startup or, if that is considered impractical, when the first call is made to an intercepted function. In either case, the configuration files are read to find the sockets to which the Client Library will connection. A connection is made to each

daemon, and `setsessid` is called on each one. For each connection where the `setsessid` returned success, the returned "opaque" data is parsed to extract the fd of the socket that the pre-exec image was using to connect to that Client Daemon. That fd is closed. Furthermore, it will also send the `fiddump` request to obtain from that Client Daemon the list of fds that were being emulated by that Client Daemon. That also tells the Client Library whether the process's current working directory is beneath a DI/ODE mount point. Finally, the Client Library will invoke the underlying `geteuid()`, `getegid()`, and `getgroups()` system calls and pass the obtained information to the Client Daemon via the `setuid`, `setgid`, and `setgroups` calls.

3.1.5 Support For Multiple Client Daemons

As our current implementations of the Client Daemon do not use kernel level threads, they cannot take advantage of multiple processors. To work around this, we will run multiple instances of the Client Daemon on each Access Server. The Client Library will therefore spread its file accesses across the Client Daemons listed in the `/etc/diode/filesock.conf` file by hashing all or part of the path or paths specified in the calls. Once a file has been opened and a fid obtained from a Client Daemon, all calls involving that fid must be sent to that same Client Daemon.

An additional complication occurs when a process attempts to use `chdir()` or `fchdir()` to change its working directory to beneath a DI/ODE mount point. The `chdir` or `fchdir` Client Protocol call can only be sent to a single Client Daemon. In the case of `fchdir`, it must obviously go to the Client Daemon on which the specified fid was opened, while in the case of `chdir`, sending it to more than one makes it non-atomic and may cause problems if the specified path is being renamed at the same time. As long as a process has a current working directory beneath a DI/ODE mount point, all relative paths must be sent to the Client Daemon which received the Client Daemon call as only it can properly resolve such paths.

3.1.6 Authentication Issues

In order to guarantee that the Client Daemon's per-process state includes the correct user ID and group IDs, the Client Library intercepts seven system calls, `setegid()`, `seteuid()`, `setgid()`, `setgroups()`, `setregid()`, `setreuid()`, and `setuid()`, and for each one calls the underlying system call. If it succeeds, the Client Library will call the corresponding 'get' call, one of `geteuid()`, `getegid()`, and `getgroups()`, and pass the result to the Client Daemon via the Client Protocol.

3.1.7 Internal Locking

To maintain the consistency of the information stored in the Client Daemons, the Client Library will protect certain operations with mutexes.

`id_mutex` For all the functions that set user, group, or supplementary group IDs, the Client Library needs to send the appropriate request to all the Client Daemons as well as invoke the underlying system call. It therefore will lock the `id_mutex` before invoking the underlying call and release it once all the Client Daemons have been updated.

`chdir_mutex` When a process changes its current working directory, the Client Library may need to tell a Client Daemon to clear its previous information regarding that process's working directory, as well as invalidate any other cached information about its working directory. The Client Library will therefore long the `chdir_mutex` before changing its working directory and before using any cached information regarding the working directory.

3.1.8 Unsupported Calls and Semantics

To make migration feasible, we restrict links and renames to be within a single directory. Therefore, each inode has a unique and constant parent inode.

If a system call or `libc` function wasn't listed in Section 3.1.2, then it is not implemented by the Client Library. However, some calls warrant further discussion. Furthermore, some of the calls that the Client Library does implement have noteworthy limitations.

Blocking vs. Non-Blocking Behavior

All operations on remote file descriptors have blocking behavior, regardless of settings by `open()` or `fcntl()`. Since UNIX semantics demand that descriptors for local file system objects must be blocking, this does not cause hardship: applications aren't expecting non-blocking behavior, so the Client Library doesn't have to provide it. The one exception to this is `flock`: attempts to lock a file are always non-blocking.

`chdir` A quirk in `chdir()`'s implementation is that the `chdir()` system call is almost always made by the Client Library. If the path is considered remote, then the Client Library will pass the path of the involved DI/ODE mount point to the local system call.

If a process that has previously changed its current working directory to beneath a mount point changes it again to outside of any DI/ODE mount points, then the Client Library will send a `chdir` call with an empty path to the Client Daemon that handle the last `chdir` or `fchdir` call. This will clear the flag in the Client Daemon marking this process as being beneath a DI/ODE mount point.

`execve`, et al After `execve()` has been called on an executable that is also linked with the Client Library, the first time the process attempts to perform any I/O, it will first contact the Client Daemon for remote file descriptor information and reconstruct its remote file descriptor table. See also section 3.1.4.

During the handling of a `chdir()` or `fchdir()` call, more than one Client Daemon may believe that it knows the process's working directory. If an `execve()` took place during that period, the Client Library initialization code would receive conflicting data from the Client Daemons and would be forced to abort. The Client Library will therefore intercept calls to `execve()` and hold the `chdir_mutex` across the `execve()` attempt.

`fchdir` As with `chdir()`, changing to a remote directory will still generate a call to the system call `chdir()` to change the process's 'real' working directory to the place holder directory marking the DI/ODE mount point.

`fcntl` Very few `fcntl()` operations are supported by the Client Library. Fortunately, most are not required by the applications we use. Currently only `F_DUPFD`, `F_GETFL`, `F_GETFD`, `F_SETFL`, and `F_SETFD` are implemented. POSIX file locking via `fcntl()` is explicitly *not* supported. Applications are expected to use `flock()` instead.

`flock` Only the exclusive behavior, `LOCK_EX`, is implemented by the Client Library. Requests for a shared lock, `LOCK_SH`, are treated as if they were exclusive requests.

If the lock is requested without the non-blocking option `LOCK_NB`, then attempts are made to obtain the lock once per second until the lock is successfully obtained or there was an error during the lock attempt.

`lstat and lchown` Symbolic links are not supported by the Client Daemon. A call to `lstat()` on a path in a remote namespace is implemented exactly the same as `stat()`, the non-symbolic link-aware version. Similarly, a call to `lchown()` on a path in a remote namespace is treated the same as a call to `chown()`.

`symlink` Attempts to create symbolic links beneath a DI/ODE mount point always fail with the error `ENOSYS`. When practical, the Client Library may catch attempts to create symbolic links in local file systems that point into a DI/ODE name space and force them to fail. In the end though, this qualifies for the "this hurts, Doctor" exemption.

`readlink` Attempts to call `readlink()` on paths beneath a DI/ODE mount point will always return the error `EINVAL`.

`mmap` Due to the intimate tie between the OS and virtual memory management, `mmap()` is an extremely difficult system call to emulate in user space. Therefore, the Client Library does not do so.

For read-only, private mappings, the Client Library can `malloc()` a hunk of memory the size of the requested mapping, `read()` data from the file into that hunk, and pass the hunk's pointer back to the caller. The caller wouldn't know the difference between that memory hunk and a "real" `mmap()`'ed one. This emulation technique cannot work well if the requested region is many megabytes (or, heaven forbid, gigabytes).

A lightly-explored alternative is to use either `mprotect()` or, for Solaris, use the `/proc` file system to manipulate regions of protected memory. The Client Library would have to trap accesses to a protected memory region, determine what memory location caused the trap, unprotect the region, `read()` data from Data Server(s) into the region, and then resume the thread that triggered the trap.

This technique would require the Client Library to become almost as involved in memory management as the OS's VM system is. If the technique were viable, it might also be applied to writable memory maps, also. However, it is still very unclear whether such a technique can be implemented in a manner that is sufficiently stable and efficient for our purposes. Even if we were to attempt it, the implementation would be extremely non-portable.

`poll and select` Both `poll()` and `select()` are not supported for remote file descriptors. Since most UNIX applications don't expect non-blocking behavior from local file system descriptors, they don't bother to put such descriptors into `poll()` or `select()` descriptor sets. Therefore, the Client Library does not bother defining either of these functions.

`getdents64, lstat64, and stat64` In an attempt to ease the porting of large-file aware system utilities, the Client Library may intercept these calls. We do not expect to intercept any other 64-bit system or library calls at this time: our suite of applications have not yet demanded them. Implementation should be straightforward if they are needed.

`readv, writev` The initial implementations of these calls are expected to be fairly naïve, with the data being copied twice. While we will almost surely perform the obvious and trivial optimization when the `iov` count is one[1], further optimization will probably require some sort of hand-coded XDR marshaling/unmarshaling routine.

`syscall` While we theoretically could intercept `syscall()` in the Client Library, doing so would be *very* unportable and time consuming, as well as of dubious value—is not the intent of `syscall()` to go directly to the kernel? We therefore have no plans to do so.

`umask` Like NFS, the Client Protocol has no concept of umask. As such, the Client Library must intercept `umask()` and not only call the underlying system call, but also apply the given value to all future calls to `open()` and `mkdir()` that are sent via the Client Protocol. The Client Library will need to call the system `umask()` function during its initialization in order to obtain the correct initial value for the internal value applied to Client Protocol operations.

[1] This is not a joke. We have seen applications where the data is all written out via calls to `writev()` with a single `iov`.

3.1.9 Supported STDIO Calls

[PAG: assuming the wrapping of the internal symbols (`_open()`, etc) works, we should be able to use the system STDIO. If so, this section should be removed.] **[NPC: Someone besides me should decide this.]**

The SFIO library [FKV00] used in conjunction with the Client Library to provide services conforming to the STDIO API. Thus the STDIO functions used by a Client Library-linked application are limited to that provided by SFIO's STDIO compatibility library.

The only known compatibility problem with the combination of the Client Library and SFIO libraries is `funopen()`, which is present in the STDIO library found in FreeBSD's `libc`. An in-core data stream created by `funopen()` in this circumstance will fail. As a workaround, `sendmail` should be compiled without using the `bf_*` modules. Including `undefine('confSTDIO_TYPE')` in a `site.config.m4` file is sufficient to disable their use.

3.1.10 Other Calls

It currently appears that Solaris does not support use of the non-reentrant `getfoobybar()` interfaces in a multi-threaded application regardless of whether concurrent calls are ever made. In particular, we experienced coredumps inside such functions when we first started testing with the Client Library linked against our multi-threaded RPC client. If this is true, we may want to provide wrappers in the Client Library for those interfaces so as to minimize the changes needed to make a single-threaded application work with the Client Library. Such wrappers don't actually need to be fully thread-safe or use thread-specific data—plain old static storage should be sufficient—but they should operate correctly when called from only the main thread.

3.1.11 Current Implementation Sketch and Details

3.2 Sleepycat DB Interface

3.2.1 List of Supported Sleepycat Calls

The Sleepycat RPC facility supports only a subset of the standard DB API. The supported calls are:

```
env_cachesize     db_bt_maxkey      db_put            dbc_del
env_close         db_bt_minkey      db_re_delim       dbc_dup
env_create        db_close          db_re_len         dbc_get
set_lk_conflict   db_create         db_re_pad         dbc_put
set_lk_detect     db_del            db_remove         lock_detect
set_lk_max        db_flags          db_rename         lock_get
env_open          db_get            db_stat           lock_id
env_remove        db_h_ffactor      db_swapped        lock_put
txn_abort         db_h_nelem        db_sync           lock_stat
txn_begin         db_key_range      db_cursor         lock_vec
txn_checkpoint    db_lorder         db_join
txn_commit        db_open           dbc_close
txn_prepare       db_pagesize       dbc_count
```

3.2.2 Client Library Internal State, DB API

Most of the state needed for the Sleepycat DB interface is already kept internally by the Sleepycat DB library itself. The only state kept by the Client Library itself is the list of socket address for the

available DB/RPC Client Daemons and an indicator of which will be used next, plus a mutex to protect it. The socket address list will be initialized from the `/etc/diode/dbsock.conf` file, while the "next to be used" pointer will be initialized with the process's PID modulus the number of entries in the table.

Support for multiple Client Daemons, part II

In order to spread the load of Sleepycat DB calls across the Client Daemons, the Client Library will intercept the `DBENV→set_server()` call and redirect it to the next socket in the table, wrapping to the beginning after the last entry. To support this, we may back-port the new `DBENV→set_rpc_server()` call from version 3.3.

3.2.3 A Note on Transactions

While DI/ODE supports the Sleepycat RPC API, it does not support the full semantics. Application hints, described later, will be used to divide tables into several fragments distributed to different back-end machines. Transactions are not allowed to span fragments. Cursors will give correct results only as long as the cursor traversal does not cross fragment boundaries. (However, a blind first-to-last cursor traversal will eventually reach all entries, though not necessarily in the btree order.)

3.3 Multithreaded RPC Client Support

3.3.1 Goal

The goal is to create a fully multithreaded RPC client-side library which will allow us to use a single stream connection for multiple simultaneous RPC requests, and handle out-of-order replies. We can use this for hooking up applications to either the file service or the database service of the DI/ODE Client Daemon.

3.3.2 Existing work

The original SUN ONC/RPC library is the basis for the RPC support code in most UNIX platforms. This basic code's implementation of RPC-over-TCP runs the socket in half-duplex: the request is sent, then the XDR stream is reversed and the reply is received. It is not thread-safe.

Solaris provides a "multithread-safe" RPC client library, but it apparently just wraps a mutex around all use of the connection, serializing all RPC requests. We'll need better concurrency for higher performance.

The Erlang ONC/RPC package already provides this multithreaded capability for the DI/ODE Client Daemon [**?**].

3.3.3 Thread and Synchronization Model

For portability to POSIX-compliant platforms, we use the pthreads threading model and API in our implementation. We rely on the ability to create a thread, serialize threads on a blocking mutex, suspend threads on a condition variable, create thread-specific data, cancel a thread, and register cancellation cleanups. Sadly, all of these capabilities are quite necessary, so we can't go with a more thread-agnostic model, such as Sleepycat's.

A single stream socket connects to the RPC server. Each application thread making an RPC call will send its request on the socket and suspend itself. The RPC library runs its own daemon thread to receive RPC replies from the socket, peek at the RPC transaction ID (XID), and awaken the appropriate thread.

Application threads contend for a single mutex to serialize sending on the socket. The daemon thread is the only thread which reads from the socket, hence it does not need to synchronize in order to do this. However, the data structure describing all outstanding requests, described below, is accessed by all application threads and the daemon thread, hence is protected by a mutex. When an application thread suspends itself to wait for a reply, it uses a private condition variable, paired with the mutex controlling the outstanding request data structure.

To minimize contention for the mutexes, each application thread will perform XDR encoding and decoding outside of the above synchronization. Since XID assignment requires a critical section of code, application threads will build RPC requests with an XID of zero, but the request XID will be overwritten with a real XID in a critical section.

Since an RPC server may send a quick reply after the application thread sends its request but before it can take any other action, the application thread must register its information in the outstanding request data structure before it starts sending its request. In the case of a quick reply, or unfortunate thread scheduling, the reply may come before the sending thread manages to suspend itself. Therefore, each condition variable is paired with a boolean predicate variable, indicating whether or not a reply has been received.

To prevent an application thread from waiting forever in case the server loses its request, each RPC call will specify a timeout. This timeout limits how long the requesting thread will wait on its condition variable. If a thread wakes up from its wait with a timeout return value *and* its predicate variable is still false, then it removes itself from the set of outstanding requests and returns an error value. If the server later sends a reply and the daemon thread cannot match it up with any outstanding request, it will assume that the requester has timed out and silently discard the reply.

Cancellation and shutdown.

Invariants:

1. Only the thread holding Ms will use the sending FD.
2. Only the thread holding Mr will read or modify XIDNEXT, STATE, or the table.
3. Client threads will examine HaveReply flag before waiting on their condition variable.
4. The daemon thread will set a client's HaveReply flag before signaling their condition variable.

3.3.4 Data Structures and Memory Management

The set of outstanding requests is implemented as a linked list. Though this data structure has poor asymptotic performance, it is simple and fast when the set size is small. Its size is limited by the number of threads in the application, so if the linked list traversal time is becoming significant, we have bigger problems to worry about.

Each application thread allocates a buffer to hold the marshaled request message. In order to size this buffer, an initial XDR stream to a dummy memory region is created, and the `xdr_sizeof()` call is used to determine the size requirements for the request. This initial XDR stream is only required because of the `xdr_sizeof()` API. The application thread then allocates its correctly-sized request buffer (leaving space for the RPC header with its authentication fields), creates an XDR stream into it, marshals the request and RPC header, and then destroys the XDR stream. Once the request is sent, the buffer is freed.

The daemon thread reads the 4 byte record marker from the socket which gives the size of the first fragment of the reply. Since the reply should almost always be a single fragment, it simply allocates a buffer to hold the entire marshaled reply. In case of a multi-fragment reply, it will `realloc()` the buffer as needed. Once the daemon gives the requester thread the reply buffer, the requester is responsible for freeing it.

Upon receiving the reply buffer, the requester creates another XDR stream for decoding the buffer, decodes the reply, destroys the XDR stream, and then frees the buffer.

3.3.5 API

```
CLIENT *clntfd_create_r(u_long prog, u_long vers, int fd)
CLIENT *clnttcp_create_r(struct sockaddr_in *raddr, u_long prog, u_long vers, int *sockp)
CLIENT *clntunix_create_r(
        struct sockaddr_un *raddr,
        u_long prog, u_long vers,
        int *sockp,
        u_int sendsz, u_int recvsz
)
```

This API should be fully documented.

Chapter 4

Client Daemon

4.1 Common Overview

The DI/ODE Client Daemon consists of two independent parts: the file side and that Sleepycat DB multiplexer. Each will be documented separately.

The Client Daemon is written in Erlang, a concurrent functional programming language, for reasons detailed in our Erlang User's Conference paper [FLC+00].

4.2 Consistent Hashing

Both the File service and DB service use a common algorithm to assist in the efficient migration of data.

In order to support arbitrary scalability of the number of Data Servers and Metadata Servers, care must be taken in partitioning data. An algorithm called *consistent hashing* [KLL+97] is utilized to partition data and metadata into *migration units*.

Consider a set of Data Servers across which data are stored. For online scalability reasons, it is critical that we be able to increase or decrease the number of Data Servers with a minimum of data copying to re-balance the amount of data stored on the Metadata servers. Consistent hashing meets this requirement. When adding the n-th back-end storage unit, consistent hashing rebalances the data distribution, changing the location of only $\frac{1}{n}$ of the data. Furthermore, data are not redistributed among the storage units which haven't been changed.

Consistent hashing works in two stages. The input to the hash, called the *handle*, is first fed through a conventional hashing function which maps the input handles uniformly onto the range of integers $\{1, \ldots, M\}$ Each back-end storage unit is given a *bucket set*. A bucket set B_i consists of some number of *buckets*, $\{b_{i1}, \ldots, b_{iK}\}$, each of which is just an integer in $\{1, \ldots, M\}$. The buckets in a bucket set are also spread uniformly across the hash range. Once a handle is mapped onto the hash range, it chooses the closest bucket to its left (wrapping around if there are no buckets to the left), and thus falls into one of the bucket sets.

If we have a large enough set of handles, they should map uniformly to the bucket sets. When we add a bucket set, only a small fraction of the handles will see the new buckets as their closest left neighbor. Similarly, if we delete a bucket set, those handles which formerly mapped to it will now be scattered uniformly among the remaining bucket sets.

Note that K, the size of the bucket sets, must be constant, hence not growing as the number of bucket sets grows. If N grows large relative to K, then the bucket sets will no longer have a uniform distribution across the hash range, and irregularities can start to appear. Therefore we will choose K to be large relative our our expected N, say 1000.

All of the above must be made explicit, so that all of our Client Daemons can determine the same hash values. Also, we need to be mindful that our algorithm will work efficiently with Erlang, which does arithmetic with integer semantics (bignums) rather than modular bit-fields. As an implementation quirk, we can greatly improve the efficiency of our Erlang integer calculations if we keep the quantities below $2^{27} - 1$, or about 128 million.

Our first-stage hashing algorithm is could be just about anything, ranging from MD4 at the expensive end, to a simple Knuth-like congruent hashing function. It will be determined later. It will map onto the set $\{0, \ldots, M-1\}$ where $M = 100\,000\,000$.

The location of bucket b_{ik} is $(1000i + k)X \bmod M$, where $X = 61\,803\,399$. We choose X to be the nearest integer representation of $M\phi$, where ϕ is the golden ratio $(\sqrt{5}-1)/2$. Successive multiples of the golden ratio modulo the unit interval have the amazing property that they always fall into the largest interval free of previous buckets. Since M and X are relatively prime, the buckets will eventually fill every integer between 0 and M.

Once the number of bucket sets is decided, a binary tree, or other data structure, can be constructed to perform the handle hash to bucket set mapping in $\mathcal{O}(\log B)$ time, where B is the number of buckets.

4.3 File Architecture

The file side of the Client Daemon accepts File Client Protocol requests from applications and makes NFS requests of the back-end Data Servers. Table 4.1 gives a simplified mapping of the two protocols.

Since the Client Daemon performs the same tasks as the UNIX kernel file service components, it makes sense to inherit the UNIX kernel architecture, at least in terms of its major data structures.

4.3.1 Session State

Corresponds to User File Descriptor Table in UNIX kernel:

session ID There will be an indexing method to find a session by its session ID.

"xtra" state This is uninterpreted by the client daemon. Currently it contains the file descriptor of client's socket to Diode server, so that it can be closed after an `exec`.

current directory Dnode table entry.

File ID table For each `fid` we store:

- fid
- open file table id
- flags

The *flags* member in the fid table stores the 'close-on-exec' flag. This flag is always cleared when a new fid added to the table. If the first request on a new connection is `setsessid`, then after associating the connection with the specified session, the client daemon must close all the fids whose 'close-on-exec' flag is set.

> **Implementation Note:** We may choose to keep the session state in a single ETS table, or in the `gen_server` state of the Erlang process serving the session.

The DI/ODE protocol has three kinds of commands:

- operations on sessions: `forksess()`, `fiddump()`, ...
- operations on fids: `read()`, `write()`, `seek()`, ...

File Protocol Procedure	Client Daemon Procedure
`open`	*namei()*, optional NFS `CREATE`
`close`	session op
`read`	NFS `READ`
`write`	NFS `WRITE`
`pread`	NFS `READ`
`pwrite`	NFS `WRITE`
`seek`	open file op
`delete`	*namei()*, NFS `REMOVE`
`rename`	*namei()*, NFS `RENAME`
`link`	*namei()*, NFS `LINK`
`mkdir`	*namei()*, NFS `MKDIR`
`rmdir`	*namei()*, NFS `RMDIR`
`setattr`	*namei()*, NFS `SETATTR`
`stat`	*namei()*, NFS `GETATTR`
`fstat`	NFS `GETATTR`
`fsetattr`	NFS `SETATTR`
`fstatfs`	NFS `STATFS`
`statfs`	*namei()*, NFS `STATFS`
`fdirlist`	NFS `READDIR`
`flock`	NFS `CREATE` or NFS `LOOKUP`
`fiddump`	session op
`setsessid`	session op
`forksess`	session op
`listmounts`	root table op
`resolv`	*namei()*
`dup2`	session op
`chdir`	*namei()*
`fchdir`	session op
`pathconf`	*namei()*, NFS `PATHCONF`
`fpathconf`	NFS `PATHCONF`
`fsync`	NFS `COMMIT`
`getflags`	open file op
`setflags`	open file op
`setuid`	session op
`setgid`	session op
`setgroups`	session op

Table 4.1: Translation of File Protocol to NFS Operations (simplified)

The *namei()* procedure comprises several NFS `LOOKUP` calls.

- operations on paths: `open()`, `mkdir()`, `rename()`, ...

Operations on sessions are local to this layer. Operations on fids go down to the open file layer. Operations on paths get interesting.

Operations on paths generally have to resolve to a dnode for a directory, followed by an operation local to that directory (potentially two directories for `rename()`). Paths come in two flavors: relative and absolute. Relative paths are resolved through a chain of `lookup()` operations on directory dnodes, starting with the current directory dnode. Absolute paths need to use the structure in the next section.

Root Table

The DI/ODE protocol specifies a mount index for many of its operations. The root table holds system state associated with the protocol's mount index:

mount id The mount identifier as used in the DI/ODE protocol.

layers The number of intermediate directories in this mount point, zero if none.

root id The Dnode identifier of the root of the mounted "partition".

This table is expected to be small, on the order of 2-4 entries.

4.3.2 Open File Table

Corresponds to Open File Table in UNIX kernel, holding state that may be shared among different `fid`s in related sessions.

open file id The key referenced by session `fid`s.

refcount One per `fid` reference.

offset 64-bit.

lock state `nil` or state and Pid of lock refresher.

open flags r, w, rw, . . .

dnode id Reference to Dnode table.

Since all locks are leases, any lock will have a helper process attached to it to refresh the lock.

> **Implementation option:** Due to overheads of immutable data structures, and limitations of ETS library, things like the open file table may actually be implemented as a set of tables:
>
> - open file table:
>   - * id
>   - – open flags
>   - – dnode id
>
>   open file refcount table:
>   - * id
>   - – refcount
>
>   open file lock state table:
>   - * id
>   - – lock state

open file offset table:

* id
– offset

("*" denotes the key of the table) though the `ets:increment_counter()` function may do what we want for some of these fields.]

Most operations on open files just pick up some open file state and proceed to operate against the underlying dnode.

4.3.3 Dnode Table

A "dnode" plays the same role in DI/ODE as an inode or vnode in a traditional UNIX kernel - to represent a file or directory, and cache some of its information in memory.

id Primary key.

refcount Sum of open file table plus parent plus session `pwd` references. Nodes with a positive layer (see below) may be destroyed once the refcount reaches zero, but nonpositive layers will persist.

parent A reference to the containing directory, or self if this is the "mount point". Due to restrictions placed on `rename()`, this field is immutable.

type Indicates whether it is a file or directory.

root Index into root table.

layer Signed integer field giving level relative to the consistent hashing level. Zero indicates that lookups on this directory must go through consistent hashing. Positive indicates levels within a migration unit. Negative indicates intermediate directories. If the layer is not positive, then this must be a directory node.

subdirs If the layer is negative, a list of {`name, dnode`} tuples for the subdirectories at this level.

fhlist A list of {`DataServer, FileId, FileHandle`} tuples. Positive levels will have a single entry (except during migration). Nonpositive levels will have one entry per Data Server.

name This node's name within its parent directory. Renames by other Client Daemons may invalidate this data, so be careful.

Implementation option: Instead of maintaining the **fhlist** as a list, we can maintain an ETS ordered set of tuples of the form {{`DS, Dnode`}, `FH`}. This gives a quick way of mapping to filehandles, as well as a convenient way to accessing all filehandles associated with a particular Data Server, by stepping through them with `ets:next/2`.]

Dnodes behave differently, depending on their level. In particular, for the following operations:

lookup layer < 0 Look at the **subdirs** field.

layer $= 0$ Go through consistent hashing to determine the Data Server, look up the information in the **fhlist**, then perform the NFS operation on the correct Data Server.

layer > 0 Perform the NFS operation on the single Data Server.

dirlist layer < 0 Dump the list of **subdirs**.

layer $= 0$ For each Data Server, perform the NFS operation and merge the results.

layer > 0 Perform the NFS operation on the single Data Server.

mkdir, rmdir layer < 0 Not allowed.

layer $= 0$ Go through consistent hashing to determine the Data Server, look up the information in the **fhlist**, then perform the NFS operation on the correct Data Server.

layer > 0 Perform the NFS operation on the single Data Server.

If the same file is opened from different sessions, they must share the same dnode. (Migration will require this property.) Therefore, the creation of dnodes must be single-threaded to resolve races.

4.3.4 NFS Mount Table

This holds the information from the CD config table, as well as dynamic state associated with the RPC connection and NFS mount data.

ds-diode-root {`ds, diode_root`} pair, for quick indexing.

remote-path

nfs-version

transport

rpc-client

root-fh

4.3.5 Consistent Hash Bucket Map Table

We need one table to map first-stage hash values to bucket sets, as described above, and a second table to map bucket sets to Data Servers.

4.3.6 Namei

Need to describe how `namei()` handles migration, mount points, etc.

Basic idea: for open files and current directories, we keep a Diode inode around - which stores enough information to use. Namei can create new ones?

Hmm. Namei *has* to create new ones, in case you're using it for, e.g. and open() chdir(), so the caller is responsible for decrementing the refcount after it's done with the namei() result.

4.3.7 Locking

In this release of DI/ODE, file locking will be performed using lock files: when a file F is to be locked, a file named F`.lock` is created in the same directory. As such, a lock is attached to the name used to open the file and not the file's inode. Unlinking a locked file, renaming a locked file, or locking a file with multiple hardlinks will exhibit non-standard behavior. In the first case a 'dangling' lock will be left, while in the last case a lock on one name will not keep another name for the same file from also being locked. Renaming a locked file both unlocks the file and leaves the original locking 'dangling'.

For DI/ODE, we require our applications to not rename or unlink file that other processes might have locked. The lock file itself is just another file as far as migration is concerned. Our file creation implementation of the `flock()` call must have the same exclusivity properties that we guarantee `creat()` during migration. The lock file must be occasionally touched to refresh the lease. Lease refreshing is performed automatically by the Client Daemon as long as the application session which requested the lock remains active. To delete the stale lock file, use the "tower of locks" algorithm, whose names are F.*level*`.lock` (with level starting at 0 and going through 9).

4.3.8 Linking and Renaming

To make migration feasible, we restrict links and renames to be within a single directory. Therefore, each inode has a unique and constant parent inode.

4.3.9 Process Tree

We need a dedicated Erlang process associated with each Client Protocol connection, due to constraints of the socket library. Since we're interleaving RPC requests from different threads over the same connection, we will dispatch a new thread for each request.

Depending on the exact implementation of the larger data stores (session states, open file table, dnode table), we may need dedicated processes for submitting and/or retrieving data from these stores.

Each RPC client to a back-end Data Server is a separate process.

4.3.10 Migration

The file migration algorithm is described in Chapter 7.

For reasonable performance, consistent hashing should only be calculated upon first lookup of the hashing handle. Data structures within the dnode table will support the Data Server mapping in one location.

This single location will need to be updated during the different phases of migration. We must be careful that a single operation making multiple queries into this table won't operate on inconsistent replies.

In order to acknowledge a change in the migration phase, the Client Daemon must ensure that all operations based on the previous phase have completed. We can accomplish this by having each session sequentially number the requests it has received, and keep track of those which have completed. When the mapping table has been updated to reflect the new View, each session will be asked to mark its current request number, and then reply when all equal and lower-numbered requests have completed. The Client Daemon can then acknowledge the phase transition.

4.4 Database Handling

The Client Daemon is, among other tasks, responsible for multiplexing Sleepycat DB RPC calls onto a set of Metadata Servers. A number of complexities arise from converting a centralized database to a distributed database. The following sections describe some of the design challenges and algorithmic design of the Sleepycat DB RPC multiplexer.

4.4.1 Theoretical constraints

Following is a brief theory-based description of some of the main issues, along with the solutions that appear most likely to fit our needs. For a more complete discussion of distributed transactions, see [GR93, BHG87, Gos91, Tan92].

General need for two phase commit

An application-initiated DB transaction is translated by the DB Multiplexer into n multiplexer-initiated transactions, where n is the number of Metadata Servers that the application transaction operates on. n is unknowable to the application. An error in any of the multiplexer transactions is handled by aborting all associated multiplexer transactions and returning a transaction error to the application. If all of the multiplexer transactions proceed without error, then when the application transaction is committed, all multiplexer transactions must be committed. If some, but not all of the

multiplexer transactions succeed, then the application transaction ends up in a partially committed (inconsistent) state due to the non-atomic nature of the set of multiplexer transaction commits. There are two possible solutions to this problem:

1. Use two phase commit. Before committing any of the multiplexer transactions, prepare them for commit, so that if there is a failure during commit, all of the prepared transactions can be rolled forward (committed) or rolled backward (aborted) during a recovery process.

2. Make sure the application can handle all possible inconsistencies.

Two phase commit is proven to work correctly, but requires considerable extra work to implement. On the other hand, assuring that the application can handle all possible inconsistencies is difficult and prone to error on an ongoing basis.

Distributed deadlock

Sleepycat DB includes facilities for deadlock detection and resolution. However, since we are distributing the database across multiple Metadata Servers, there is the potential for distributed deadlock, which is undetectable at the individual Metadata Server level. There are a several algorithms that address distributed deadlock detection and resolution. Algorithms for distributed deadlock detection and/or resolution include:

- Assure that no transactions have the potential for distributed deadlock. In general this is a very difficult approach, but it may be feasible for the limited set of transactions that our application does.

- Abort transactions that make no progress within a timeout period. This is the simplest approach, but has the disadvantage of introducing delays in deadlocked transactions. In addition, since no forward progress is guaranteed (old transactions may be aborted in favor of younger transactions), there is the potential for livelock.

- Use the wound-wait algorithm. Assign all transactions a global sequence number. Whenever there is contention for a resource, abort the younger transaction. This approach has the disadvantage that it opportunistically aborts transactions to avoid any deadlock, when statistically few of the resource contentions will result in deadlock.

- Use a global lock manager. All locks are controlled by a single entity. This solution has significant negative performance implications, since locking needs to be fast, and IPC is generally one to three orders of magnitude slower than native locking.

- Maintain a global "waits-for graph". Every time a transaction has to wait for a lock, the waits-for graph is updated, and when a cycle is detected in the graph, one or more transactions are aborted.

- Use what is referred to as "edge pushing" to communicate potentially interesting waits-for graph edges between Metadata Servers, but do not maintain a global waits-for graph. Edges are pushed in such a way that at least one Metadata Server can detect any deadlock.

We will use timeouts for deadlock recovery (option #2 in the list above) in the initial Client Daemon implementation. We may have to spend time hand tuning application transactions to reduce the deadlock rate if performance suffers as a result of the naïveté of using timeouts for deadlock resolution. We have chosen this method because it's easy to implement and we are expecting that the number of deadlock/livelock we actually encounter in practice will be small. If this is not the case, then we will try would-wait. If that doesn't work, then Sleepycat DB is probably not a good choice for that application.

We will need to empirically determine an appropriate timeout interval.

Distributed deadlock outside of transactions

It appears as though there is no recourse for distributed deadlock that does not occur within a transaction. That is, it is possible for two simultaneous application requests to result in distributed deadlock, and there will be no transaction timeout to break the deadlock. This problem can be resolved in any of the following ways:

- Make sure the application never acquires more than one lock at a time outside of a transaction. In the case of Sleepycat DB, this is feasible as long as records are small enough to avoid the use of overflow pages. This limitation also applies to direct use of the Sleepycat DB locking API.
- Modify Sleepycat DB to time out operations even outside of transactions.
- Implement a distributed deadlock resolver that is not based on timeouts.

Only the first method is being supported for the initial release. As such, applications are required to comply with this limitation.

4.4.2 Categorization of Sleepycat DB RPC messages/responses

Depending on the type of Sleepycat DB RPC message, there are different mechanisms for handling messages/responses.

The following messages are broadcast to all Metadata Servers:

- env_cachesize (May be removed from the Sleepycat DB RPC protocol.)
- env_create
- env_remove
- db_bt_maxkey (Global maximum is returned.)
- db_bt_minkey (Global minimum is returned.)
- db_create
- db_flags
- db_h_ffactor (We may want to adjust this.)
- db_h_nelem (Results are merged.)
- db_key_range (Results are merged.)
- db_lorder
- db_pagesize
- db_re_delim
- db_re_len
- db_re_pad
- db_remove
- db_rename

- db_stat (Results are merged.)
- db_sync
- lock_detect
- lock_stat (Results are merged.)

The following message pairs are mapped to a single reference-counted broadcast to all Metadata Servers:

- env_open, env_close
- db_open, db_close

The following messages can be sent to any Metadata Server:

- db_swapped (Not used by some applications.)

The following messages are sent to one or more Metadata Servers, according to consistent hash results:

- txn_abort
- txn_begin
- txn_checkpoint
- txn_commit
- txn_prepare
- db_cursor
- db_del
- db_get
- db_join (Not used by many applications, very complex to implement.)
- db_put
- dbc_close
- dbc_count
- dbc_del
- dbc_dup
- dbc_get
- dbc_put
- lock_get
- lock_id
- lock_put
- lock_vec

lock_id deserves special mention, since a single lock ID may need to be used across multiple Metadata servers. The multiplexer will need to maintain a lazily expanded table of application lock IDs that map to per-Metadata Server lock IDs. This is similar to how transaction and cursor IDs need to be managed.

In order to support lock migration, lock_get and lock_put maintain a table of outstanding locks (acquisition pending or owned). See Chapter 8 for details.

4.4.3 Broadcasted DB operations

In the general case, the following algorithm is used to broadcast Sleepycat DB calls that need to be sent to all Metadata Servers. Such calls include both broadcast requests and reference counted broadcast requests.

Where $\mathcal{MDS}$ is the set of all Metadata Servers:

1. Receive request R from the application.
2. $\forall\{M : M \in \mathcal{MDS}\}$:
   (a) Create request R_m, which is a proxied version of R, and send it to M.
   (b) Receive the response to R_m. If there is an error, try to back out all operations associated with R and return an error to the application.
3. Send a response to the application.

Since the above algorithm is not protected by transactions, additional care must be taken to detect and handle situations where not all Metadata Servers are in the same state. For example, during a db_create call, if some, but not all Metadata servers already contain the table that is to be created, the table must first be removed from Metadata servers that contain the table.

4.4.4 Multiplexed DB transactions

All Sleepycat DB calls within a transaction must be transparently multiplexed onto a set of transactions that map directly to the set of Metadata Servers involved in the transaction.

A typical successful application-initiated transaction consists of roughly the following steps from the application's perspective:

1. Begin transaction T.
2. A sequence of any of the following operations (not an exhaustive list):
   - Get a record.
   - Put a record.
   - Perform a child transaction.
   - Perform a cursor operation.
   - Perform a lock operation.
3. Commit transaction T.

Given the set $\mathcal{MDS}$ of Metadata Servers, this gets translated by the multiplexer to:

1. Create a transaction number for T and return it to the application.
2. $\forall R \in T$, where R is a request:

   (a) Use the consistent hash to determine the set $\mathcal{M}$ of Metadata Servers, where $\mathcal{M} \subseteq \mathcal{MDS}$, that R maps to.
   (b) $\forall \{M : M \in \mathcal{M}\}$:
      i. Create a request R_m to send to M.
      ii. If M has not yet participated in T, create the list of parent transactions on M.
      iii. Send R_m to M.
      iv. Receive the response to R_m and translate it for later collation into the response to R.
   (c) Return the collated response to R to the application.

3. Prepare all transactions associated with T.

4. Commit all transactions associated with T.

Chapter 5

Data Protocol

5.1 File Interface

5.1.1 Implementation

The File Interface Data Protocol is NFS version 3. The Client Daemon also supports NFS version 2, but we do not support its use for Boardwalk. We support NFS transport over TCP or UDP, but at this point UDP seems marginally preferable.

There are pieces of the NFS protocol that we do not currently support. The unsupported commands are: SYMLINK, MKNOD, and READLINK. We currently choose not to implement READDIRPLUS. While READDIRPLUS was designed to be a performance benefit, when performed on NFS servers that have directories with a large number of entries by clients that are only interested in one directory element at a time, it can be a serious performance bottleneck. Substantially, we expect that our applications fall into this camp. If, later, we want to use READDIRPLUS for some reason, we can implement it, although its implementation would be challenging.

5.2 Sleepycat DB Interface

Sleepycat DB calls are sent over ONC RPC. Rather than encoding and sending the actual internal DB structures (`DB`, `DB_ENV`, etc.), these are represented over-the-wire using 32-bit ID numbers, allocated by the server when a request creates a new such structure.

5.2.1 Goals

- The server should be able to take advantage of and load-balance across multiple CPUs.
- Actions that would have been violations of the DB library's threading requirements (for example, simultaneous requests to the same cursor, or to a cursor and its associated transaction) do not need to be serialized by the server: while it should be robust in the face of such bugs in the client, the server is not intended as a means of lifting such restrictions. Returning failure for the violating call and logging the event will be considered sufficient.
- In general, the server should be robust in the face of client-side errors and crashes. It should not leak memory or itself crash, no matter what the client does.
- While initially developed for Solaris 2.x, the server should be portable to, at least, other pthreads implementations.

- While it would be 'nice' if the same code-base could be compiled as either a single threaded or multithreaded server, that is *not* a requirement of the Boardwalk project.
- The server needs to handle the calls made by applications. Design limitations that are outside of an application's usage of DB are tolerable, if disliked. (See section 5.2.6.)
- The design and product should be supportable by Sleepycat. This implies that design limitations must be acceptable to the wider DB community or resolvable without undue effort.
- The server should hide the recovery process from clients as much as is possible.

5.2.2 The Threading Constraint

The DB library requires that all cursor and transaction calls be made by the thread that created the cursor or transaction. All requests involving a given cursor or transaction must be forwarded to the correct thread and performed there.

Actually, this is not true. The DB library merely requires that no cursor or transaction be 'involved' in concurrent calls. The design documented here was written using the previous, stricter constraint. As this design has been reviewed both internally and by people at Sleepycat and appears correct, and as the implementation of it appears to meet our initial performance goals, we have decided to postpone the redesign of the daemon under the more lenient version of the threading constraint until such time as it becomes a performance, management, or maintenance issue.

5.2.3 Prior Work

Sleepycat DB version 3.1.17 includes an RPC server. However, it was written mainly as a proof-of-concept and for debugging purposes, and is single threaded. While portable and an excellent base to work from, it doesn't meet our performance goals.

In Solaris 2.4, Sun extended rpcgen and added library entry points to allow an RPC server to be multithreaded. The threading API thus implemented supports two different modes of operation, *auto* and *user*. The former automatically processes each request in a separate thread, while the later calls invokes the dispatch function (e.g., `db_serverproc_1()`) in the same thread but lets the programmer hand off the task to other threads and delay the response as needed. The Solaris implementation only supports multithreaded operation when used with TI-RPC (Transport Independent RPC), built on top of the TLI network API. Together, this renders the RPC startup routine generated by rpcgen practically unportable, though the general thread-safe calling conventions generated by rpcgen's `-M` flag may be used even in a single threaded server, albeit with some loss in speed. We have confirmed that the Solaris multithreaded RPC server implementation allows requests and replies on a single connection to be interleaved.

In short, this is a functionality that is built-in to Solaris that may have to be implemented by developers if we port the Sleepycat DB interface to DI/ODE to other operating systems.

5.2.4 Implementation

Thread models

In order to meet the DB library's thread requirements, the server will need to associate a backend ("worker") thread with each tree of transactions or cursors. To maximize concurrency, each top-level transaction will have its own worker thread, as will each transactionless cursor that isn't a duplicate of another cursor. Note that when a transactionless cursor is duped with `DBC→c_dup()`, the new cursor must be associated with the same worker thread.

The RPC side ("frontend") of the server will need to decode the request and, for those requests involving cursors or transactions, make an 'inter-thread call' to the correct worker thread. Where

the RPC server developed by Sleepycat has a single function to perform the processing for a request, the multithreaded server will need two: one that does the ID→DB entity mapping and inter-thread call and is executed by the request thread, and one that performs the actual DB call and is executed by the correct worker thread.

Threading the frontend of the server is complicated by the RPC service entry routine, `svc_run()`, being in the system libraries. Using the Solaris version as a model, a multithreaded implementation of the `svc_run()` will eventually need to be written. Development of the rest of the server can take place under Solaris using the system version.

The Solaris implementation introduces a dichotomy between what it calls *auto* and *user* mode. In *auto* mode, the user defined `serverproc()` routine (e.g., `db_serverproc_1()`) is executed in a separate thread for each request. In *user* mode, the `serverproc()` routine is executed in a single-threaded manner—`svc_run()` will not accept another request until the `serverproc()` routine returns—but the actual reply can be sent asynchronously from another thread.

As applied here, the *auto* mode would result in a 1–M–N model: one thread in `svc_run()` performing the select/poll loop, M requests threads handling the concurrent requests, and N worker threads to perform the transaction and cursor specific calls. This can be done with no changes to the `db_serverproc_1()` routine created by rpcgen.

The *user* mode would allow a 1–N+ model: one thread performing the select/poll, XDR decode, and selection of worker thread, N worker threads handling for transaction and cursor calls, with additional threads handling other calls (the '+'). This would require splitting in half the `db_serverproc_1()` routine, or just writing it from scratch (the Sun Answerbook itself illustrates user mode with a hand-written `serverproc()` routine).

While coding to a Solaris specific API may seem inadvisable, Sun has released the code for their multithreaded `svc_run()` routine and other OSes are starting to bring in this new code. For example, the main branch of FreeBSD development now includes a version of TI-RPC and plans are being developed for the integration of the multithreaded code. Given that, targeting that API seems a good choice for long term portability. If it is determined that we need to release Boardwalk or some later version on a platform that has not yet taken up Sun's code, then we can at that time implement our own version, probably starting from the code that Sun has released. Either way, doing so allows us to develop a stable frontend implementation against which we can implement and test a backend that solves its half of the threading problem.

That said, the Solaris API has a few deficiencies that we have to work around. In particular, there is no way to perform a 'clean shutdown', wherein new connections would no longer be accepted and existing connections would only be usable for sending replies to outstanding requests (c.f. the `shutdown()` system call), nor is there any way for the RPC server to detect when a particular connection has been closed so as to perform cleanup of resources associated with that connection. We don't currently have a workaround to the former, while the idle-environment timeout provides for the eventual release of resources associated with a closed connection.

Writing our own `svc_run` would be unpleasant. This means that we need to manage graceful shutdowns ourselves, but since we need to handle abrupt shutdowns anyway, this shouldn't be an extra restriction. No matter what we do, at disconnect time, some connections will be processed, some will be gracefully aborted, and some will probably be lost. Our applications need to be able to handle these situations when they happen on a potentially inconsistent data repository.

Data Structures

Logically, the various DB structures—`DB_ENV`, `DB`, `DB_TXN`, `DBC`—are arranged in a hierarchy as in Figure 5.1.

Each of the DB structures has a corresponding structure in the RPC server—`ct_base_env`, `ct_db`, `ct_txn`, `ct_dbc`—that contains the pointer to the underlying DB structure and enough additional information to let the RPC server perform close operations and timeouts completely (no dangling

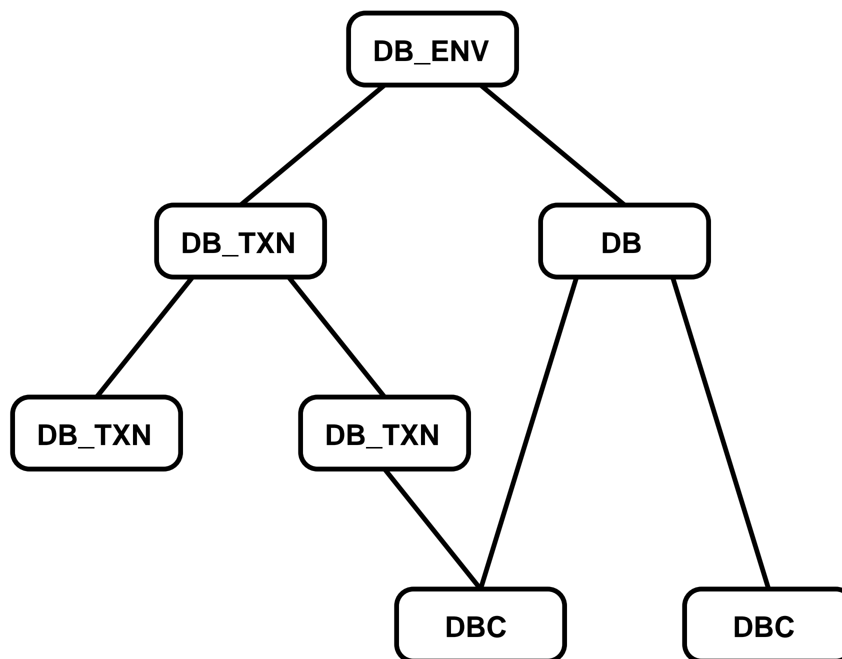

Figure 5.1: Database Structure Hierarchy

structures) and safely (properly ordered). The `DB_ENV` structure actually has two structures associated with it in the RPC server: `ct_base_env` and `ct_env`. The former contains the actual pointer as well as enough information to run recovery and reopen the environment. The latter is the per-connection view of the environment and contains the ID number and timeout information.

To provide a location for information specific to a worker thread, I've created the `ct_wrk` structure. This includes fields for passing the information needed for the inter-thread call, as well as the synchronization variables to protect them. This structure is inserted into the graph between top-level transactions and their parent environment, as well as between transactionless cursors and their parent environment.

The `NULL` transaction When a new top-level transaction or transactionless cursor is created the `txn_begin()` or `DB`→`cursor()` call is passed a `NULL` pointer. This is represented in the RPC protocol as a zero ID. To simplify management of the cursor and transaction lists, the `ct_wrk` structure includes an instance of the `ct_txn` structure that has a `NULL` `DB_TXN` pointer and a zero ID.

The logical graph therefore ends up looking like that in Figure 5.2, with the arrows pointing from children to parent transactions.

All of the links but one in Figure 5.2 are bi-directional: each structure keeps a list of the structures of each type that are beneath it, as well as a link to its parent or parents. The single exception is in the `ct_txn` structure: its 'parent' pointer references its associated `ct_wrk`, not its immediate `ct_txn` parent. The 'downward' links are necessary to allow 'close' operations and timeouts to properly close and free all structures beneath the structure being closed, while the 'upward' links are necessary for proper locking during list operations, for detecting cross-environment operations, and to make some tree operations more self-contained.

Note that all the lists are done using 'head' and 'next' pointers. For example, the `ct_db` structure contains a 'next' pointer for the list of `ct_db`s beneath its parent `ct_env` and a 'head' pointer for the list of `ct_dbc`s beneath it. The `ct_dbc` structures contains two 'next' pointers, one for its parent

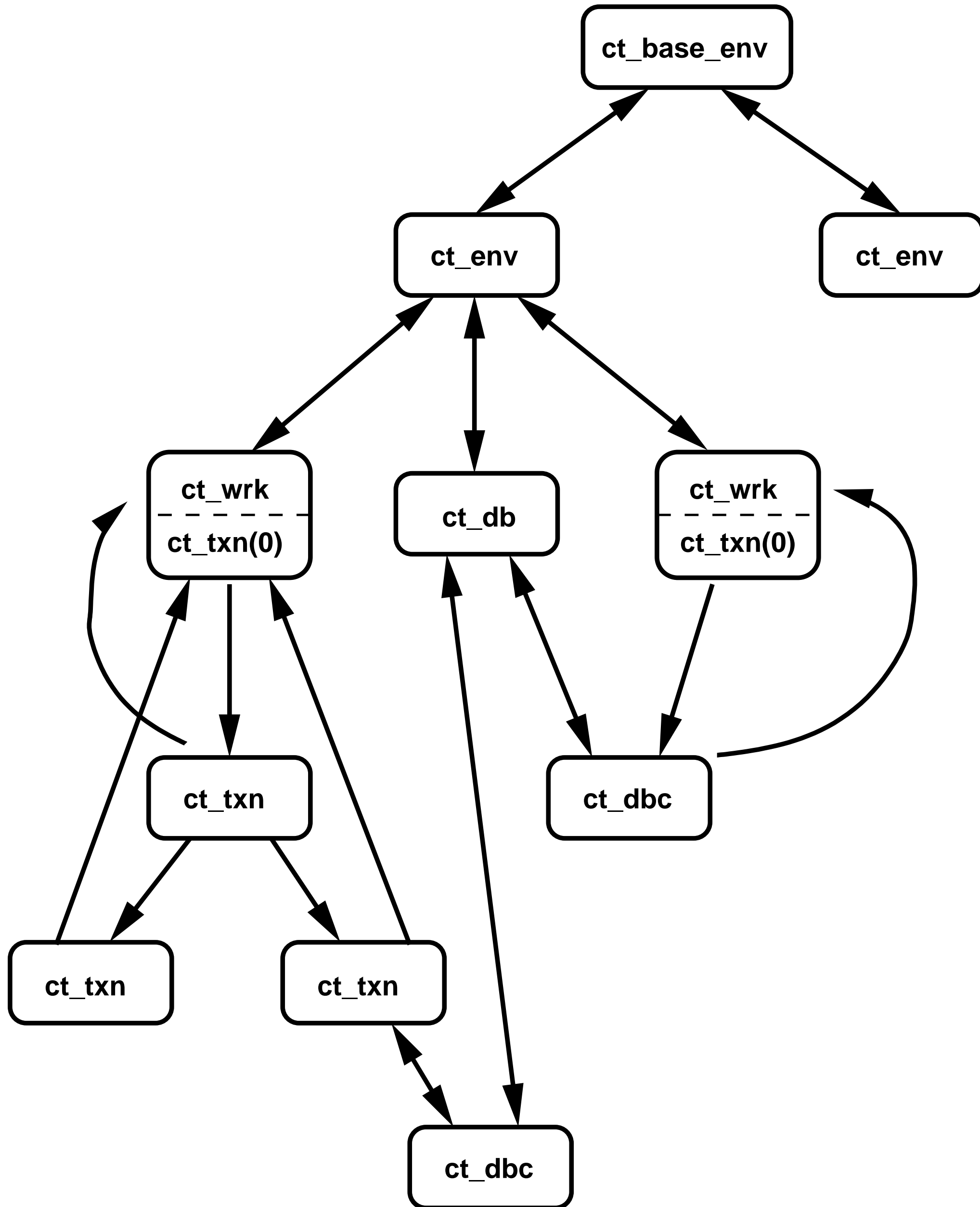

Figure 5.2: RPC Server Structure Hierarchy

`ct_txn`s list of cursors, and one for its parent `ct_db`s list. These are defined and accessed via the `LIST_*` macros defined in the `queue.h` header file. These macros and definitions permit the removal of an item from a list given just the pointer to the item itself and the name of the member containing the 'next' pointer.

Structure definitions

typedefs and forward declarations

```
typedef u_int32_t id_int;

struct ct_base_env;
struct ct_env;
struct ct_wrk;
struct ct_txn;
struct ct_db;
struct ct_dbc;
LIST_HEAD(ct_env_list, ct_env);
LIST_HEAD(ct_wrk_list, ct_wrk);
LIST_HEAD(ct_txn_list, ct_txn);
LIST_HEAD(ct_db_list, ct_db);
LIST_HEAD(ct_dbc_list, ct_dbc);

typedef void (request_proc)(struct ct_wrk *, void *argp, void *resultp);
struct request {
        request_proc    *req_proc;
        void            *req_argp;
        void            *req_resultp;
};
```

The `id_int` type is used to hold IDs and must match the type of the RPC field used to pass IDs (in particular, its maximum value better be no larger than that of the RPC field). The `LIST_HEAD(tag, item)` macro declares a structure, `struct tag`, to act as the head of a list of `struct item`. Similarly, the `LIST_ENTRY(item)` macro is used when declaring 'next' pointer structure members. The target function of an inter-thread call is cast to the `request_proc` type and called in that form, so argument types better be compatible! The `struct request` type packages up such a function pointer and the argument and result pointers which are passed between threads in the call itself.

```
struct ct_base_env      *db_base_envs;
size_t                   db_base_env_count;
```

This is the global array of 'exported' environments. This array is created at system startup and never changes size. When a request is received to open or remove an environment, the RPC server looks for the specified home in the structures in this array.

```
struct ct_base_env {
        /* head links */
        struct ct_env_list       ct_envs;

        /* synchronization */
        pthread_mutex_t          ct_mutex;
        pthread_cond_t           ct_cond;
```

```
        /* local data */
        unsigned int             ct_count;
        unsigned int             ct_flags;
#define RPCBASE_RECOVERING      0x01    /* recovery is in process */
#define RPCBASE_SHUTDOWN        0x02    /* the server is going down */
#define RPCBASE_DEAD            0x04    /* this base_env has shutdown */
#define RPCBASE_TXN_NEARWRAP    0x08    /* txn-ids are getting up there */
#define RPCBASE_TXN_WRAPPED     0x10    /* out of txn-ids, must handle it */

#define RPCBASE_CANTUSE         0x13    /* can't use it for some reason */

        /* items set when the environment is actually opened */
        DB_ENV                  *ct_envp;
        long                     ct_idle_timeout;
        long                     ct_max_timeout;
        long                     ct_def_timeout;

        /* items saved when it's opened in case we need to reopen it */
        u_int32_t                ct_open_flags;
        int32_t                  ct_open_mode;

        /* immutable items */
        const char              *ct_name;
        const char              *ct_path;
};
```

This is the server-wide, per-environment structure. Each instance of this structure has a thread associated with it that handles idle timeouts and auto-recovery.

The `ct_name` member is the value looked for by environment open and remove requests. The `ct_path` member contains the full path that will be passed to the DBENV→`open()` or DBENV→`remove()` call. The `ct_count` member is used during auto-recovery to keep track of how many `ct_wrk` have yet to finish closing their cursors and transactions.

If the `RPCBASE_RECOVERING` flag is set, then an operation in this environment has returned `DB_RUN_RECOVERY` and auto-recovery is being performed (c.f. Section 5.2.4). If the `RPCBASE_SHUTDOWN` flag is set then the server is being shutdown. When the thread for this `ct_base_env` has finished shutting down, it sets the `RPCBASE_DEAD` flag and kills exits. The `RPCBASE_TXN_NEARWRAP` and `RPCBASE_TXN_WRAPPED` flags are used to indicate when this environment is either getting 'close' to running out of transaction IDs, or when it has done so. They are set by a worker thread performing `txn_begin()`. Handling is similar to auto-recovery.

The `ct_envs` member is the head of the list of per-client-environment structures for this environment. That is, every time a client sends a DBENV→`open()` request for this environment, a `ct_env` is added to the `ct_envs` list. If the list was previously empty, the `ct_cond` member must be signaled on to guarantee that the thread for this `ct_base_env` will check its idle timeout. DBENV→`remove()` requests check this list and will fail with status `EBUSY` if the list is not empty. The `DB_FORCE` flag to DBENV→`remove()` is not supported.

The `ct_idle_timeout`, `ct_max_timeout`, and `ct_def_timeout` members are initialized to the global idle timeout, global max transaction timeout, and global default transaction timeout, respectively, but when the environment is opened these values may be overridden by settings in the environment's `DB_CONFIG` file.

The `ct_open_flags` and `ct_open_mode` members store the flags and mode that were passed to

DBENV→open() when it was first opened. These are saved here so that when the environment is reopened as part of auto-recovery it can be opened with the same values as before.

When support for the DBENV→set_flags() call is added, another member will be added to store the flags used with it, also for use during auto-recovery. The same goes for any other supported 'pre-open' environment call that sets state.

Finally, note that this array subsumes the __dbsrv_home list that previous designs have used to map environment names to their directories.

In all the structures that follow, the ct_id member is (for now) considered private to the ID subsystem and should not be examined or changed by other routines.

```
struct ct_env {
        id_int                  ct_id;

        /* next links */
        LIST_ENTRY(ct_env)      ct_next;

        /* head links */
        struct ct_wrk_list      ct_wrks;
        struct ct_db_list       ct_dbs;

        /* synchronization */
        pthread_mutex_t         ct_mutex;
        pthread_cond_t          ct_cond;

        /* local data */
        time_t                  ct_time;
        unsigned int            ct_count;
        unsigned int            ct_flags;
#define RPCENV_CLOSING         0x01            /* no new uses allowed */
#define RPCENV_RECOVERING      0x02            /* temporarily unavailable */
        struct ct_base_env     *ct_base;

        /* immutable items */
        long                    ct_timeout;
};
```

The ct_time member contains the time of the most recent of the following:

1. the processing of a request that directly specified this ct_env
2. the closing of a child database
3. the closing of the last transaction or cursor associated with a child worker

The ct_count member is a count of the number of requests currently being processed that directly specify this ct_env. The ct_time and ct_count members and the two list heads, ct_wrks and ct_dbs, are all protected by the ct_mutex member. Furthermore, whenever a thread removes the last item from either list, or decrements ct_count to zero, it should signal on the ct_cond member. The RPCENV_CLOSING flag is set when a DBENV→close() request has been received and indicates that the environment should be considered as already closed for new requests while old requests and handles are still being dealt with. The RPCENV_RECOVERING flag indicates that auto-recovery is in process and that requests that operate directly on this environment should wait on the ct_cond

member until the flag is no longer set. The `ct_base` member is set when `DBENV→open()` is called. This means that an actual `DB_ENV` structure is not created until the client does the open, and then only if that environment directory has not been opened for another request already. The `ct_timeout` member of the `ct_env` structure is initially set from the call that creates the environment and then updated against the `ct_max_timeout` and `ct_def_timeout` members of the `ct_base_env` structure when the environment is opened. Once the environment is opened the timeout cannot change until the `ct_env` structure is returned to the free pool.

```
struct ct_env_list      db_unopened_envs;
pthread_mutex_t         db_unopened_env_mutex;
```

When a `ct_env` structures is created by a client request, it is not associated with a `ct_base_env` structure. That doesn't happen until the client sends a `DBENV→open()` or `DBENV→remove()` request. Such unassociated `ct_env` structures are kept on the `db_unopened_envs` list.

```
struct ct_txn {
        id_int                  ct_id;

        /* next links */
        LIST_ENTRY(ct_txn)      ct_next;

        /* head links */
        struct ct_txn_list      ct_sub_txns;
        struct ct_dbc_list      ct_dbcs;

        /* parent link */
        struct ct_wrk          *ct_parent;

        /* immutable items */
        DB_TXN                 *ct_txnp;
};
```

The `ct_txn` structure has no members concerned with synchronization as, with one exception, it is only accessed by the worker thread that it is associated with. The exception is the `ct_parent` member which may be accessed by a request thread coming from a child cursor. This is fine, as the `ct_parent` member is set at allocation and is immutable until the `ct_txn` is returned to the free pool, and that cannot happen until all child cursors are closed.

```
struct ct_wrk {
        /* next links */
        LIST_ENTRY(ct_wrk)      ct_next;

        /* head links */
        struct ct_dbc_list      ct_dbcs;

        /* parent link */
        struct ct_env          *ct_parent;

        /* synchronization */
        pthread_mutex_t         ct_mutex;
        pthread_cond_t          ct_cond;
```

```
        unsigned int             ct_state;
#define WRK_IDLE        0x00    /* R: worker is idle */
#define WRK_WORKING     0x01    /* R: request has been loaded */
#define WRK_DONE        0x02    /* W: I'm done, requester should go on */
#define WRK_MASK        0x03            /* mask of the above */
#define WRK_SHUTDOWN    0x04    /* S: shutdown if/when next idle */
#define WRK_DIE         0x08    /* S: kill yourself */
#define WRK_CLOSEALL    0x10    /* S: the environment needs to be
                                **    recovered, close all real handles,
                                **    return DB_RETRY */

        /* local data */
        struct request           ct_request;
        struct ct_txn            ct_txn;        /* the zero-id txn */
        time_t                   ct_time;
};
```

The `ct_state` variable is used to communicate between threads: threads lock the `ct_mutex` member, change `ct_state`, possibly change the fields inside the `ct_request` member, then *broadcast* on the `ct_cond` member (c.f. Section 5.2.4) and unlock the `ct_mutex` member. The `WRK_SHUTDOWN` flag is used to signal, without being held up by a request that's being processed, that the parent environment is being closed and that the worker thread should close all the cursors and abort all the transactions under it. If the `WRK_DIE` flag is set then the worker is being completely destroyed (not just returned to the free pool) and the worker thread, when next idle, should free the `ct_wrk` structure then call the thread exit routine. The `WRK_CLOSEALL` flag is set when some operation in this environment has returned `DB_RUN_RECOVERY` and indicates that the worker should call the underlying `DBC→close()` and `txn_abort()` functions on all the cursors and transactions associated with this worker without deleting the IDs at the same time.

The `ct_time` member holds the activity timestamp for all the transactions associated with this worker, as well as for any cursors that are inside those transactions. Transactionless cursors do not use this timestamp.

The `ct_dbcs` list is used to support the closing of databases that contain open cursors. If the `ct_dbs` list of open cursors in not empty, the `DB→close()` handler moves the open cursors from its list to the `ct_dbcs` list in the `ct_wrk` that is the parent of each cursor. The worker thread checks this list at the top of its main loop and closes any cursors that it finds on the list.

```
struct ct_db {
        id_int                   ct_id;

        /* next links */
        LIST_ENTRY(ct_db)        ct_next;

        /* head links */
        struct ct_dbc_list       ct_dbcs;

        /* parent link */
        struct ct_env           *ct_parent;

        /* synchronization */
        pthread_mutex_t          ct_mutex;
        pthread_cond_t           ct_cond;
```

```
        /* local data */
        DB                      *ct_dbp;
        time_t                   ct_time;
        unsigned int             ct_count;
        unsigned int             ct_flags;
#define RPCDB_CLOSING           0x01            /* no new uses allowed */
#define RPCDB_RECOVERING        0x02            /* temporarily unavailable */
};
```

The `ct_count` member is normally a count of the number of requests currently being processed that directly specify this `ct_db`. However, during the process of closing the database, it is overloaded as the count of open cursors. The `RPCDB_CLOSING` and `RPCDB_RECOVERING` flags mirror the corresponding flags in the `ct_env` structure.

```
struct ct_dbc {
        id_int                   ct_id;

        /* next links */
        LIST_ENTRY(ct_dbc)       ct_next_txn;
        LIST_ENTRY(ct_dbc)       ct_next;       /* db or wrk */

        /* parent links */
        struct ct_txn           *ct_parent_txn;
        struct ct_db            *ct_parent_db;

        /* local data */
        DBC                     *ct_dbcp;
        time_t                   ct_time;
        struct ct_dbc           *ct_join;
        unsigned int             ct_flags;
#define RPCDBC_NO_TXN           0x01
#define RPCDBC_RESULT_OF_JOIN   0x02
};
```

The `ct_join` member is normally `NULL`. If the cursor is passed to the `DB→join()` function then the `ct_join` member should be set to point at the resulting cursors's `ct_dbc` structure. As long as the `ct_join` member is not `NULL`, the cursor is considered 'busy' and will neither timeout nor be accessible to normal requests. The `RPCDBC_RESULT_OF_JOIN` flag is set for the cursor that was created by the `DB→join()` function. When such a cursor is closed or timed out, all the cursors whose `ct_join` member points to that cursor must have their `ct_join` member set to `NULL` at that time. This provides the proper semantics for timeouts on 'join' and 'joined' cursors. Note that since all a join cursor and its members must all be in the same transaction or all transactionless but in the same worker, finding all the member cursors is merely $O(n)$ on the number of cursors in that transaction or worker, instead of $O(n)$ on the number of member cursors.

The `RPCDBC_NO_TXN` flag is used to indicate whether the `ct_time` member of this structure is in use: if it is set, then this is a transactionless cursor which has its own activity timestamp in the `ct_time` member, while if it is reset, then the `ct_time` member of the parent worker (`this->ct_parent_txn->ct_parent->ct_time`) should be used instead.

While the `ct_next_txn` member is a normal 'next' pointer for the list of cursors attached to a given transaction, the `ct_next` member acts as a 'next' pointer for one of two different lists; see the description of the `ct_dbcs` member of the `ct_wrk` structure above for details.

Mutex Ordering

The hierarchy of mutexes is, in order of locking:

1. The `ct_mutex` member of a `ct_base_env` structure or the global `db_unopened_env_mutex`
2. The `ct_mutex` member of a `ct_env` structure
3. The `ct_mutex` member of a `ct_db` structure
4. The `ct_mutex` member of a `ct_wrk` structure

A thread may not lock a mutex earlier on the list than any mutex it already holds locked. For those times when a thread needs to move up the hierarchy, it must unlock its current mutexes, possibly after marking the involved structure as busy in some way, then acquire the higher mutex.

Structure Access Rules

The `ct_env` and `ct_db` structures must be accessible by multiple threads simultaneously. We therefore need some way of guaranteeing that such a structure doesn't get closed while another thread is using it. Such an occurrence indicates a client bug if the close was by request, but a timeout occurring just as a request comes in will legitimately (if unlikely) bring about this situation.

Our solution is to keep a count of how many requests are currently using each structure plus a flag to indicate that the structure has started the process of closing down (thus cutting off any future requests). A mutex protects these and a condition variable is used to synchronize a thread trying to perform a close with the count being lowered to zero.

A thread that wants to close a structure so protected will lock the mutex, set the flag, then wait on the condition variable until the count goes to zero. At that point it knows that no requests are currently using that structure, nor will any future requests do so.

On the other side, when a thread wants to access one of these structures, it locks the mutex and checks the flag. If a close is in process, then it unlocks the mutex and returns an error as appropriate. If not, it increments the count and unlocks the mutex. When it is done with the structure, it locks the mutex again, decrements the count, and unlocks the mutex. If it decremented the count to zero then it signals on the condition variable in case a close is pending.

While the above works just fine, it doesn't solve the problem of the entire structure disappearing from underneath an incoming request that arrives after the close had started. It protects the structure contents but not the access to the structure itself. We therefore need to guarantee for each method of getting a pointer to each structure that the obtained pointer will be valid. The key observation is that we need to 'chain' the above access lock onto what lock protects the source of the pointer. When the pointer is obtained by walking a list of these structures, the thread must obtain the access lock while still holding the mutex on the list. When the pointer is obtained from the ID database, the thread must obtain the access lock while the ID subsystem is locking out delete operations. Finally, if the pointer is another structure's 'parent' pointer, the thread must mark the child structure as 'busy'. This has usually already been done as part of the processing of the request, but an exception exists while processing the `DB→cursor()` request: when creating a transactionless cursor, the thread must explicitly obtain an access lock on the parent `ct_env` to guarantee that the new `ct_wrk` structure is linked in before a simultaneously occurring environment timeout can progress.

Structure Close procedures

In order to maintain the proper invariants in the data structures described above, particular procedures must be followed when closing the structures, whether by request or as the result of a timeout. While the precise variants have yet to be documented, the following procedures are believed to be "safe".

Closing Environments When an environment is closed or removed, the following steps are performed:

1. `id_delete_wrk()` is called. The prevents new requests from using this environment directly.
2. The close routine waits until any 'in process' requests that directly use this environment are complete. This is done by waiting on the `ct_cond` member until the `ct_count` goes to zero.
3. If there are any workers associated with the environment, the close routine tells each one to shutdown (`WRK_SHUTDOWN`), then waits for the `ct_wrks` list to empty by waiting on the `ct_cond` member until the `ct_wrks` member is empty. Each worker closes all its transactions and cursors and then removes itself from the `ct_wrks` list in the `ct_env`.
4. If there are any databases open inside the environment, the close routine closes them. Databases that were already in the process of closing will be skipped here.
5. If any databases are still in the list, they were already being closed before we got to them, so the close routine needs to wait for them to finish by waiting on the `ct_cond` member until the `ct_dbs` member is empty.
6. The actual `DBENV→close()` or `DBENV→remove()` routine is called.
7. The `ct_env` structure is pulled from the global list and put on the free list

Idle timeouts can be handled by the same procedure, with the exception that the presence of 'in process' requests, workers, or 'recently' active databases beneath the `ct_wrk` should abort the timeout.

Closing Databases Closing a database is complicated by the requirement that doing so closes all the cursors that are open in it. In the non-RPC case there is a constraint that all the cursors were created by the thread doing the `DB→close()`, however the RPC server doesn't have enough information to enforce or take advantage of that constraint, so it must be able to handle the general case of a database close being requested while child cursors belonging to multiple workers are open in the database. We therefore provide a means for the routine that handles `DB→close()` to request the closing of cursors inside that database. This is done using the `ct_dbcs` member of the `ct_wrk` structure, as mentioned above. When a `DB→close()` request is received, the following steps are performed:

1. `id_delete_db()` is called. This prevents new requests from using the database.
2. The close routine waits until any 'in process' requests that directly use this database are complete. This is done by waiting on the `ct_cond` member until the `ct_count` goes to zero.
3. The close routine saves a copy of the `ct_dbp` member of the `ct_db` structure and sets the member to `NULL`.
4. The close routine then moves each `ct_dbc` structure in the `ct_db`'s `ct_dbcs` list to the list headed by the `ct_dbcs` member of the cursor's associated `ct_wrk`. Once all the cursors for a given worker have been transferred between the lists, the close routine broadcasts on the `ct_wrk`'s condition variable. It also stores the total count of open cursors in the `ct_db`'s `ct_count` member. Note that the `ct_dbcs` member of the `ct_wrk` structure is protected by the `ct_mutex` member of the same `ct_wrk` structure.
5. The close routine then waits for all the cursors to be closed, by waiting on the `ct_cond` member until the `ct_count` goes to zero.

6. The actual DB→close() routine is called.
7. The ct_db structure is removed from the parent ct_env's ct_dbs list and moved to the free list. If the parent's ct_dbs list is now empty, signal on the parent's ct_cond member.

The same routine handles DB→remove() and DB→rename() requests.

Closing Cursors Closing a cursor proceeds with the following steps:

1. id_delete_dbc() is called.
2. Remove the ct_dbc from its parent ct_txn's ct_dbcs list
3. Do the actual DBC→c_close()
4. Lock the parent ct_db's ct_mutex member.
5. If the RPCDB_CLOSING flag is not set in the parent ct_db's ct_flags member then the cursor is in the list beneath the parent ct_db's ct_dbcs member and it can simply be removed from there. At that point the mutex that was locked above can be unlocked, the ct_dbc returned to the free pool, and the close is complete.
6. Otherwise, the parent database is in the process of being closed and this cursor is instead in the list underneath its worker's ct_wrk's ct_dbcs member.
7. Decrement the cursor count found in the parent ct_db's ct_count member.
8. If the count went to zero, signal on the parent ct_db's ct_cond member.
9. Unlock the parent ct_db's ct_mutex from above.
10. Lock the worker's ct_wrk's ct_mutex member
11. Remove the ct_dbc from the worker's ct_wrk's ct_dbcs list
12. Lock the worker's ct_wrk's ct_mutex member and return the ct_dbc to the free pool.
13. If this was a transactionless cursor, then lock the ct_mutex member of the environment that this is in and update its ct_time active timestamp with the ct_time active timestamp for this cursor.
14. If this worker is now empty, remove it from the ct_env's ct_wrks list and move it to the free list.

Closing Transactions Closing a transaction is the simplest of the lot:

1. id_delete_txn() is called.
2. Child transactions of the transaction being committed or aborted are recursively committed or aborted. If any of those fail with error DB_LOCK_DEADLOCK then the rest of them can be aborted and the top-level operation is forced to be an abort instead of a commit.
3. Do the actual txn_commit() or txn_abort().
4. Remove the ct_txn from the list under the parent ct_txn's ct_sub_txns member and add it to the free list.

Auto-recovery procedures

While the RPC server provided by Sleepycat with version 3.1.17 performs recovery on all environments when it first starts up, it has no special handling of recovery after that. In particular, if an operation returns `DB_RUN_RECOVERY`, the client has to close the environment and reopen it with the `DB_RECOVER` flag. Unfortunately, this doesn't work if more than one client has the environment open. Recovery must be single-threaded: running recovery while another thread or process has the environment open can unrecoverably corrupt the environment. Given that we expect a Boardwalk system to have each environment open from each Access Server, we need a way to mitigate the single-threaded recovery requirement.

The overall plan is to unify the environment handles in the server, such that each environment is only opened once by the server, regardless of how many clients open the environment. If an operation returns `DB_RUN_RECOVERY`, all the transactions and cursors that are open in that environment will be closed *without* deleting either the IDs associated with them from the ID database or their `ct_foo` structures. Once all the underlying cursors have been closed and transaction have been aborted, the underlying databases will be closed and the environment will be closed and recovered. After recovery has completed, the previously open databases will be reopened. Meanwhile, all requests except `DBC`→`close()` and `txn_abort()` that attempt to use the cursors and transaction that have been closed above will receive the (new) `DB_RETRY` status.

The result of the above should be that stateless items, the environment and its databases, appear unchanged to the client—their IDs are preserved across recovery—while stateful items, transactions and cursors, have to be closed and built back up by the client.

In order to implement this, when a operations returns `DB_RUN_RECOVERY`, the thread which received it will attempt to set the `RPCBASE_RECOVERING` flag in the `ct_flags` member of the `ct_base_env` structure for the involved environment. If that flag was not already set, then the thread will create a new thread to perform auto-recovery. That thread will set the `RPCENV_RECOVERING` flag in each `ct_env` structure and the `WRK_CLOSEALL` flag in each worker to tell it to close its handles. It'll store the total number of workers so flagged in the `ct_count` member of the `ct_base_env` structure and then wait on the `ct_cond` member for the count to go to zero. When a worker finishes closing its handles, it decrements the `ct_count` member, signaling on the `ct_cond` member if the count is now zero. Once the count is zero, the auto-recovery thread will close the databases and environment and perform recovery. If recovery succeeds, then it'll reopen all the databases that were previously open and update the `ct_dbp` members of the `ct_db` structures and clear the 'in recovery' flags set above. If recovery fails, the server will log the error and shutdown completely.

The handling of transaction ID wraparound is mostly similar to auto-recovery except that instead (or in addition) to performing recovery, the environment is removed and recreated. This resets the transaction ID counter. If additional magic is necessary for this, it'll be handled by the per-environment thread.

ID subsystem

Requests received by the RPC server reference previously created structures using the ID number returned by the server when the structure was created. The server needs methods for allocating and deallocating unique IDs and for mapping them to the appropriate C structures.

Since either the lookup of a current ID or the allocation of a new ID is a necessary condition for access to a structure involved in servicing a request, these functions are an ideal point for updating the 'active' timestamps that keep structures from being timed out.

Finally, the lookup functions must guarantee that if an ID was successfully mapped to a structure, then a 'read lock' has been taken out on the structure such that it cannot be freed until the thread that did the lookup has released its lock.

All the following C functions return zero on success and a DB-style error number on failure.

```
int id_init(u_int32_t flags, const char *dir);
void id_close(void);
```

These functions respectively initialize and shutdown the ID subsystem. The `flags` argument to `id_init()` is documented below.

```
int id_new_env(struct ct_env *, time_t, id_int *);
int id_new_db(struct ct_db *, time_t, id_int *);
int id_new_txn(struct ct_txn *, time_t, id_int *);
int id_new_dbc(struct ct_dbc *, time_t, id_int *);
```

Given a pointer to a structure and the current time, these functions allocate an ID for the structure, update the active timestamp as appropriate for the structure, and return the new ID via the last argument.

```
int id_lookup_env(id_int, time_t, struct ct_env **);
int id_lookup_db(id_int, time_t, struct ct_db **);
int id_lookup_txn(id_int, time_t, struct ct_txn **, struct ct_wrk **);
int id_lookup_dbc(id_int, time_t, struct ct_dbc **, struct ct_wrk **);
```

These functions map the given ID to a C pointer, take a read lock on the structure, update the active timestamp with the given time, and return the pointer via the third argument. The fourth argument to `id_lookup_txn()` and `id_lookup_dbc()` are used to return the pointer to the `ct_wrk` structure associated with the `ct_txn` or `ct_dbc` that was looked up so that it may be used for an inter-thread call. If the ID of a cursor that is involved in a DB→`join()` is looked up, `id_lookup_dbc()` will return `EINVAL`.

```
int id_lookup_dbc_many(unsigned int, const id_int *, time_t,
                       struct ct_dbc **, struct ct_wrk **);
```

This function supports the DB→`join()` request. The first argument is a count of how many cursors are involved in the DB→`join()`. The IDs of those cursors are passed in the second argument as an array. The fourth argument must point to an previously allocated array into which the mapped C pointers can be stored. This functions performs additional checking to guarantee that the specified IDs all belong to a single worker thread and that none of the cursors are currently involved in a DB→`join()`, returning `EINVAL` in either of those cases.

```
int id_delete_env(struct ct_env *);
int id_delete_db(struct ct_db *);
int id_delete_txn(struct ct_txn *);
int id_delete_dbc(struct ct_dbc *);
```

These functions remove from the ID subsystem the mapping to the given structure. These functions take as an argument the structure involved instead of its ID as a convenience: the former is always on hand when a mapping is being removed, while the ID is not obviously available while performing a timeout.

On success, the `id_lookup_txn()`, `id_lookup_dbc()`, and `id_lookup_dbc_many()` functions all return with the `ct_mutex` member of the associated `ct_wrk` structure locked. This lets them guarantee that the transaction or cursor will not disappear underneath the request. The `id_lookup_env()` and `id_lookup_db()` functions make a similar guarantee by incrementing the `ct_count` member of the structure being looked up while under the protection of the `ct_mutex` member.

In our current implementation, the ID subsystem uses an in-memory Sleepycat database, run as a Concurrent Data Store to provide many-read/single-writer semantics. The `flags` argument to the `id_init()` function are a bitwise OR of zero or more of the following DB flags to be used when creating the DB environment for the ID database: `DB_USE_ENVIRON`, `DB_USE_ENVIRON_ROOT`, `DB_LOCKDOWN`. The `dir` argument, when non-`NULL`, specifies the path of the directory to be passed to `DBENV`→`open()`. Additionally, the ID database itself will be created as an on-disk database in that directory, allowing debugging and profiling with the standard DB utilities.

The database keys are the ID numbers in 32-bit big-endian format and the values are the pointers to the associated C structure plus additional information to provide consistency checking, currently just the type of the structure to which the ID maps. IDs are allocated sequentially, starting at one (recall that zero is reserved for representing the `NULL` pointer). If the ID is already present in the database—the ID space has wrapped around 2^{32} and a long lived ID has been encountered—then the allocation function simply keeps incrementing until it finds a free ID. The ID allocation functions not only return the new ID through their third argument, but also store it in the structure itself, for later use by the deallocation functions. This also provides a consistency check when a lookup is performed.

The keys are stored in big-endian byte-order so that they sort in ID order. The hope is that by doing so and using a BTREE database, the insert operation should be efficient. Alternatively, the keys could be left in native byte-order and a comparison function could be specified. Or, performance testing could reveal that this just doesn't matter. The current code uses native byte-order for the keys. If there's any chance byte-order would be different on the Access vs. Metadata Servers, this should be changed.

Locking Calls

The RPC server distributed by Sleepycat supports only a subset of the standard DB library functions, including most of the environment, database and cursors calls. The current version of the Boardwalk architecture requires support for additional functions, including all of the 'lock' family of calls: `lock_detect()`, `lock_get()`, `DBENV`→`set_lk_conflicts()`, `DBENV`→`set_lk_detect()`, `DBENV`→`set_lk_max()`, `lock_id()`, `lock_put()`, `lock_stat()`, and `lock_vec()`. Supporting these will require that certain additional data structures be represents over the RPC protocol, including the `DB_LOCK` and `DB_LOCKREQ` structures.

The `DB_LOCK` structure is opaque to the user and can therefore be passed directly over the wire, though this will require the server and client to have the same sizes for it. Encoding these as IDs seems simple, but requires the IDs to be stable through the closing of the environment and even past server restart. An on-disk lock ID database would have to be stored with the environment, which gets really complicated really fast. Given the relatively small fixed size of the `DB_LOCK` structure, direct representation seems preferable at this time.

The `DB_LOCKREQ` structure is more complicated to represent in the RPC protocol. If the `op` member is `DB_LOCK_GET` then the `mode` and `obj` members must be passed from client to server, and the `lock` member (or the ID representing it, to be precise) must be returned in the reply. If the `op` member is `DB_LOCK_PUT` then the `DB_LOCK` structure must be passed from client to server, while nothing needs to be returned. If the `op` member is `DB_LOCK_PUT_OBJ` then the `obj` member must be passed from client to server with nothing needed in the reply, while if it is `DB_LOCK_PUT_ALL` then absolutely nothing additional is needed to be passed either direction. I therefore propose that the RPC encoding of the `lock_vec()` arguments use a variable length array of a discriminated union. The resulting request structures would end up being represented in the RPC language parsed by rpcgen with the following:

```
struct __LOCKREQ_get {
        unsigned int mode;
        opaque obj<>;
```

```
};

union __LOCKREQarray switch (int op) {
        case DB_LOCK_GET:
                __LOCKREQ_get get;
        case DB_LOCK_PUT:
                opaque lock<>;
        case DB_LOCK_OBJ:
                opaque obj<>;
        default:
                void;
};

struct __lock_vec_msg {
        unsigned int dbenvcl_id;
        unsigned int locker;
        unsigned int flags;
        __LOCKREQarray list<>
};
```

On the reply side, the `elistp` argument will be returned from the server as an offset from the start of the list of requests. If there are any `DB_LOCK_GET` requests, the IDs of the granted locks will be returned as a list. Note that if the request is only partially successful, the list of lock IDs returned might not cover all the `DB_LOCK_GET` requests that were sent; the client should zero out the `DB_LOCK` member of any `DB_LOCKREQ` structures in the request list beyond the last successfully processed one, as determined from the returned `elist_offset`.

```
struct __LOCKreparray {
        opaque ent<>;
};

struct __lock_vec_reply {
        int status;
        unsigned int elist_offset;
        __LOCKreparray locks<>;
};
```

Control Flow

The threads inside the RPC server can be broken into four groups: request threads, worker threads, per-environment threads, and server-wide threads. Each group has its own control flow.

Request threads Request threads are created by the multithreaded `svc_run()` routine. They follow the same RPC request processing flow of control as seen in the single-threaded server:

`db_serverproc_1()` `db_server_svc.c` This is generated by rpcgen and performs the XDR decoding, calls the request-specific handler, calls `svc_sendreply()` to XDR encode and send the reply, and then frees any space allocated by the request and the reply.

`__db_foo_1_svc()` (e.g., `__db_txn_begin_1_svc()`) `gen_db_server.c` This simple glue function packs list arguments in the request into `NULL` terminated arrays and transforms the single-argument structure into multiple arguments for the next function. It's automatically generated by `gen_rpc.awk`.

`__foo_1_proc()` **(e.g.,** `__txn_begin_1_proc()`**)** `db_server_proc.c` This function maps the IDs to pointers and then either calls the actual DB routine or makes an inter-thread call to the correct worker thread. If a new worker is needed, this routine requests one from the allocation code.

Worker threads Worker threads are created by the allocation code, as invoked from a request thread. Worker threads have four things to do:

1. Handle inter-thread requests by calling through a function pointer passed from a request thread
2. Timeout child transactions and cursors
3. Close child cursors on the behalf of the `DB→close()` handler
4. Close child cursors and abort child transactions on the behalf of `DBENV→close()` and server shutdown handlers

Per-Environment threads Each actual DB environment is associated with a `ct_base_env` structure and a thread. That thread does two things: implement the idle timeout for the 'instances' of that environment (the `ct_env` structures associated with it) and perform auto-recovery. It spends most of its time waiting on the `ct_cond` member of the `ct_base_env` structure for either a timeout (in which case it checks for idle environment instances) or a signal indicating that a flag has been set and some special process is needed.

Server-wide threads Two server-wide activities have dedicated threads: the RPC select/poll loop and the idle unopened environment timeout checker. The former executes the multithreaded `svc_run()` routine and thereby spawns all the request threads. The latter wakes up once a day and checks the unopened environment list for environment that have been idle too long.

Condition Variable Usage

Whenever a thread needs to wait for some abstract state to be changed by another thread, a condition variable and mutex are needed to provide synchronization. In order to avoid 'lost wakeups' and deadlocks, the following four rules apply to all condition variable uses:

1. A thread may not wait on a condition variable unless it has, since it last locked the mutex, checked *all* of the state it is interested in.
2. Since spurious wakeups are allowed by the POSIX standard, waiting on a condition variable must always be wrapped in a loop that checks to see if the state has really changed.
3. If not all of the threads waiting on a condition variable are 'interested' in a particular change of state, then the thread that changed the state must broadcast on the condition variable instead of signaling on it.
4. A thread must not signal or broadcast on a condition variable unless it has held the associated mutex since it changed the abstract state. (Taken in a slightly more restrictive reading, this implies that a single mutex should protect *all* of the abstract state for that condition variable.)

Given those rules, the basic procedure for waiting for the state to change looks like:

```
lock mutex
while (state != what the thread is interested in)
        cond_wait(cond, mutex)
do whatever, including possibly changing the state
if this thread changed the state, signal or broadcast here...
unlock mutex
...or maybe here
```

Memory Management

Memory management is expected to be fairly simple. RPC and XDR structures are allocated and freed by the RPC code as the requests are received and replied to by the request threads. One complication exists and is already dealt with by the RPC server code: a few DB routines may return the same byte-string as was passed to them. To keep XDR from freeing the same memory area twice, these byte-strings must be duplicated so that the request and reply structures do not share pointers.

In non-multithreaded RPC server code, it is traditional to statically allocate the reply structure. That obviously doesn't work in multi-threaded code, so the rpcgen program, when invoked with the `-M` flag, makes the reply structure an automatic variable in the `serverproc()` routine, `db_serverproc_1()` and calls a new hook, `db_serverprog_1_freeresult()`, to free the reply structure after the reply is sent.

For the structures that appear in Figure 5.1, simple free pools. Actual worker thread creation and destruction will be tied to the allocation and deallocation of the `ct_wrk` structures. Worker threads associated with `ct_wrk` structures in the free pool will just sit waiting for an inter-thread call or a "kill yourself" request.

Timeouts and Shutdown

Worker threads keep track of and perform timeouts for their associated transactions and cursors. To this end, the worker thread keeps track of the soonest time a transaction or transactionless cursors associated with the worker can timeout. Whenever it goes to wait on its condition variable, if that timeout exists (workers waiting on the free list have no children), then it passes that value to the call to `pthread_cond_timedwait()`. If the worker has no 'soonest timeout', then it can just call `pthread_cond_wait()`. If the previous call to `pthread_cond_timedwait()` returned `ETIMEDOUT`, then before waiting on its condition variable again the worker checks its children transactions and transactionless cursors to see if any really have been inactive that long, and closes any that have. At the same time, it updates its 'soonest timeout' value to reflect the current state.

The idle timeout on open environments is enforced by the thread for the associated `ct_base_env`. Another thread is responsible for timing out environments that have been created but not opened.

Configuration

Previous versions of the RPC server have not had a configuration file, with all configuration taking place via command line options. In this version, a standard Sleepycat environment configuration file, `DB_CONFIG`, will be used to store configuration data for the RPC server itself, though the command line options will also be supported. To this end, a hook will be added to the DB library internal `__db_parse()` routine to call-out into the RPC server code when, while parsing a `DB_CONFIG` file, it encounters a configuration option that starts with `rpc_`. Server-wide configuration information will be obtained from the environment created to store the ID mapping, while certain options may be overridden by an exported environment's own `DB_CONFIG`.

The currently planned RPC specific options and their corresponding command line option are:

`rpc_server_port -P` The TCP port on which the server should listen. If zero, the server lets the RPC system select the port and register it with the portmapper. If not zero, the server will bind to that port and not register with the portmapper.

`rpc_server_logfile -L` The path to the RPC server's logfile.

`rpc_environment_dir -h` The path to a directory that is to be available to clients. The last component of the path will be used be clients to specify this environment. This option may occur more than once.

`rpc_env_idle_timeout -I` The idle timeout for environments. This value may be overridden in an environment's own `DB_CONFIG` file.

`rpc_def_cursor_txn_idle_timeout -t` The default timeout for transactions and cursors. This value is used whenever the create environment request specifies a timeout of zero. The server-wide value may be overridden in an environment's own `DB_CONFIG` file.

`rpc_max_cursor_txn_idle_timeout -T` The maximum timeout for transactions and cursors. If a create environment request specifies a timeout greater than this value, then this value is used instead. The server-wide value may be overridden in an environment's own `DB_CONFIG` file.

The path to the server-wide configuration environment will be specified using the `-D` option. Other debugging options will only be available via the command line. These include `-d` (enable debugging output), `-R` *`filename`* (enable and specify RPC packet log), `-V` (display Sleepycat library version and exit), and `-v` (enable verbose startup output).

5.2.5 Unresolved Issues

What error should the server return if a request specifies entities that are in different environments?

What error should the server return if a request specifies a cursor involved in a join? What error should it return if a request specifies cursors in different workers?

What should the idle timeout thread do if it detects a 'hung' request (i.e., the active timestamp is old but the count is non-zero)?

5.2.6 Problems

There are couple problems with this design. The most subtle is that because the server uses a new worker thread each time a new transactionless cursor is created with the `DB→cursor()` request, it is in general not possible to use `DB→join()` with transactionless cursors: the constituent cursors will have been created in different threads and therefore be unjoinable. Solving this without breaking other applications would require the server to not always use a new worker thread for each transactionless cursor. Note that if two transactionless cursors are created by two different threads in the client, the server must use different workers threads for those two cursors, otherwise it may cause deadlock in a correctly threaded application. The only way to safely share a worker thread between transactionless cursors is for the client to pass a 'client-side thread id' to the server with the `DB→cursor()` request.

Note that if a transactionless cursor is duplicated (`DBC→c_dup()`) then the duplicate will be associated with the same worker thread as the original. Indeed, it is already possible to use `DB→join()` with such duplicates, but that's a very limited set of possibilities.

The only other means for removing this restriction that I can think of would be to modify the DB 'core' itself to weaken the thread constraint (5.2.2) to allow cross-thread use of cursors by `DB→join()`, or more precisely, in `__db_join_get()`, the function that implements the `c_get` method for join cursors.

Since the some applications don't use `DB→join()` at all, this limitation is acceptable to the Boardwalk project.

Other situations exist where this design is lenient to threading errors. In the current version of Sleepycat DB (3.1.17), the `DB→close()` call may be made when cursors are open in that database, in which case they will be closed before the database is closed. In a multithreaded application, this is only legal if all the cursors involved were opened in the thread calling `DB→close()`. Since the RPC server has no way of checking that, it will 'safely' perform all the cursor closings, even when they were opened in different threads on the client side. Detecting such an error in the server would require not only the above expansion of the `DB→cursor()` call in the RPC protocol, but a similar change to `txn_begin()` and data structure changes to match.

Note that while a multi-threaded RPC server should be able to handle a single threaded client, the reverse is not true: deadlock may trivially occur. Single-threaded versions of the server should therefore return an error when `db_env_create()` is invoked with the `DB_THREAD` flag.

Chapter 6

Command Server

The Command Server is designed to be a single control point for implementing state change (policy) decisions across the entire DI/ODE system as well as a nexus for system monitoring and human interaction.

The term "Command Server" names both the machine and the server software running on it. It should be clear from context which is which.

6.1 Configuration

While DI/ODE system components may cache local copies of their control information in configuration files, the master copies always reside on the Command Server. After these files are changed on the Command Server, a distribution system will push them out to the DI/ODE components.

Files will be distributed to the Access and Metadata Servers using the SSH protocol. The Access and Metadata Servers will run the SSH daemon, `sshd` as a stand-alone service.

The `/` directory of each Data Server will be mounted on `/etc/diode/ds/`*dsname*`/`. Therefore, updating files on each Data Server will be done by copying them from their location on the Command Server to their destination on the Data Server's file system. Remote execution on the Data Server will be performed using the `rsh` command. Unfortunately, a more secure mechanism cannot be used as these are not supported by the Network Appliance Data Servers.

The Command Server stores files for other servers in a local directory hierarchy. Under the directory `/etc/diode` on the Command Server are directories for each of the additional server types: `as`, `ds`, and `mds`. Underneath these directories will be one directory per server, for example, `as001`, `as002`,... under the `as` directory, `mds001`, `mds002`,... under the `mds` directory, etc..

6.1.1 Access Server Information

Within each Access Server's own directory (e.g., `/etc/diode/as/as001`) several files may be present:

`up` If this file is present, it indicates that this Access Server is ready for service.

`roles` This file lists the daemons which should be run on the Access Server, one per line, for example:

```
app1
app2
```

`logs` This file contains a list of log files (with absolute paths) to be retrieved from the Access Server, one per line.

Also in each Access Server's directory is the `conf` subdirectory, which contains the master copies of all DI/ODE configuration files for that Access Server, to wit, the entire contents of the Access Server's `/etc/diode` directory. These files are:

`filesock.conf` This file contains a list, one per line, of socket names on which Client Daemons will be listening for connections from Client Libraries. Each socket name will consist of a protocol family specifier and protocol family specific addressing, separated by a colon. We will initially support the "INET" and "UNIX" protocol families. For "INET" sockets, the address will be a TCP port number followed by an optional "`@`" and hostname or IP address. If the host part is not given, the address of the loopback interface is used. For "UNIX" sockets, the address will be a file path. The following are some legal examples:

```
INET:12345@localhost
INET:12345@10.0.0.1
INET:12345
UNIX:/var/run/diode.file.1
```

The number of entries in this file will implicitly provide the number of file-related Client Daemons which will run on the host. On startup, a Client Daemon will read through the file trying to `bind()` to the INET or UNIX socket. If either fails with `EADDRINUSE`, then it will proceed down the list until it finds one that is available or it falls off the end of the list.

`dbsock.conf` This is the Sleepycat analog to the `filesock.conf` file. This file contains a list of socket names, one per line, on which Client Daemons will be listening for connections from the Sleepycat DB library for RPC service. The same restrictions on entries in the `filesock.conf` file apply to `dbsock.conf` entries.

A Client Daemon will be invoked with a command-line flag to start either the file service or the DB service. Initialization code will read the appropriate *service*`sock.conf` file.

`paths.conf` This file contains a list of the virtual mount point which the Client Library will use to determine which paths in the file namespace will be intercepted and redirected to the Client Daemon. Everything occurring before the double-slash "`//`" gives the virtual local mount point between local and remote storage. The part of the path after the double-slash is a template showing which components are used in the hash computation by the Client Daemon to determine which Data Server hosts the data and which are ignored. Path components labeled "`%h`" are included in the hash, while those labeled "`%i`" are not. For example:

```
/var/spool/lpd//%h
/data/web//%i/%i/%h
```

Note, that `%i` directories that precede one or more `%h` directories must exist on all Data Servers. In this example, if a filename to be accessed is `/data/web/a0/d3/131/foo.html`, we'd know since this file exists under the `/data/web` directory, it belongs on the Data Servers, and that the Client Daemon should use "131" as the argument to its hashing function to determine which Data Server stores this particular data. Further, the `a0/d3` directory tree should already exist on each Data Server.

`metakeys.conf` This file describes how the Client Daemon hashes a database key to select the Metadata Server for that key. Each line has four fields, specifying the name of the environment, the name of the database, and an encoded description of how the key is formatted, and an expression of which parts of the key are hashed. For example:

```
database  company.db                        qq    2
database  department.db                     qq    1
database  employee.db                       q     1
database  project.db                        qqq   3
```

`mount.conf` This file lists the Data Servers and the mount points that they provide, including information for calculating the consistent hash. Its format will list one Data Server per line with five whitespace separated fields. The fields are: Date Server hostname, hash bin number, local "mount point", Data Server export path, and Data Protocol. For example:

```
ds1    1      /var/spool/lpd              /data/lpd/as1        nfs3:udp
ds2    2      /var/spool/lpd              /data/lpd/old        nfs3:udp
ds1    1      /data/web                   /data/web            nfs3:udp
ds2    2      /data/web                   /data/web            nfs3:udp
```

Column 1, the Data Server hostname is just that, the hostname of one particular Data Server. When a consistent hash bin set is calculated for each Data or Metadata Server, each of these is assigned a number to create a unique ordering. Because this is distinct from the Data Server name, we can migrate away from any malfunctioning Data Server by associating its consistent hash bin set with a new server. The third column contains the virtual local mount point. Each entry here should appear before a double-slash in the `paths.conf` file. The absolute path name of the files stored on the Data Server are formed by appending the portion of the file path below the mount point to the absolute path given in the fourth column. The fourth column by itself represents the export point from the Data Server. The fifth column contains the Data Protocol and transport to be used. It is there for convenience, as we intend to support only NFSv3 over UDP for Boardwalk.

`metamount.conf` The `metamount.conf` file serves a similar function for the Metadata Servers although its syntax is simpler. Each line in the file represents a logical Metadata Server with the first column containing the Metadata Server's name, the second column contains the consistent hash bin number, and the third column contains the TCP port number on which the Metadata Server Daemon is listening. A port of zero indicates that the portmapper should be queried to determine the correct port. For example:

```
mds001       1        1286
mds002       2        1287
```

For migration, we need a "before" and "after" view of the Data and/or Metadata Server set. The "before" set is defined by the `mount.conf` and `metamount.conf` files. The "after" set is defined by the `mount.conf.migrate` and `metamount.conf.migrate` files. The differences between the two files define the migration.

6.1.2 Data Server Information

Under the `ds` directory there is, logically enough, one subdirectory for each Data Server named using the hostname of that Data Server (e.g. ds001, ds002, etc.). Under that directory is a subdirectory called `conf` where the files live that are copied into `/etc` on each of the Data Servers. The files that reside here include: `hosts`, `rc`, `exports`, and `syslog.conf`.

Also in each Data Server's directory is a file named `up` if that Data Server should be an active part of the DI/ODE system. Note that the existence of this file does not mean that the system would be actively used. If a new Data Server is brought online and this file is present it means that this Data Server is now ready to be added to active service via the Migration process.

6.1.3 Metadata Server Information

Under the `mds` directory is one subdirectory per each Metadata Server, each bearing the hostname of one of the Metadata Servers. In each of these subdirectories (for example, `/etc/diode/mds/mds001`) may be a file, `up`, which exists if the Metadata Server is configured and ready for service. Like the Data Server case, the existence of an `up` file does not necessarily mean that a Metadata Server is in service. It is either in service or eligible to be put in service via a Migration process.

In the `/etc/diode/mds/`*servername*`/conf` directory is a file, `DB_CONFIG`. It contains a list of configuration directives for the Sleepycat DB RPC server. The directory path (not the file path) on the Metadata Server in which this file is installed will be passed to the Metadata Server Daemon via the `-D` option. An example config file would be:

```
## Default is no logfile
rpc_server_logfile /data/web/server.log
## Default is to obtain a port dynamically and register with
## the portmapper.  Specifying a port also disables registering
## with the portmapper.
#rpc_server_port 12345

## Timeouts are in seconds.
rpc_env_idle_timeout 3600
#rpc_def_cursor_txn_idle_timeout 20
rpc_max_cursor_txn_idle_timeout 30

## What environments are exported?
rpc_environment_dir /data/web/sessions
rpc_environment_dir /data/web/authorization
rpc_environment_dir /data/web/content-index
```

Note that this file is an extended form of the normal `DB_CONFIG` file supported by Sleepycat for environment configuration.

Each environment that is to be exported by a Metadata Server Daemon may also have its own standard `DB_CONFIG` file containing directives that are particular to that environment. On the Command Server, these files will be stored in `/etc/diode/mds/`*servername*`/conf/`*environment*`-DB_CONFIG`.

6.2 Command Line Utilities

The Command Server will have command-line utilities for performing the following sets of functions.

There will be commands to control Access Servers. Whenever the command takes a *hostname* option, the special hostname "ALL" may be given. It does what one would expect.

`asadd` *hostname* Add a new Access Server to the system, initially in the *down* state.

`asstat` *[hostname...]* Give the status of the named Access Servers, or all Access Servers if none are named.

`asup` *hostname* Set the state of the Access Server to *up*.

`asdown` *hostname* Set the state of the Access Server to *down*.

`asgetlog` *hostname* Retrieve the log files off of the Access Server.

`serviceup app1|app2|ALL` *hostname* Startup of Internet service daemons, via remote execution of a script on an Access Server. If the "ALL" option is used, all the daemons the Command Server has configured that system to start will be run.

`servicedown app1|app2|ALL` *hostname* Shutdown of Internet service daemons, via remote execution of a script on an Access Server. If the "ALL" option is used, all diode-ified service daemons will be shut down on that server.

`pushconfig` *hostname* Pushes a copy of the master copies of application configuration files from the Command Server to an Access Server, overwriting any files which may be present.

`diffconfig` *hostname* Show any differences between the master copies of application configuration files on the Command Server versus those actually installed on an Access Server.

There will be a command to control distribution of objects across back-end servers, to control the migration process for changing this distribution, and commands to report what would happen in a proposed migration. This will all be done with the `migrate` command, whose usages are described below.

`migratecheck ds|mds` *filename* In normal DI/ODE operation, the `/etc/diode/mount.conf` and `/etc/diode/metamount.conf` files list the names of the operational Data and Metadata Servers, their hash-bin number (for use in the consistent hashing algorithm), and, for Data Servers, the exported directory on that machine. For a migration, a `/etc/diode/mount.conf.migrate` file (or `metamount.conf.migrate` file for Metadata Servers) is created. The filename to be checked is given as an argument on the command line, and it and its `.migrate` file are interpreted and the command prints out what data transitions will be made in order to accomplish this migration. This output can be checked to make sure that the appropriate `.migrate` file meets the administrator's goals. The `.migrate` file is generated by hand on the Command Server.

`migrate ds|mds propagate|begin|continue|doit` This command is used to propagate the appropriate `.migrate` files, transition to phase 1, transition to phase 2, or just do it all.

`migrate status` Show whether a migration is in progress, what phase it is in, and whether the sweeper is active or not.

`migrate sweeper enable/disable` *hostname* The migration Sweeper process may be temporarily disabled to reduce the I/O load on the system. It must eventually be re-enabled to complete the migration.

`datamap` *path* Gives the Data Server and physical directory or file corresponding to the given path, in the application's namespace.

`metamap` *database key* Gives the Metadata Server which holds the given datum.

There will be commands to administer the back-end machines:

`dsadd` *index* Add the Data Server into the Command Server's state, in the *down* state.

`mdsadd` *index* Add the Metadata Server into the Command Server's state, in the *down* state.

`dsup` *index* Tell all Access Servers that a given Data Server is up.

`dsdown` *index* Tell all Access Servers that a given Data Server is down.

`mdsup` *index* Tell all Access Servers that a given Metadata Server is up.

`mdsdown` *index* Tell all Access Servers that a given Metadata Server is down.

`dsmount` Soft-mount the root ("/") directories of all Data Servers, using the Command Server's kernel NFS client.

`dsunmount` Unmount the above directories.

`dsstaydown` *index* Tell the Data Server to disable NFS services, even if it reboots.

`mdsstaydown` *index* Tell the Metadata Server to not start the Metadata Server Daemon, even if it reboots.

These commands will be used for general administration of all nodes:

`diodecontrol start|stop` Global (re-)start or shutdown of the entire DI/ODE system.

`diodestat` *hostname* Retrieve operational statistics from the named server.

`diodeupgrade` *class version hostname* Software upgrades/downgrades of other machines.

Lastly, there will be commands for the control of the Command Server itself:

`cscontrol start|stop` Command Server Daemon control.

`csbackup` *device* Perform an online backup of important Command Server data.

6.3 Interactive Command Vocabulary

Most of the above command-line utilities will be short shell scripts which, as appropriate, connect to a well-known TCP port or UNIX-domain socket on the Command Server (machine), which is served by the Command Server (process). Other utilities, namely `dsmount`, `dsunmount`, `cscontrol`, and `csbackup`, will be shell scripts which don't connect to the Command Server (process).

6.4 Monitoring

Some statistics about the performance of individual nodes will be available to the Command Server. What we choose to gather in the initial release will be fairly limited, nonetheless, it should provide a pretty good view of what's going on in the system. We divide the monitoring description up via host type.

6.4.1 Data Servers

The monitoring information we can obtain from Data Servers is limited by the command set we can execute on these machines. Basically, it's limited to the output of two commands.

`sysstat` The `sysstat` command takes a single argument, the amount of time (in seconds) spent aggregating data before the results are printed to standard output. It runs until interrupted on a per Data Server basis.

`nfsstat -h` When `nfsstat` is executed with the `-h` flag, it prints out information on the NFS v2 and v3 calls received by the server on a per NFS client basis.

These commands are executed on the Data Server directly from the Command Server onto the Data Server using the Unix `rsh` utility, for example:
`rsh ds1 sysstat 5`

6.4.2 Metadata Servers

Statistics will be gathered from the Metadata Servers by remote execution (via `ssh`) of the `db_stat` command.

6.4.3 Access Servers

The Command Server provides the ability to query the Client Daemon(s) on each Access Server for statistics. The following events will be statistically measured:

- Information about Client Protocol requests made by applications, broken out per session.
- Information about DB requests made by applications, broken out per session.
- Information about NFS requests made to Data Servers, broken out by Data Server.
- Information about DB requests made to Metadata Servers, broken out by Metadata Server.

For each of the above areas, the statistics gathered will be:

- Protocol-level statistics on both the client and server side, comparable to `nfsstat -cn` on a Solaris server.
- RPC-level statistics on both the client and server side, comparable to `nfsstat -cr` on a Solaris server.
- RPC performance statistics on depth of request queues, latency statistics, and bandwidth over the wire.

6.5 Implementation

6.5.1 Application Structure

The DI/ODE Command Server will be structured as a single Erlang/OTP application. It will rely on the Client Daemon and its facilities.

6.5.2 Process Tree

The following are organized under a top-level supervisor.

- Command Session Master: Accepts new command session connections from command-line tools, and monitors existing sessions.
  - Command Sessions: one I/O process converts the I/O protocol into messages on the socket, another process uses the I/O protocol to parse commands and write the replies.
- Client Daemon Monitor Master: controls the contents of the Client Daemon table and monitors the per-Client Daemon processes.
  - Client Daemon Monitors: Once process per Client Daemon in the DI/ODE cluster. Responsible for XappXing a periodic ping to the Client Daemon, polling for events, alarms, and statistics.
- Ping Listener: listens for detached Client Daemon homing pings, and instructs the Client Daemon Monitor Master to start monitoring the orphaned Client Daemon.
- Data Server Monitor Master: controls the Data Server table and supervises the Data Server Monitors.
  - Data Server Monitors: Will periodically ping Data Servers, and remotely execute programs to gather statistics.
- Metadata Server Monitor Master: controls the Metadata Server table and supervises the Metadata Server Monitors.
  - Metadata Server Monitors: Will periodically ping Metadata Servers and remotely execute programs, or make RPC calls, to gather status and statistics.
- Access Server Monitor Master: controls the Access Server table and supervises the Access Server Monitors.
  - Access Server Monitors: Will periodically ping Access Servers and remotely execute programs to get Operating System status and statistics (distinct from those provided by the Client Daemon Monitor).
- Alarm Handler
- Error Logger

6.5.3 Global Tables

The Command Server has the following ETS tables:

- Client Daemon table
- Access Server table
- Data Server table
- Metadata Server table

6.5.4 Event and Alarm Handling

In OTP/SASL (Erlang SASL, not RFC 2222 (authentication) SASL), an *event* is a one-time occurrence of importance, and an *alarm* is a persistent state of importance. Applications have APIs for generating events and raising alarms. Custom modules can be added to centralized event handlers to produce behavior that is decoupled from the code that actually generates the events or alarms.

In the Client Daemon, we have a flexible mechanism for being able to raise alarms based on sequences of events, or generating events based on raised alarms, further decoupling these decisions from the code that raises them.

The Command Server will poll each Client Daemon for its current alarm state and recent event history. These alarms and events will be placed into the Command Server's own alarm and event infrastructure. Thus, a monitoring tool need only monitor the alarms or events on the Command Server to see the behavior of the entire DI/ODE cluster.

6.5.5 Report Browsing

Events can be logged to local disk with the `log_mf_h` event handler. The Client Daemons will use this to log events to their local disks, using only a bounded amount of storage. The Command Server will be able to browse its own local log, and also remotely browse the logs of the Client Daemons.

6.5.6 Crash and Restart

As the Command Server is not necessary for the operation of the DI/ODE cluster, and will not be redundant in the future, we need a way for a new Command Server to be brought, stateless, into a running DI/ODE system.

All Client Daemons will expect a periodic signal from the Command Server. If they do not receive this signal within some time interval, say 10 minutes, they will begin to send a beacon message (consisting of their Erlang node name) to a broadcast address on the Command Network or the Data Network. The Command Server will listen for these beacon messages in order to find all operating Client Daemons, hence Access Servers. The Command Server can then query the Client Daemons for their File and Metadata Views in order to determine the set of back-end machines and the current migration state. In this way, the configuration state of a running DI/ODE system can be dynamically rediscovered by a new Command Server.

Chapter 7

File Migration

7.1 File Migration Problems

Any successful migration scheme must address a number of concerns. First and foremost, it should permit the I/O subsystem to emulate the semantics of UNIX file systems as closely as possible. In addition, there are a number of issues the migration scheme must address. These issues are subtle and end up making the proposals in this section look awfully complex. We enumerate many of them here to demonstrate why the proposals are as complex as they are.

1. The migration algorithm must be robust in the event of a Client Daemon failure.
2. The migration algorithm must be robust in the event of a Data Server failure.
3. The arbitrary renaming of files and of directories can cause enormous synchronization problems, particularly when moving a file or directory into a parent, ancestor, or cousin directory. Coordination with any "sweeper" process(es) used by the algorithm can get evil.
4. It must avoid race conditions between one process writing a file and another process migrating that file. Similar race conditions occur when a delete operation on a file is attempted while that file is being read, migrated, renamed, etc.
5. It would be nice if file link counts could be preserved, as well as file and directory `st_ctime` and `st_ino` attributes. The latter is impossible with NFS, while the former would merely be handy.
6. The migration scheme should work correctly with NFSv3-based Data Servers. Ideally, it should not do anything that would be difficult or impossible with Data Servers using NFSv4, DAFS, or other shared file systems.
7. The migration scheme should be as application-independent as possible. For example, we do not wish to tie the migration scheme to use the locking mechanism used by applications.

This is not intended to be a complete list. Rather, it's food for thought when considering the proposals below.

7.1.1 Why Directory Renaming Is Evil

One of the items listed above concerns stability under arbitrary directory moves. The fundamental problem posed by such moves is that of obtaining a file or directory handle in the new view that corresponds to an already open handle in the old view. We have considered various schemes for

doing such a mapping while directory renames are going on and suspect that while possible, the complexity of such schemes is greater than what we can design and implement confidently in a reasonable schedule.

7.2 The Freezing Process

An object (file or directory) in the OLD view is said to be frozen if attempts to operate on the object are prohibited by the Data Server. The attributes used to freeze an object must be distinguishable from object attributes unrelated to file migration, such as an attempt to write to a file with permissions 0444 by a non-0 UID application during a non-migration period (i.e. the file is "supposed" to have permissions 0444).

Freezing may also be used to deny read access to an object: setting the object's permissions to 0000, for example. However, the object migration algorithm will not do this: it can be advantageous, from an operation latency point of view, to have read-only access to a frozen object.

Specifically for Boardwalk 1.0, a frozen object in OLD should be owned by a UID reserved specifically for frozen objects, the group ownership will be the object's "real" GID, and all write permissions bits will be forced off.

A fundamental assumption about the freezing process is that there is no harm in "refreezing" an object. More specifically, applying the freezing attributes in the OLD location to an object that already has them should do no harm. In the case of NFSv3, we know that is true.

Another fundamental assumption about the freezing process is that an object in the OLD location, once it has frozen attributes, cannot have those attributes removed. Stated another way, the object in the OLD location cannot be "unfrozen". The migration algorithms layered on top of the freezing process rely on being absolutely certain that once an object in OLD is frozen that it cannot be subsequently modified.

We also place the restriction on the migration process that once started, it cannot be aborted or suspended. It must be continued through to completion. Of course, a subsequent set of migration operations may undo a previous migration.

7.2.1 The Freezing Process, RPC Credentials, and File Ownership

Changes to an object in the OLD location are not permitted during Phase 2 of migration. The only exception is during the brief period of time when some Client Daemons are in Phase 2 and others are still operating in Phase 1. There is no contradiction, however, because the Client Daemons in Phase 1 haven't yet found out about the change to Phase 2. These phases are presented in detail beginning in Section 7.5.

The whole purpose for freezing a directory or file is to prohibit any change to that object after freezing it. Any migration algorithm will fall apart (or at least become much more complex) without the freezing technique.

The current Client Daemon implementation performs all NFS operations using UID 0 and GID 0 in the RPC call credential. This allows the Client Daemon to emulate all of the file I/O semantics that the UNIX kernel provides, including the ability of the "root" user to bypass file system security checks, change ownership of files, etc.

There are two problems with using UID 0 in the RPC call credential. The first is that it's possible to accidentally "unfreeze" a file. If a file is frozen, then there must be at least one Client Daemon in the cluster that is operating in Phase 2. Another Client Daemon, still operating in Phase 1, could reset the UID of a frozen file, "unfreezing" it.

The freezing process *must* be able to prohibit operations on objects owned by UID 0. However, by definition, an NFS operation with UID 0 in the credential bypasses all security checks. There is an unavoidable race condition where a Client Daemon operating in Phase 2 freezes a file and begins

migrating it. Another Client Daemon, operating in Phase 1, modifies the file. The result is a file in the NEW location with possibly bogus data.

Therefore, we cannot use UID 0 in NFS RPC credentials (at minimum) during Phase 1. The strawman proposal below goes even further: UID 0 RPC credentials will only be used for migration purposes.

7.2.2 File ownership strawman for Boardwalk 1.0

All files on disk will be owned by the same non-zero UID and GID; call it “diode”. (Of course, the UID and GID must be numeric, but “diode” is a nice label.) If that creates security problems (one application fiddling with files “owned” by another), then segregate the mount points for different apps. If that isn’t sufficient, restrictions could be added to the Client Library and/or the Client Daemon to enforce segregation.

This “diode” UID and GID should be configurable in a Client Daemon configuration file.

I recommend some sort of Command Protocol addition to pass application effective UID and effective GID (and secondary group membership) info to the Client Daemon.

If an application’s EUID == 0, then `chown()` and `chgrp()` operations are no-ops, always returning success. If the EUID != 0, then `chown()` always returns `EPERM`, and `chgrp()` would either fail with `EPERM` or be a no-op success, depending on the process’s group membership list.

All non-migration NFS operations are done using RPC credential containing the “diode” UID and GID list of the application process. Migration operations may be done using UID “diode” or UID 0, as necessary.

The UID for frozen files shall be a specially-reserved UID; call it “diode_frozen”. Furthermore, it may be advantageous to have states beyond merely “diode_frozen”, and some additional reserved UIDs could be useful for this purpose.

7.3 Migrating a directory

0. A directory cannot be migrated unless its parent directory is already frozen.

   NOTE: Directories above the first hash directory component cannot be migrated. For example, for a mount specification `/var/spool/mqueue//%h`, nothing above `/var/spool/mqueue/foo` can be migrated.

1. Read the directory’s attributes in the OLD location. If the directory is frozen, your task is done.

2. Create the directory in the NEW location. If the MKDIR fails with `EEXIST`, you’ve lost an honest race with another process migrating the directory, so your task is done.

   NOTE: We require that the Data Server’s NFSv3 implementation’s MKDIR operation be guaranteed atomic in the same why that a GUARDED or EXCLUSIVE file CREATE operation is guaranteed atomic.

3. Freeze the directory in the OLD location. The NFS SETATTR operation will give you the directory’s attributes at the instant the file was frozen.

4. If the attributes from steps #1 and #3 differ, then another process changed the directory’s attributes behind your back. Change the directory’s attributes in the NEW location to reflect the change.

NOTE: There is no solution to the race between fetching the directory’s attributes in the OLD location and later freezing it. It is possible for directory’s attributes to be changed in the OLD

Original	**Where it's stored**
UID	Lower 23 bits of GID in NEW
GID	GID in NEW
lower 9 permissions bits	Upper 9 bits of GID in NEW
upper 3 permissions bits	Upper 3 permissions bits in NEW

Figure 7.1: File Attribute Encoding

location between steps #1 and #3 and *then* have the party performing the migration fail after step #3 and before the change step #4 can be performed. We agree to live with this race condition.

NOTE: Throughout this doc, I intentionally make a distinction between migrating a directory and migrating the contents of a directory. Note that the algorithm above does not make any reference to the contents of the directory. This is intentional. Migration of a directory's contents is handled by separate algorithm(s).

7.4 Migrating a file

0. A file cannot be migrated unless the directory storing the file is already frozen.

1. Read the file's attributes in the OLD location. If the file does not exist, your task is done. If the file exists and is frozen, check the file's attributes in the NEW location. If the mtime in the NEW location hasn't been updated "recently enough", then the Client Daemon migrating this file has crashed. Use the "tower of locks" algorithm to determine who gets the responsibility to finish migrating the file.

   If the file does not exist in the NEW location, attempt to create (exclusively!) the file in the NEW location. If the CREATE operation succeeds, you have responsibility to migrate the file's contents. If the file creation failed due to `EEXIST`, you lost the race with someone else attempting to migrate the same file. As the loser, you must block until you detect that the file's migration has finished or that you have detected that the migration stalled and you yourself have successfully finished migrating the file.

   The file in the NEW location is created using the file's original name, but it is owned by the migration UID and permission bits 0700. The original permissions bits are encoded in the GID. This is done to permit recovery if the migrator crashes. See Figure 7.1 for details.

2. Freeze the file in the OLD location. The SETATTR call will return the file's attributes at the instant the file was frozen. If the attributes are different from the attributes found in step #1, then another process changed the file's attributes behind your back. Change the file's attributes in the NEW location to reflect the change.

3. Copy the file's contents from OLD to NEW locations, sequentially, starting from the first byte of the file, so the size of the NEW file is a progress indicator, and in the case of a failure of the migrator, whoever acquires the lock can pick up where the previous migrator left off.

4. Reset UID, GID, and permissions bits in the NEW location to its original values.

5. Unlink the file from the OLD location.

7.4.1 `EPERM` note

Someone operating upon a file while in transit will receive an `EPERM` error. Since all files on a Boardwalk 1.0 Data Server are owned by the same UID, `EPERM` can only mean that the file is in the middle of being migrated. If the file's mtime in the NEW location is "recent", then there's nothing to do other than to wait for the migrator to finish the job. If the file's mtime is not "recent", then the Tower of Locks algorithm is used to fight for the right to resume the file's migration.

Once the file has been successfully migrated, then the original I/O operation can be retried.

7.4.2 RENAME and Migration

Subtleties exist in the migration algorithm with a RENAME (or REMOVE) operation. One might at first expect that if a RENAME happens right after step #4, and the migrator crashes before step #5 can be done, then the file will be migrated again later, won't it?

The answer is no. Both the RENAME and REMOVE operations require that the file must first be migrated to the NEW location. The RENAME/REMOVE operation *must* make absolutely certain that the file does not exist in the OLD location before it can perform the RENAME/REMOVE in the NEW location. If it does still exist in OLD, then migration was interrupted and must be completed before performing the RENAME/REMOVE operation in NEW.

Assume for a moment that a file migration was interrupted between steps #4 and #5. Since the file's ownership has been reset to normal in the NEW location, it's possible for the file to be modified (by WRITE, SETATTR) before anyone notices that the migration was incomplete. When resuming the migration, if the file in NEW is not owned by the migration user, then you can only assume that the migration was interrupted between steps #4 and #5: the contents of the file in NEW may not match what's in OLD. However, the contents of the file in NEW do not matter: the process that resumes migration of the file knows the migration was interrupted after step #4, so it simply performs step #5.

7.5 Migration Phase 0

The Client Daemon operates using old location view mapping and NFS procedures. For notation purposes, the Client Daemon is using a view called V_{old}.

7.6 Migration Phase 1

1. The Client Daemon receives a Command Server directive to move to Phase 1 of migration from V_{old} to a new view, V_{new}.

2. The Client Daemon resets its internal state to perform all new I/O operations according to the rules of Phase 1, using the views V_{old} and V_{new}.

3. The Client Daemon monitors all pending I/O operations performed using V_{old}. When they have all completed, then the Client Daemon may acknowledge that it has successfully completed the transition to Phase 1.

NOTE: Due to the asynchronous message passing used in Erlang, the implementor(s) must be careful to be absolutely certain that all pending Phase 0 operations really have completed and that no "new" Phase 0-style operations are unintentionally performed. The same applies to changes from Phase 1 to Phase 2.

7.6.1 Phase 1 Operation Rules

All I/O operations will continue to take place in locations dictated by V_{old}. However, if a frozen directory or file is discovered, the Client Daemon will make its transition to Phase 2 as if it had received the directive from the Command Server. There is no need to look in the V_{new} location.

Any operation that fails due to `EPERM` or `ESTALE` presents a special problem. `EPERM` is the result of attempting to operate on a frozen file or directory, and `ESTALE` is the result of either unlinking the file due to migration or unlinking for some other reason. In either error case, the status of the directory containing the object must be checked for frozen status. If it is frozen, then proceed to Phase 2, then retry the operation.

There is an invariant that the freezing algorithm provides for directories: the parent directory of a frozen directory must also be frozen. Therefore, if it is (for whatever reason) inconvenient to check a parent directory's frozen status, it suffices to check the status of the top-level directory: if the top-level directory is frozen, it's certainly possible that your operation failed due to freezing. Regardless of the real reason why the operation failed, the Client Daemon must proceed to Phase 2, then retry the operation.

7.6.2 Directory Renaming

The logistics of directory renaming during a migration are tricky. However, the freezing algorithm ends up making the task easier in some respects.

There are three particularly nasty problems involving directory renaming during a migration. Both are problematic both during Phase 1 and Phase 2. One is the problem of trying to find a directory FH (file handle) in the NEW location while that directory is being renamed. (The other is attempting to resolve paths "above" a current working directory when that cwd or one of its ancestor directories is renamed and attempting to resolve paths "below" the directory being renamed.)

7.6.3 Other Possible Solutions

For completeness, we discuss here some ideas which deserve to be documented which we will not implement. Assume (for now) that there is no separate migration phase to coordinate and synchronize directory renaming while migration is underway. This problem could be solved if such synchronization were done, but since the synchronization problem is itself difficult, we just won't go there.

Furthermore, we assume that a directory cannot be renamed in the OLD location while in Phase 2. That would violate a rule of Phase 2 operation: non-migration-related changes to data or metadata cannot be made to any object in the OLD location. It might be possible to relax that rule specifically to deal with directory renaming problems, but such an exception would make an migration algorithm tremendously more complex.

7.6.4 The Boardwalk 1.0 Solution

Before a switch to Phase 2 can be acknowledged, all cache directory FHs must have a counterpart directory FH in the NEW location cached, too. However, in order to cache a FH, that FH must exist, which means the directory in NEW must exist. Since we're in a migration phase, we'll almost certainly have to create the directory in NEW first.

In order to create that directory in the proper location in NEW, we need to know its full path in OLD. We use the getcwd algorithm to walk up the directory tree in OLD, then walk back down again to make certain that no one has renamed our directory (or one of its ancestors). [**NPC: The "getcwd" algorithm needs to be explained somewhere.**] Unlike the typical getcwd algorithm, however, we will create directories in the NEW location as we work our way back down. As a further side effect of doing so, each corresponding directory in OLD will be frozen.

If the getcwd algorithm fails its walk back down the tree, we know someone has renamed a directory along the way. Fortunately, the directories along the way have been frozen, so further modification is not possible. The getcwd algorithm is retried from the start, walking up and down the directory tree again. If the walk down fails again, we know another renaming happened, so iterate getcwd again. This process continues until getcwd succeeds. The iteration *must* eventually terminate, because eventually every single directory in OLD will be frozen as a side-effect, so eventually there will be no unfrozen directory for a pathological race winner to rename a directory to.

The logistics of implementing cross-directory renaming are so complicated that we don't know if the problem is solvable. Therefore, for Boardwalk 1.0, cross-directory renaming is prohibited. No attempt will be made by Boardwalk 1.0 to differentiate between renaming directories and renaming files.

7.7 Migration Phase 2

1. The Client Daemon receives a Command Server directive to move to Phase 2 of migration from V_{old} to a new view, V_{new}.

2. The Client Daemon resets its internal state to perform all new I/O operations according to the rules of Phase 2, using the views V_{old} and V_{new}.

3. Before acknowledging a transition to Phase 2, all cached information regarding directory and file NFS filehandles must be updated to include NFS filehandles for corresponding objects in V_{new}. This means migrating each directory and file for which there's a cached NFS filehandle. Furthermore, each file in the OLD location cannot yet be deleted.

4. When they have Phase 1 I/O operations have completed *and* all open file and cached directory information have been updated with filehandles for objects in their NEW locations, then the Client Daemon may acknowledge that it has successfully completed the transition to Phase 2a.

5. When all Client Daemons have acknowledged that they have switched to Phase 2a, the Command Server will issue the command to switch to Phase 2b.

6. While in Phase 2 (a or b), migration of files will be done in an on-demand, "file at a time" basis. See above for discussion of operation-specific prerequisites, handling of `ESTALE` & `EPERM` errors, etc. (Translation: I'm too lazy to repeat them all here.)

The only operation difference between Phase 2a and Phase 2b is that a Client Daemon may be able to cache status information while in Phase 2b about objects in OLD locations that it cannot cache during Phase 2a. It is not necessary to implement such a cache, and in which case, the distinctions between 2a and 2b are moot.

7.8 Variations on Migration Techniques

It is necessary to determine which NFS operations will trigger the migration of an object. Four possible criteria are listed here:

1. Any data- or metadata-modifying operation upon the object, e.g. WRITE, SETATTR, RMDIR.

2. Any data- or metadata-modifying operation as well as any READ operation upon a file. Read operations on a directory object (LOOKUP, READDIR) would not trigger migration of that object.

3. Any operation, including read operations on directories.

4. A migration sweeper.

In previous discussions about migration algorithms, variations #1 and #2 above give behavior similar to Jim's "file at a time" proposal. Variation #3 gives behavior similar to Scott's "directory at a time" proposal.

For the Boardwalk release, we propose that criteria #1 and #2 will trigger a migration, and a special sweeper utility will be constructed that will force migration using method #4.

7.8.1 Migration: "Directory at a Time" or "File at a Time"?

For Boardwalk 1.0 we will use the "file at a time" migration technique. The freezing technique makes it much more tractable than when it was originally proposed (without freezing).

7.9 Phase 2 Operation Rules

Here's a list of implementation notes for "file at a time" migration, sorted in operation order.

7.9.1 REMOVE

It seems wasteful to migrate a file, perhaps many megabytes, and then immediately unlink it. However, all other solutions pondered to date (see revision 1.4 of this doc for a list) have unacceptable problems. Therefore, the file must be migrated to the NEW location before it can be removed.

The Client Daemon should check if the client application making the REMOVE request also holds valid file descriptors for that file. If so, the file should instead be renamed using the `.nfs*` naming convention in order to avoid `ESTALE` errors should those file descriptors be used later. Also, the Client Daemon should take care to remove the temporary file in the event that those descriptors are closed or the client application crashes.

7.9.2 WRITE

Prerequisite: Since WRITE is a data-altering operation, the file being written must already be migrated before the WRITE operation can be attempted.

7.9.3 SETATTR

Prerequisite: the target object must first be migrated to NEW.

7.9.4 CREATE

Prerequisite: Since the file being created must be put into the NEW location, it's no surprise that the target directory in NEW must exist, which means that the target directory in the OLD location must be frozen.

A small optimization would be to check in OLD: if something exists with that name in OLD, immediately return `EEXIST`. This doesn't cause races with a migrator: if something by that name exists in OLD, then it's either in the middle of migration, so returning `EEXIST` is correct, or it hasn't been migrated yet, so returning `EEXIST` also correct.

7.9.5 MKDIR

Same situation as CREATE. See above.

7.9.6 RMDIR

Prerequisite: All of the parent directory's contents must be migrated before attempting the RMDIR operation in NEW; complete, recursive migration of that directory's contents is not necessary.

7.9.7 RENAME

Prerequisites: As discussed earlier, the object to be renamed must be completely, recursively migrated before the RENAME can take place. Furthermore, the parent directory of the "old" and "new" name must be the same: cross-directory renaming is prohibited during migration. Also, if something in the OLD location has the same name as the "new" name (i.e. the RENAME will clobber an existing file), that something must also be completely, recursively migrated before the RENAME can take place.

NOTE: An application process, A, should not rename any file that may be open by another application process, B. If B renames a file held open by A, there is no guarantee that A's file descriptor will be valid after the file has been migrated in Phase 2. Applications that have files opened by multiple processes at the same time are not supported in Boardwalk 1.0.

NOTE: If multiple processes have the same file open, it is possible that the file descriptor owned by one or more of those processes may become invalid in a race with a transition to Phase 2 and someone simultaneously renaming that file. The solutions to this problem involve suspending RENAME operations for a period of time or storing an NFS filehandle for the file's NEW location in the OLD directory somewhere (or in an external database).

In the interest of conserving development time, this is a problem we've reluctantly agreed we will not solve. We believe that our applications will not cause this problem, therefore a "head in the sand" approach is sufficient for Boardwalk 1.0. We will revisit this problem if the schedule permits.

7.9.8 LINK

Prerequisites: The original file (1st arg of link(2)) must already be migrated. If the "new" name already exists in OLD, we can immediately return `EEXIST`. (This is the same situation discussed in the cases of CREATE and MKDIR.) If the "new" name doesn't exist in OLD, then performing the LINK in the NEW location will race honestly with all other operations in NEW.

7.9.9 READDIR

Nothing special here, other than to state the obvious requirement that the union of directory contents in the OLD view and in the NEW view is required. Due to races with a migrator, the contents of the OLD view should be retrieved first.

7.9.10 Other read-only operations

All other read-only operations should first be attempted in the OLD location. If the object does not exist there, attempt again in the NEW location.

`ESTALE` and `EPERM` errors are handled as they are during Phase 1: if they occur while operating on an object in its OLD location, this is probably a sign that the object has been migrated to its NEW location. Retry the operation in the NEW location.

7.9.11 Tagging "Migration Done" In Directories In OLD

This isn't necessary for implementation of the migration algorithm, but tagging is mentioned here in case it becomes necessary. We can tag directories in OLD to indicate that "contents of this directory have been completely migrated".

Once all of the files from a directory in OLD have been migrated, the ownership of that directory would be changed from the migrator's UID to the "contents migrated" UID. (This UID would be taken from the DI/ODE reserved UID pool.) A directory owned by "contents migrated" would be interpreted to be frozen (as if owned by the frozen UID).

The benefit of this tag would be to permit caching of migration info: the Client Daemon's could avoid LOOKUP operations for files in a directory OLD if they already knew that directory's contents had been migrated. Each Client Daemon could cache this info on its own, but using the "contents migrated" UID could communicate that fact to other Client Daemons.

Like freezing, the "contents migrated" tag cannot be removed or undone.

It may also be beneficial for caching purposes to have another reserved UID, "contents recursively migrated". That could prevent unnecessary LOOKUP operations in OLD for intermediate directory components.

7.9.12 Tagging "Migration Done" In Directories In NEW

Also, it may be useful for a similar tag in the NEW location. Using magic directory UIDs is not possible in NEW, so I suggest a magic file name, e.g. ".migration_complete".

Since file operations during Phase 2 operate in view order of OLD then NEW, the only beneficial use of such a status tag would be when the Data Server storing the directory in the OLD view has crashed: directory operations such as namei, LOOKUP, and LINK could take place without any information from OLD if we know for certain that a directory's contents have been completely migrated to NEW.

This magic file would be filtered from applications' view by the Client Daemon, and it would be removed in a cleanup sweep once Phase 3 is entered.

7.9.13 Phase 3 Cleanup Sweep

Whenever an empty directory in the OLD location is found (either by the Phase 2 migration sweeper or in the normal course of a Client Daemon's operation), that directory may be removed. This is safe because each Client Daemon will cache a chain of NFS filehandles for ".." directories.

It may be necessary to have a separate sweeper process run in Phase 3 to remove any straggler directories that should have been removed during Phase 2.

Chapter 8

Metadata Migration

The metadata are partitioned into migration units which correspond to Metadata Servers. Migration units are determined by applying the consistent hashing algorithm to an application-specific portion of the keys of Sleepycat DB records. In the case of metadata redundancy, migration units can overlap each other; otherwise migration units are non-overlapping portions of the metadata.

For non-trivial metadata migrations, the set of Metadata Servers changes. This requires some subset of the total metadata to be copied to one or more Metadata Servers (doing multiple copies of the same metadata is only necessary in the case of redundancy).

Metadata migration employs a multi-phase algorithm, in which Client Daemons transition between three states under the coordination of the Command Server. The Command Server assures that at any given time, the set of Client Daemons are in no more than two distinct adjacent states. The possible states are:

1. OLD Location Only: Normal operation. All instances of metadata are correct and complete.
2. Phase 1, Prepare to Migrate: Metadata edits are done using the old location, just as during normal operation. However, metadata lookups and reads are checked first in the new location, in case another Client Daemon is already in phase two of migration and has migrated metadata, then in the old location. If at any time during phase one a Client Daemon detects that phase two of migration is in progress, it immediately transitions to phase two of migration.
3. Phase 2, Active Migration: If a metadatum exists in the new location, it is considered authoritative, regardless of whether a copy of the metadatum exists in the old location. If a metadatum exists only in the old location, it is copied from the old location to the new location, then the copy in the old location is deleted. All metadata modifications take place in the new location. For every metadatum access, the metadatum is deleted from the old location if a copy still resides there.
4. Post Migration: Normal operation again. This is the same as Phase 1 except with a new "View".

8.1 Phase Change Algorithms

For the following algorithms, $\mathcal{MDS}$ is the set of all Metadata Servers before migration begins (old view) and $\mathcal{MDS}'$ is the set of all Metadata Servers after migration completes (new view). Other sets of interest include $\{\mathcal{MDS} \cup \mathcal{MDS}'\}$ (all Metadata Servers both before and after migration), $\{\mathcal{MDS} \notin \mathcal{MDS}'\}$ (Metadata Servers in the old view but not the new), and $\{\mathcal{MDS}' \notin \mathcal{MDS}\}$ (Metadata Servers in the new view but not the old).

The Client Daemon uses the following algorithm to transition from normal operation to phase one of migration:

1. Receive a message from the Command Server to transition to phase one of migration.
2. $\forall\{M : M \in \{\mathcal{MDS}' \notin \mathcal{MDS}\}\}$:
   (a) Call `env_open`. If the environment doesn't exist, send an error response to the Command Server's "change migration phase" request, terminate this algorithm, and continue with normal operation.
   (b) $\forall\{D : D \in \mathcal{D}\}$, where $\mathcal{D}$ is the set of databases, call `db_open`. If the database doesn't exist, send an error response to the Command Server, terminate this algorithm, and continue with normal operation.

   Note that this implies the requirement that all environments and databases exist on new Metadata Servers before migration begins. This requirement is due to the fact that the Client Daemon does not have the knowledge necessary to correctly configure databases for the application's needs.
3. Start using the migration locking algorithms for new `lock_get` and `lock_put` operations. Doing this before the next two steps assures that at the end of the transition from normal operation to phase one of migration, all lock operations are using the migration locking algorithms.
4. For all pending `lock_get` operations:
   (a) Wait until the lock is acquired.

   Note that in the general case, it is possible that this step will never complete. However, some applications only uses non-blocking lock acquisition attempts. Furthermore, it is generally considered broken behavior for an application to hold a lock for a long period of time, so we may never need to solve this problem.
5. For all owned locks:
   (a) Acquire the lock in the new location.
6. Transition to checking for signs of second phase migration.
7. Send a response to the Command Server that the Client Daemon is now in phase one of migration.

The Client Daemon can transition to phase two of migration either due to an explicit request from the Command Server, or due to detecting that another Metadata Server has already started phase two of migration.

The Client Daemon uses the following algorithm to transition from phase one of migration to phase two of migration:

1. Either receive a request from the Command Server to transition to phase two of migration, or detect that another Metadata Server is in phase two of migration.
2. Fork new cursors for all existing cursors that point to objects that will be migrated.
3. Transition to phase two of migration.
4. If the Command Server sent a request to transition to phase two of migration, send a response to the Command Server that the Client Daemon is now in phase two of migration.

The Client Daemon uses the following algorithm to transition from phase two of migration to normal operation:

1. Receive a request from the Command Server to transition to normal operation.
2. Close all cursors that pointed to objects that were migrated.
3. $\forall \{M : M \in \{\mathcal{MDS} \notin \mathcal{MDS}'\}\}$:
    (a) $\forall \{D : D \in \mathcal{D}\}$, where $\mathcal{D}$ is the set of databases, call `db_close`.
    (b) Call `env_close`.
4. Transition to normal operation.
5. Send a response to the Command Server that the Client Daemon is now in normal operation.

For any of the phases of migration, if the Command Server sends a request to transition to the phase that the the Client Daemon is already in, the Client Daemon sends a response that the phase transition was successful. This is necessary for two reasons:

1. Since the Client Daemon can automatically transition from phase one to phase two before it receives a state transition request from the Command Server, the Client Daemon needs to pretend that it made the phase transition due to the Command Server's request.
2. A partial system failure such as the Command Server crashing or a transient network failure may make it necessary for the Command Server to re-send state transition requests to get all Client Daemons to known states.

8.2 Phase one algorithms

In some cases it is necessary to use transactions to protect operations in the old and new metadata locations. For these cases we denote the transactions as T_O and T_N, which are used for the old and new locations, respectively. The old and new versions of a metadatum are referred to as m_O and m_N, respectively.

During phase one of migration, some Sleepycat DB RPC requests must be handled in such a way as to detect if another Client Daemon has already begun phase two of migration, and in some cases the requests behave differently. Following is a list of differences from normal operation:

db_cursor: If the cursor points to an object in a region that is affected by migration, create cursors both for the old location and for the new location.

db_del, dbc_del: These operations avoid migration, since the object being operated on is going away. The operations attempt to delete the object in both the old location and the new location, in case there are Client Daemons already in phase two that have written the object to the new location.

1. Begin T_O and T_N.
2. db[c]_del m_O.
3. db[c]_del m_N.
4. Commit T_O if m_O exists; otherwise abort T_O.
5. Commit T_N if m_N exists; otherwise abort T_N.

db_flags, db_re_delim, db_re_len, db_re_pad, db_remove, db_rename: Not supported. All databases are guaranteed to be open during migration, so they cannot be configured, removed, or renamed.

db_get, dbc_get: These operations check for other Metadata Servers having made changes in phase two of migration.

1. Begin T_O and T_N.
2. db[c]_get m_O.
3. db[c]_get m_N.
4. If m_N exists:
   (a) Abort T_O and T_N.
   (b) Terminate this algorithm.
   (c) Execute the algorithm for changing from phase one to phase two of migration.
   (d) Run the phase two algorithm for this operation.
5. Abort T_O and T_N.

m_O contains the metadata to be returned to the application.

db_h_nelem: Inaccurate. Since objects can temporarily exist in two places during migration, there is the possibility that one or more objects will be counted twice. There is no scalable solution to this inaccuracy.

db_key_range: Due to possible duplicates, the result may be less accurate than normal. There is no scalable solution to this inaccuracy.

db_put, dbc_put: These operations check for other Metadata Servers having made changes in phase two of migration.

1. Begin T_O and T_N.
2. db[c]_put m_O.
3. db[c]_get m_N.
4. If m_N exists:
   (a) Abort T_O and T_N.
   (b) Terminate this algorithm.
   (c) Execute the algorithm for changing from phase one to phase two of migration.
   (d) Run the phase two algorithm for this operation.
5. Commit T_O.
6. Abort T_N.

dbc_close: Close both Client Daemon-level cursors associated with the application-level cursor.

dbc_count: (Not used by some applications.) Inaccurate. A range of objects can be partially migrated when `dbc_count` is called, which means that one or more objects that the `dbc_count` operation counts can be both in the old location and in the new location. The count of objects in the old and new locations is summed, so there is the possibility for objects to get counted twice.

In order to fix this problem, we would probably want to atomically move all objects with identical keys during migration. However, `dbc_count` is not used at all by some applications, so it will not be implemented at all initially.

dbc_dup: Since each application-level cursor actually has two Client Daemon-level cursors associated with it (one each for the old and new locations), duplicating an application-level cursor requires duplicating both Client Daemon-level cursors.

lock_get: During migration, locks must be acquired in both the old location and the new location in order to be compatible with normal operation before migration, then normal operation after migration. Order is important, since we are relying on a consistent locking hierarchy.

1. Acquire the lock in the old location.
2. Acquire the lock in the new location.

lock_put: The order of lock release operations is not important for correctness, but releasing the locks in the reverse order they are acquired reduces the probability of a `lock_get` operation failing with one lock held.

1. Release the lock in the new location.
2. Release the lock in the old location.

lock_vec: This call is composed of one or more `lock_get` and `lock_put` operations. The composed operations are piece-wise performed as described above (i.e., the first operation in the `lock_vec` call is performed as described above, then the second operation, etc.). In the general case, certain groups of operations may be optimizable into single calls to `lock_vec` in both the old and new locations, but we do not believe our applications use such groups of operations.

8.3 Phase two algorithms

During phase two of migration, some Sleepycat DB RPC requests either must be handled differently or behave differently than during normal operation. Following is a list of differences from normal operation:

db_cursor: Same as for phase one.

db_del, dbc_del: Same algorithm as for phase one.

db_flags, db_re_delim, db_re_len, db_re_pad, db_remove, db_rename: Not supported (same as for phase one).

db_get, dbc_get: These operations can cause automatic migration.

1. Begin T_O and T_N.
2. db[c]_get m_N with the RMW flag set.
3. db[c]_get m_O with the RMW flag set.
4. If m_N doesn't exist, db[c]_put the value of m_O to m_N.
5. If m_O exists, db[c]_del it.
6. Commit T_N.
7. Commit T_O if m_O exists; otherwise abort T_O.

db_h_nelem: Inaccurate (same as for phase one).

db_key_range: Inaccurate (same as for phase one).

db_put, dbc_put: These operations can cause automatic migration.

1. Begin T_O and T_N.
2. db[c]_put m_N.
3. db[c]_del m_O.
4. Commit T_N.
5. Commit T_O if m_O exists; otherwise abort T_O.

dbc_close: Same as for phase one.

dbc_count: (Not used by many applications.) Inaccurate (same as for phase one).

dbc_dup: Same as for phase one.

lock_get: Same as for phase one.

lock_put: Same as for phase one.

lock_vec: Same as for phase one.

Chapter 9

Future Work

This chapter describes the work that we anticipate in future releases of the DI/ODE system. The system will be structured to easily accommodate these anticipated features, even in its first release.

9.1 Redundancy Considerations

The most important future feature is the implementation of various types of redundancy. This will enable a DI/ODE-based system to continue operation without perceptible loss of service when any back-end system goes down. Though the design work for redundancy isn't completely fleshed out at the moment, we do have enough sketched out so that our Boardwalk implementation should make allowances for this future work.

9.1.1 Types of Redundancy

We will need to implement redundancy for each of the back-end components of the DI/ODE system.

Data Servers

Data Server redundancy is the most important and most problematic of the redundancy schemes. We need to ensure that replicated copies of data can be consistently and correctly read and written, in the presence of failures, with only a stock NFS server on the back-end. Lacking support for multiphase commit, we've had to invent other schemes.

Immutable Files Consistency problems are easy when the data don't change. For immutable files, we create $f + 1$ replicas to guard against f failures. Any available replica contains the correct data. The only subtleties are ensuring atomic creation of the files, and determining when the files should be deleted.

Mutable Files For mutable files, we create $2f + 1$ replicas to guard against f failures, and determine results based on a quorum vote of at least $f + 1$ available replicas. As we've discovered, the unit of reading has to be a subset of the unit of writing (for comparison and repair), therefore we restrict updates to be whole-file updates, implemented by moving a new copy into place.

Log Files Log files are append-only files with well-defined records.

Since the log file is mutable, we cannot use the standard stable storage algorithm to ensure consistency with only two copies of the data. If we need to use the full quorum voting algorithm, things will suck greatly, since:

- we'll have to store this one file three times, increasing disk and I/O consumption;
- we'll have to copy the entire file before we make our modifications in order to get the atomic update we need for quorum voting, hence giving $O(n^2)$ performance.

However, we can take advantage of the structure of the log file in order to show that a modified stable storage algorithm is amenable to our needs.

We need atomic appends to avoid partial records. The atomic append scheme needs to be easily verifiable — it is not acceptable to have to parse the entire log file to see if there is a partial record at the end or not.

The easiest way to implement atomic append is to have a constant record size — applications need only check that file length is a multiple of the record size. Failing that, we can either use an out-of-band value as a record terminator, or create such an out-of-band value by appropriate quoting of the data. Lastly, we could hold the "committed" file length in a side file, small enough that length values are written there atomically, which is updated once a log file write has completed. Partially-written records must be overwritten by the next log append.

The replica repair scheme also relies on an identifiable ordering of records in the log file. The most reasonable way to handle this is to require that each record start with a log sequence number (LSN), that will be known to the Client Daemon. Many applications will already have these properties, to allow log-scanning.

The log redundancy algorithm requires $f+1$ replicas to tolerate f failures. Log writes are done to all available replicas. Log reads must be done as a scan. The scan reads from a merge of all available replicas, merging them by LSN.

Note that faulty replicas can't be repaired without rewriting the potentially-huge log file. Therefore, the log storage format is best for short-lived or non-critical data.

We have to be cautious about the problem of multiple writers after a network partition, where each writer sees its own replica. If LSNs are incremented sequentially, the partitioned writers may use the same LSN for different records, which will complicate the merge process. To fix this, we'll require that part of the LSN will be a monotonically-increasing number within the log file, the other part a Client Daemon identifier. Upon the repair of a partition, each Client Daemon will scan the replicas for the highest monotonic part, and use that number uniformly for its next append. The per-Client Daemon identifier ensures that the merge of all records created during the partition time will fit into some allowable ordering.

Metadata Servers

Redundancy of the Metadata Servers is easier, since the underlying DB mechanisms are so much more powerful. With full transactional control of reads and writes, combined with multiphase commits, we anticipate little problem in adopting the quorum voting $2f+1$-redundant scheme.

Some subtleties may be needed to handle cursors.

Data Network

We will need redundant Data Networks, to ensure connectivity between the Access Servers and the Data Servers and Metadata Servers, even if the central switch of the Data Network Fails. Due to limits on the number of cards that we can insert into the target workstations of Boardwalk, we will limit ourselves to two redundant Data Networks, rather than $N+1$ redundancy. The implication of this is that we do not gain capacity via the addition of the second Data Network. If the bandwidth requirements exceed that of a single Data Network, then the second one cannot be considered "optional", it is now required. In order to help detect this situation, we elect to use only one of the two Data Networks in the absence of any failure in order to make sure that we have adequate bandwidth available in case a failure does occur.

One exception will be made to this policy — having network interface (NIC) cards for each of the redundant networks helps any node to ride out the failure of a NIC, in addition to the failure of the switch. Therefore, a host with a failed NIC to the active network may use the standby network.

Problems With the NFS Duplicate Request Cache

There is a problem with this, though, and that is in the way that most NFS servers implement their duplicate request cache. It is possible for an NFS client to send a request to the server which it processes and then ACKs. If the ACK is never received by the client, it will think that the operation was never performed and will resend it. This can result in data corruption on the server.

To mitigate this, the NFS client will resend these requests with the same transaction identifier as the original request. The NFS server will maintain a list of recently processed IP addresses plus transaction identifiers. So, if the scenario described above happens, the server will know that the previous request was completed, will not perform the action a second time, and will just ACK the request.

However, if the IP address from which the connection is sent changes between the original ACK being lost and the resend due to the failover of the redundant network, data corruption could result. Some possible solutions are: IP address alias trickery (might work on some kernels), modifying the server duplicate request cache to store host names rather than IP addresses, and adding state in the Client Daemon to attempt to detect these situations.

Non-Redundant Components of DI/ODE

Since the Access Servers are directly exposed to the Internet, we cannot provide redundancy for them. However, since they are dataless, clients can reconnect to any other Access Server providing the same service.

The Command Server does not need to be redundant, since it is not critical for operations. Likewise, the Command Network need not be redundant, as it can fail over to the Data Network.

9.1.2 Implications for Client Daemon Design

The Client Daemon is the multiplexer of I/O, and is the main component that will be changed to implement redundancy. Largely, the other components will be unaffected by redundancy, with exceptions as noted below.

General

Client Daemon operations which work on a single back-end entity will have to be modified to work on a collection of back-end entities. This has implications for both the process model and the data model.

Redundancy will turn every remote operation that the Client Daemon now performs into an ensemble of remote operations, each of which may succeed, fail, or time out. With each non-successful operation or inconsistent result returned, the Client Daemon may have to perform some background cleanup work. Also, any Erlang process cooperating in this work may encounter a fault of its own. Therefore, it makes sense to have the actions on each data replica take place within a separate process. To manage them all, a parent process should maintain links.

The data representations used within the Client Daemon need to be abstract against the creation of replicas of data. In particular, information about “the location” of data needs to be held at only a low level.

Consistent Hashing

Consistent hashing will be modified to return location sets instead of single locations. Since the application code already has to take migrations into account, we expect that the hashing machinery should be sufficiently abstract to handle redundancy as well.

File Service

The appropriate place for the data abstraction is in the inode layer.

DB Service

As with the file service, we need to be more abstract in the mapping of migration units to back-end servers.

RPC Service

Data network redundancy considerations for RPC calls.

The following Sleepycat DB RPC requests are not used by some applications, so support for them can be safely left until later:

- db_join
- db_swapped
- dbc_count

9.1.3 Implications for Metadata Server Design

Here, we *can* implement our own duplicate request cache based on hostname rather than IP address. Normally, this doesn't make much sense since we're using TCP as the transport protocol. However, now we're specifically guarding against failures that result in the termination of the TCP connection before the result can be returned to the client. [**NPC: Need implications for Data Server as well, specifically duplicate request cache.**]

9.1.4 Implications for Command Server Design

The Command Server needs to juggle a little more state, but fundamentally, things won't change much. It still maintains information about each Data Server, Metadata Server, and Command Server. It also remains in contact with each Client Daemon and Access Server, this doesn't change. Updates to Data and Metadata Servers pass through the Client Daemon on the Command Server just as they do on the Access Server, so the same issues apply to administrative utilities run on the Command Server as any Client Library linked application on the Access Servers.

Some administrative policy decisions may change, such as migration control in the face of redundant data operations, but the implications of these are TBD.

9.1.5 Implications for Application Design

If all goes well, these changes should be transparent to the application.

Appendix A

DI/ODE Client Protocol RPC Definition

```
/*
** This file contains the definition of the DI/ODE Client Protocol.
**
*/

/*
** Mount point definitions.
*/

/*
** DI/ODE status codes.
** Based on the NFS v3 status codes.
*/
```

Bibliography

[AVWkW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mi ke Williams. *Concurrent Programming in ERLANG, 2nd Ed.* Prentice Hall, 1996.

[BHG87] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[CPS95] Brent Callaghan, Brian Pawlowski, and Peter Staubach. Nfs version 3 protocol specification. RFC 1813, Internet Engineering Task Force, 1995.

[FKV00] Glenn Fowler, David Korn, and Kiem-Phong Vo. Sfio: A safe/fast i/o library, 2000. `http://www.research.att.com/sw/tools/sfio/`.

[FLC+00] Scott Fritchie, Jim Larson, Nick Christenson, Debi Jones, and Lennart Öhman. Sendmail meets erlang: Experiences using erlang for email applications. In *Proceedings of the 2000 Erlang/OTP User Conference*, 2000.

[Gos91] Andrzej Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley Publishing Company, 1991.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman Publishers, Inc., 1993.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[Sof00] Sleepycat Software. Berkeley db reference guide: Rpc client/server, 2000. `http://www.sleepycat.com/docs/ref/rpc/intro.html`.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., 1992.