A High Performance and Reliable Distributed File Facility

Rajmohan Panadiwal and Andrzej M. Goscinski

School of Computing and Mathematics
Deakin University, Geelong, VIC 3217, Australia

Abstract

In this paper we demonstrate that the typical problem of loss in performance due to disk I/O can be efficiently dealt with by using separate strategies for the storage of file data and for the data structures required for file management. Furthermore, to make the design very reliable, stable storage is provided. The design of the disk service allows the contents of a file to be distributed among more than one disk drive. Striving for reliability has also generated a new file facility architecture where a separate interface is provided for the transaction service. We claim that using transaction semantics file operations in not only database applications but also in system programming can be made resilient against system and media failure. In order to further improve performance the design of the caching module takes into consideration all the aspects of basic file and transaction services.

1 Introduction

In a distributed environment, it is of utmost importance to identify the critical areas for designing a disk service which influence the performance of the overall system: techniques to search the largest possible contiguous space in the disk; and the management of free space in the disk.

Either the absence of caching in the client machine as in the case of the 'Bullet server' of Amoeba [4] or poor implementation of caching could prove a major bottleneck in achieving the expected performance of the system. This implies that a significant gain in the performance due to the caching system alone can be easily realised, provided it is made available at the transaction level, the file service level and the disk service level.

Most systems do not provide to their users direct access to a disk service. Their users, as a consequence of it, use the file facility of such systems as an interface to the disk service. This results in inefficient and poorly written programs. Therefore, the performance of such programs can improve significantly, if they are allowed to directly use the functions provided by the disk service, however, in a limited and a protected manner.

One obvious place where there is a scope to improve performance is in the design of file organization. We believe, that any carefully designed file organization should be adequate to harness the power of a distributed system to its fullest extent, provided it allows access to the maximum amount of file data with the minimum number of disk references; allows files data to be spread throughout the system; and records the number of blocks which are contiguous.

We strongly believe that the provision of a uniform yet optional system-wide architecture for the implementation of a transaction service has the potential to avoid the proliferation of ad hoc mechanisms to provide concurrency control and recovery methods at both the system and application levels.

The issues which have been raised in the foregoing discussion show a strong possibility that their thorough investigation and careful implementation could lead to a major breakthrough in building a very efficient, highly reliable and available file facility for the distributed environment of RHODOS. The design of the RHODOS distributed file facility is oriented towards high performance compared to the utilization of disk space. The whole architecture and many solutions of the distributed file facility are original. However, we carried out our general design process based on the following [2, 5, 6, 11]. Moreover, the logical design of several services of the facility has been influenced by other works referenced in the following sections.

2 Development platform

2.1 Design goals of the RHODOS distributed file facility

The goals for designing the RHODOS distributed file facility are as follows:

- High performance – Performance of a distributed file system should be such that users should not see differences between a distributed system and a time sharing system using similar resources.
- Reliability – The design should provide the concept of stable storage to maintain mirror images of all the vital structural information and must have the provision to support the concept of file replication.
- Matching user requirements – A file facility should provide only those services to a user that

1063-6927/94 $03.00 © 1994 IEEE

match their requirements. This suggests that file operations involving transactions should be implemented separately.

- Storage of data – To yield the best performance, the disk service should provide adequate size logical units for storing file data and structural information.
- Modification policy – The modification policy for saving modified disk blocks back to the disk should take into consideration the requirements of a basic file service and a transaction service.
- Cache management – The caching module should allow caching of data in the client's computer and the file server to improve the performance.
- File size – For all practical purposes there should not be any limit on the size of a file.
- File facility semantics – The semantics of a distributed file facility should be easy to understand for the sake of easy adaptability.
- Resilience – User processes and servers must be able to recover easily from computer crashes and network problems.
- Configurability – Process(es) responsible for providing access to the transaction service should be created only when there is a need and they should cease to exist after providing the service.

2.2 Architecture

An analysis of the distributed file facilities described in [2, 3, 5, 9, 10, 15], shows that their development is based on different architectural models. However, there are some common elements that allow us to propose a general architecture, useful for research purposes [6]. We propose the architecture of a distributed file facility for RHODOS as shown in Fig. 1.

This facility consists of the following services: naming, replication, transaction, basic file, and disk. Each of these services has been implemented as a separate layer and provides a clean interface to its users. Each service performs a well-defined function.

We claim that the proposed architecture is oriented towards high performance as, it provides direct access to the disk service: to allow users to write better and more efficient programs; it provides a separate event driven interface to speed up access to a totally optional transaction service at the operating system level; and it provides caching at each level to avoid descending to a lower level to satisfy each request from the client.

The design does not take into account the physical location of the following services: naming, file and disk. These services can either co-exist on the same machine or be located separately on different machines.

As a commitment towards providing a flexible user-oriented distributed file facility we classify a file depending upon the context in which it is used and the

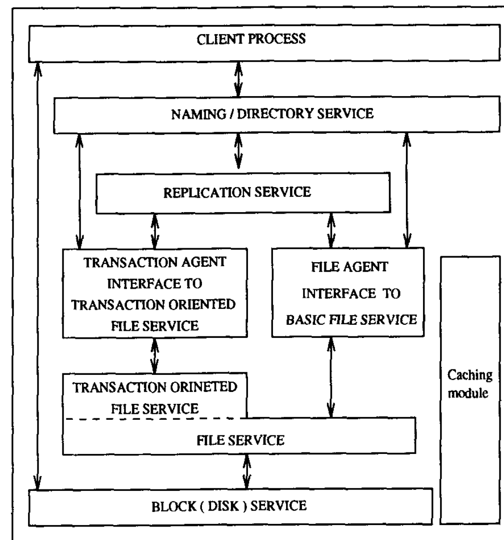

Figure 1: Architecture of the RHODOS distributed file facility

semantics of the operations performed on it. At any moment a file can be used either as a **basic file** (if operations on it are performed using the semantics of the *basic file service*); or as a **transaction file** (if operations on it are performed using the semantics of the *transaction service*).

There is no effort made to check the consistency of the final result of a number of processes concurrently reading and writing data from/to the same file using the semantics of the basic file service. The transaction service solves concurrency control and recovery problems. It also allows its users to assume that each of their programs executes atomically and reliably.

3 Client-Server interface

The RHODOS distributed file facility is based on the client-server model and its processing power is distributed among personal workstations and servers.

On each machine, all client processes acquire the services of the distributed file facility through special processes known as a **file agent** and a **transaction agent** for basic file service and transaction service, respectively. Also on each machine, there is one process called a **device agent** which facilitates I/O on devices such as communication ports, keyboards, and monitors.

In order to perform I/O on devices and files, processes in the RHODOS system use the attributed names of these devices, **TTY objects**, and files, **FILE objects**. However, the device agent refers to a device by its system name. Similarly, the file agent, transaction agent and the file service always refer to a file by its system name. Therefore, the process of evaluation and resolution of an attributed name of a device or file to its system name is performed by the

RHODOS naming service. After acquiring the system name of the device or file to be opened, the device, file and transaction agents return an integer known as an **object descriptor** which is used to uniquely identify each instance of a device and file opened by a process. In order to allow the redirection of I/O in the RHODOS system, the object descriptor returned by the device agent is always less than a predecided integer say 100,000. Whereas the object descriptor returned by the file and transaction agents is always greater than 100,000.

To allow I/O on devices each process is created with three global environment variables: *stdin, stdout* and *stderr* with 0, 1, and 2 as their default value respectively. If a process requests redirection of its standard output then its *stdout* variable is initialized with 100001. Similarly, if a process requests redirection of its standard input then its *stdin* variable is initialized with 100002. And finally, the variable *stderr* is initialized with 100003, if a process requests redirection of its standard error.

Certain errors caused by computer failures and communication delays may lead to repeated execution of some operations. However, their repetition in RHODOS does not produce any uncertain effect. This is because the semantics of the messages exchanged among the file agent, transaction agent, file service, and naming service constitute idempotent operations.

Due to idempotent file operations, a file agent maintains both the state of files when performing operations on them and the information about all past requests. As a consequence, the RHODOS file service is 'nearly' stateless.

The file agent and transaction agent cache a substantial amount of file data to avoid trying to access the file service for each request from a client. The file service also maintains its own cache to avoid access to the disk service for each request from a client. Finally, the disk service also maintains its cache to minimize the seek and latency time each time it accesses a disk.

A distributed file facility plays a very significant role in coordinating the creation of a mediumweight process. A mediumweight process in RHODOS shares its text and data space with at least one other process, but its stack is separate from stakes of other processes. This implies that a child of a mediumweight process will inherit all the object descriptors of the devices and files opened by the parent process and also the transaction descriptors of all the transactions initiated by the parent process. However, inheritance of the transaction descriptors of the parent process poses a serious threat to the serializability property of a transaction. Therefore, processes which perform I/O on devices and files using the semantics of the basic file service can only invoke the *process_twin* operation to create a mediumweight child process.

4 Disk service

We use blocks and fragments to represent logical units of information storage. A large block reduces the effect of latency. Moreover, if the block size is the same as the page size then it further reduces wastage of space in main memory. In order to improve the performance blocks are used for storing file's data.

Optimization of the disk space using a fragment is a very ordinary solution. The most important point which is overlooked in the process of optimization is that the use of fragments increases the disk I/O to a disproportionate extent. We believe, that for the storage of structural information of fairly small size the use of fragments can substantially reduce communication overheads and thereby improve performance. A fragment is 2K bytes long whereas a block is 8K bytes long.

Motivated by the goal of high performance, the RHODOS disk service implements its own caching strategy. This service retrieves only those blocks/fragments from a disk track which are necessary to immediately fulfill the requirement of a read request. Then the disk service caches the rest of the data from the same track. This is done in order to satisfy any subsequent requests to read data from blocks/fragments pertaining to the same track.

The algorithm to manage free space in the disk is heavily influenced by the fact that, generally, several contiguous blocks and fragments are allocated or freed simultaneously.

There is one disk server corresponding to each disk in the RHODOS system. Each disk server maintains a bitmap of the disk to which it is associated. A bitmap is updated when block(s) or fragment(s) are freed. In addition to a bitmap, the disk server also maintains a two dimensional array of the order of 64 rows and 64 columns for the maintenance of free spaces in the disk. The initialization and subsequent updation of this array is carried out by scanning the bitmap. The first row stores the references to single free fragments available on the disk. Each element of the second row is a reference to a group of two contiguous free fragments in the disk. Similarly, each element of the third row is a reference to a group of three contiguous free fragments in the disk and so on. Note, four contiguous fragments makes one block. The objective of this array is to check quickly whether a requested number of contiguous fragments or blocks are available or not. The use of this array not only improves the performance but also improves the storage utilization as each element of this array stores references to more than one fragment or block, except the first row of this array.

The RHODOS disk service provides the following service functions: *allocate_block, free_block, flush_block, get_block, and put_block*. In order to implement an efficient caching system, the semantics of these functions have been designed in such a way that any operation on a set of contiguous blocks/fragments can be accomplished in one single reference to the disk. The fact that stable storage is provided is reflected in the syntax of *put_block*. This function facilitates a user to specify if the data is to be saved exclusively on stable storage (as in the case of a shadow page) or on its original location and on stable storage (as in the case of the file index table). Besides saving data on original location if it is to be saved on stable storage then this function also allows its user to specify whether call

should be returned before saving the data on stable storage or after. The semantics of *get_block* also takes into account the provision of stable storage because it allows a user to specify whether a block/fragment is to be retrieved from the main storage (default) or stable storage.

5 Basic file service

The design of the RHODOS basic file service has been strongly influenced by the following: it should provide a platform with bare minimum overheads to suit applications which manage their own concurrency control and crash recovery; it should provide direct access to at least half a megabyte of file's data to allow all files up to half a megabyte to be cached using one single reference to disk; it should provide virtually no limitation on file size; and retrieval of all the contiguous disk blocks should be carried out using one single reference to the disk.

The RHODOS basic file service is essentially a **flat file service** because it is concerned only with implementing operations on a set of files without concern for any structure or relationship between the files. Like NFS [12] and LOCUS [14], files in RHODOS are *mutable* as opposed to *immutable* file used in AMOEBA [10].

Depending upon the semantics of the file operations, a file in RHODOS is classified either as a **basic file** or as a **transaction file**.

The blocks of a file may or may not be contiguous on a storage medium. Therefore, there is a need for recording a sequence of block descriptors for the blocks of which each file is composed. The sequence of block descriptors is stored in a separate data structure called a **file index table**. This allows both sequential and random access to a file's data. The location where a block descriptor is stored in the file index table is defined as a **block-index**.

We distinguish disk blocks on the basis of their contents, i.e., whether they contain file data or references to file data. For example, a disk block which stores the block descriptors as references to data blocks is defined as an **indirect block**. Whereas a disk block which stores file data is termed as **data block**; defined either as a **direct data block** if it stores file data and can be referenced directly from the file index table, or as an **indirect data block** if it stores file data but cannot be referenced directly from the file index table.

The use of block descriptors makes it possible to refer to the data blocks and indirect blocks regardless of their physical location. As a consequence of this a data block/indirect block can exist anywhere in the RHODOS system. This implies that files data can be distributed throughout the disk.

One of the unique features of this design is that in order to minimize the references to disk, the file index table stores along with each block descriptor a two byte count to indicate the number of contiguous successive disk blocks. The knowledge of the field count comes in very handy in achieving the goal of high performance, because all successive blocks, which are contiguous, can be cached using one single invocation of *get_block*, instead of *count* number of invocations of *get_block*. The file index table also stores the **file-specific** attributes: file size; date and time of file creation; last read access; a reference count to indicate the number of instances a file is opened simultaneously; service type to indicate whether operations on a file follow the semantics of the basic file service or transaction service; locking level to indicate level of locking; and space to indicate the amount of extra space needed for storing the file-specific attributes.

A copy of the file index table is always available in stable storage. Moreover, the file index table in RHODOS is only created dynamically, i.e., when there is a need to create a file. The advantages of dynamic creation are as follows: no wastage of memory; the file index table and at least the first data block are always contiguous thus eliminating the seek time to retrieve the first data block; the file index tables are distributed throughout the disk and hence the file facility does not run the risk of loosing all of them together.

Based on our requirements and the result of an analysis of distributed file systems presented in [2, 9], we propose the following file operations: *create, open, delete, read, write, pread, pwrite, get_attribute, lseek*, and *close*.

To achieve the goal of high performance and motivated with the result of an analysis of the caching system presented in [11, 13, 15, 16], we propose for RHODOS a caching system based on the main memory of the client and file service. The objective of this system is to reduce the cost of accessing data by storing recently-used blocks in local memory of the file service or client's machine, and reusing them when they are valid.

The space for caching a fragment and block is acquired from a **fragment-pool** and **block-pool**, respectively. The size of these pools is determined on the basis of the amount of main memory available. These pools of free buffers are maintained by the file agent, transaction agent and the file service.

To suit the semantics of the file operations supported by the basic file service, we decided to implement the *delayed-write* policy to save modifications made to data cached by the file agent. However, the *delayed-write* policy alone is not sufficient to cater for the needs of the file service, because file service is also responsible for coordinating access to file data using the semantics of the transaction services. This is why the *delayed-write* together with *write-through* policies are adapted to save modifications made to data cached by the file service.

We use three distinct steps to locate the file's data. The first step is to locate the file service which manages the file. The second step is concerned with locating and thereby subsequently caching the file index table from the file service identified in the first step. The third step is concerned with locating the data block(s), caching them, and finally passing the requested number of bytes to a user's address space.

6 Transaction service

One of the very important features of this design is its strong support for the provision of optional features

of transactions as part of the RHODOS distributed file facility. We believe that programs built using the RHODOS transaction service should outperform those which use transactions semantics at both system and application level. Another reason for the transaction service is that it should simplify reasoning about programs by allowing a programmer to think of each transaction as occurring indivisibly. Yet another reason which has influenced our decision is the provision that the transaction service should form the basis of a uniform system-wide architecture to avoid the proliferation of an ad hoc transaction mechanism at both the system and application level. Moreover, it should minimize the application programmer's responsibility and ensure that updates to files are always protected by transactions.

The *transaction agent* in RHODOS is a process which allows operations on a file using the semantics of transactions. The transaction agent process is highly dynamic because the first request to initiate a transaction in a client's machine brings this process into existence and it ceases to exist as soon as the last transaction in the client's machine either completes successfully or aborts.

In order to allow I/O on a file using the semantics of transactions, we provide a different set of file operations: *tbegin, tcreate, topen, tdelete, tread, tpread, twrite, tpwrite, tget_attribute, tlseek, tclose, tend*, and *tabort* [2]. The advantages of doing this are: improved performance, and no ambiguity as to whether a particular file operation belongs to the basic file service or the transaction service.

6.1 Concurrency control

The objective of concurrency control in the RHODOS transaction service is: to maximize the concurrency in the system, to preserve the consistency of file's data, and to guarantee successful completion of each transaction submitted to the transaction service.

Locking is the most popular concurrency control technique mainly because its use does not assume the serialization order a priori, but depends on the progress of transaction execution. Moreover, for the implementation of locking both in centralized and distributed systems efficient hierarchical algorithms are available. However, use of locking limits access to data item. We demonstrate that this problem can be easily tackled by carefully choosing the granularity of a lock and locking level. In RHODOS, we do not impose any restriction on the granularity; we provide record, page, or complete file locking.

In record level locking the size of a *data item* on which a transaction can set a lock is a record. A transaction is said to be operating in **record mode** with respect to a file from/to which it reads/writes data in the units of a records. The very purpose of fine granularity is to improve concurrency by allowing a transaction to lock only those *data items* it accesses. Therefore record level locking is the most suitable where the updates are small and the probability that a *data item* is subject to two simultaneous updates is remote. The disadvantage of record level locking is that it involves higher locking overhead, since more locks are requested.

In page level locking the size of a *data item* on which a transaction can set a lock is a page. A transaction is said to be operating in **page mode** with respect to a file from/to which it reads/writes data in the units of a page. This locking level is there to match the requirement of those applications where the updates are moderately large, and the probability of a *data item* being subject to more than one update is significant.

In file level locking the size of a *data item* on which a transaction can set a lock is a file. A transaction is said to be operating in **file mode** with respect to a file, provided it locks an entire file before it reads/writes from/to it. File level locking provides coarse granularity to the users of the RHODOS transaction service. Its usage incurs low overhead due to locking, since there are fewer locks to manage, hence it is most suitable where the updates are extremely large, totally unwieldy, and the probability of a *data item* being subject to more than one update is very significant. However, note file level locking reduces concurrency, since operations are more likely to conflict.

The design is flexible to the extent that the same file can undergo different levels of locking at the same time by different transactions. But to avoid complexity, we will assume that a file cannot be subjected to more than one level of locking by concurrent transactions. This constraint can be relaxed, if required, at a later stage.

6.2 Two-phase locking algorithm

Based on our requirement of high concurrency and an analysis of the concurrency control algorithm presented in [1], we propose to use the two-phase locking algorithm also known as the 2PL algorithm. It was shown in [7] that the 2PL algorithm is an optimal locking algorithm in the sense that it ensures the highest concurrency degree if: *data items* are independent of each other and represent the same level of abstraction; and transactions can be data dependent.

The execution of every transaction in the RHODOS transaction service proceeds through two phases: *locking* and *unlocking*. In the first phase new locks are acquired. In this phase a transaction has its own isolated copy of a *data item* to tentatively record the changes made in the *data item*. This isolated copy of the *data item* is defined as a **tentative data item** and its contents are invisible to other transactions.

In the second phase locks are released. Until a transaction commits or aborts, it cannot be decided whether other concurrent transactions should use the values of affected *data items* as they were before the transaction started or as they will be after the transaction has committed. Therefore locks cannot be released until after a transaction has been committed and the *data items* have been permanently updated.

6.3 Locks and their compatibility

The RHODOS system maintains a set of locks to synchronize the access to *data items*. These locks are *read_only RO*, *Iread IR*, and *Iwrite IW*. Two locks are *compatible* if they can be applied to the same *data item* by two or more transactions, otherwise they are *incompatible*. Lock compatibility for the RHODOS

Lock set	Lock to be set		
	read_only	Iread	Iwrite
None	ok	ok	ok
read_only	ok	ok	wait
Iread	wait	wait	Iread can be changed to Iwrite by the same transaction
Iwrite	wait	wait	wait

Table 1: Lock compatibility

transaction service is illustrated in Table 1. Locks can be *converted* into another. It happens when a transaction having imposed a lock on a *data item* requests locking the same *data item* in a new mode.

A *data item* is read_only locked by a transaction if the *data item* is needed to perform some query. A *data item* can be read_only locked by a transaction provided the *data item* is not locked by any transaction, i.e., it is free; or being *read_only* locked by other transactions. Otherwise, the transaction will be put into the wait queue. This is the only lock which can be shared by other *read_only* locks and a single *Iread* lock as illustrated in Table 1.

If a transaction reads a *data item* to modify it, then the *data item* is locked using an *Iread* lock, provided the *data item* is either locked with *read_only* lock by other transaction(s) or not locked by any transaction.

In order to prevent the occurrence of permanent blocking, once a *data item* is locked with an *Iread* lock, no transaction is allowed to set a new *read_only* lock on the the *data item.* This ensures that if a *data item* is read then its contents remain the same throughout the life of a transaction. If more than one transaction is allowed to share the *Iread* lock and if one of these transactions modifies the *data item* and commits, then the system will have to abort all other transactions resulting in a sheer wastage of computational resources.

A transaction can set an *Iwrite* lock on a *data item* provided the *data item* is not locked by any transaction, or the *data item* is *Iread* locked by the same transaction. An *Iwrite* cannot be shared by any other lock.

6.4 Deadlock in the RHODOS system

Based on our requirement and the result of an analysis of XDFS [8], we propose to use timeouts to resolve the deadlock in the RHODOS system.

The degradation of the RHODOS performance possibly due to its transaction service experiencing deadlock and/or permanent blocking could cause a transaction to take a long time. A transaction can also take a long time if it is nested or highly computation oriented.

This implies that until it is proved that a transaction is deadlocked and/or permanently blocked, there is no way to tell why a transaction is taking a long time. However, whatever may be the cause of the prolonged blockade of a transaction, a system cannot allow any transaction to be blocked for ever.

According to our solution, a transaction can set a lock on a *data item* provided the *data item* is free or the lock to be set is compatible with the lock already in place. Each lock is given a limited period, LT, in which it is invulnerable. After the expiry of LT, if no other transaction is competing for the *data item* that is locked, the *data item* with the invulnerable lock is allowed to remain invulnerable for another LT period. This way a lock can remain invulnerable for a maximum of N * LT. After the Nth expiry of LT, if the *data item* remains locked by the same transaction then it is suspected that the transaction is deadlocked and therefore its lock is broken and the transaction is aborted regardless of whether other transactions are waiting to acquire the lock or not.

The idea behind LT is to give an opportunity to the transaction holding the lock to complete its processing and to commit successfully.

There are two main problems with the use of timeouts. Firstly, the number of transactions timing out will increase as the load on the RHODOS system increases. Secondly, transactions taking a long time will be penalized. This is because our solution is based on the assumption that all those transactions which are taking a long time to complete are deadlocked and/or permanently blocked.

These problems imply that computing a value for the timeout period is not a simple matter. It depends on many hard-to-quantify variables: the physical characteristics of the sites and communication lines, the processing load, accuracy of clocks, among others [1]. However, we believe that this problem can be resolved by carefully choosing the value of the timeout.

6.5 Lock table

A lock table is a list of records: process identifier, transaction descriptor, phase of the transaction, type of lock, lock granted or not, retry count, descriptor of *data item*, and references to the same transaction and same *data items.* There is one record for each transaction which accesses a *data item* in a lock table, no matter whether access to the *data item* is granted or not. A transaction service in RHODOS supports the following operations on a lock: *get_lock_record, set_lock* and *unlock.* For each level of locking, a file server maintains a separate lock table. There are two advantages of maintaining a separate table for each level of locking: this arrangement makes the design of the transaction service clean and simple by avoiding the complexity of maintaining the locking information of three logically different data items in the same place; and it significantly reduces the number of records managed by each lock table and hence the time to search a record in the lock table is reduced, thereby making the RHODOS transaction service a high performance one.

All the records in the lock table pertaining to the transactions waiting to set a lock on the same *data item* are arranged in a queue using a singly linked list. This facilitates the first transaction in the queue to set the lock on a *data item* as soon the the transaction who holds the lock commits or gets aborted. Similarly, all the records in the lock table pertaining to differ-

ent *data items* are arranged in a queue using a singly linked list. It facilitates the searching of an entry of a particular *data item* in the lock table.

6.6 Transaction recoverability

The design of the RHODOS transaction service takes care of all sorts of failures (except for catastrophes). To safeguard against the loss of data in volatile and non volatile storage the concept of stable storage is used.

There are two commonly-used approaches to recovery from system and media failures. These are the intentions list approach and file version approach.

The file version approach is costly with respect to disk operations. Thus, taking into consideration high performance, we propose to use the intentions list approach for the provision of recovery after failure.

6.7 Implementation of the intentions list

Based on our requirements of high performance and reliability we propose to use the following operations on an intentions list: *get_intention, set_intention* and *remove_intention.*

There are two ways to make changes in the intentions list permanent when a transaction commits. These are: the shadow page technique and the write ahead logging (*wal*) technique. The *wal* technique does not change the sequences of disk blocks which stores the file's data. As a consequence of this, if the disk blocks which store file's data were contiguous before the beginning of a transaction they remain contiguous even after the transaction is committed. Contiguous allocation of disk blocks is preferred in RHODOS to minimize the seek and latency time while performing I/O on a file. The use of the *wal* technique retains the performance gain achieved due to the contiguous allocation of disk blocks which contains file's data. For this reason, we have decided to use the *wal* technique for the implementation of intentions list to support page level locking.

There is no justification to tie up a complete block or fragment for applications where record level locking is desired. For these applications, we also use the *wal* technique for the implementation of intentions list.

The shadow page technique requires lesser I/O overhead than the *wal* technique, because there is no need to copy blocks in the commit phase of the transaction. However, there are three major disadvantages of the shadow page technique. First, it uses an entire disk block in each intention, instead of the exact range of items specified in an operation. Therefore, this technique is not suitable for database applications where record level locking is desired compared to page level locking. Second, if the data blocks are contiguous before the beginning of the transaction then they are no longer contiguous after the transaction commits. Thus, this technique destroys the contiguity of data blocks. Third, it requires the replacement of the block descriptor of the original data block with that of the shadow block in the file index table.

Based on this we propose to use the shadow page technique when the data blocks are not contiguous and the *wal* technique when the data blocks are contiguous. Whether data blocks are contiguous or not is very easy to determine by using the knowledge of the file-specific attribute *count* associated with each block descriptor in the file index table. We also propose to use the *wal* technique to support the implementation of record level locking.

An intention flag maintains the information about the status of a transaction: tentative, commit or abort. The status of a transaction during the first phase is tentative. When a transaction enters the second phase then the status of the transaction is changed either to commit, if the transaction can be committed or abort. If the transaction can be committed then the transaction is completed by making permanent the changes listed in the intentions list. The intention flag keeps necessary information to allow a file server to take a decision on how the changes in the intentions list will be made permanent, i.e., by shadow page technique or *wal* approach. After making the changes permanent the records from the intentions list are deleted.

The exact contents of a record in the intentions list depends upon the mode in which a transaction is operating. If a transaction is operating in either file or page mode then the *tentative data item* is represented by a page or pages.

Each record in the intentions list maintains the descriptors of the *data item* and *tentative data items.* When a transaction operates in record mode then the transaction service does not pose any limitation on the size of a record as a *data item.* Hence, the granularity of a record as a *data item* can be as fine as a single byte or it can be as coarse as an entire file. Therefore a *tentative data item* of type record is represented using either fragments or blocks.

7 Conclusions

The initial results of the logical design and the initial stage of the implementation of the RHODOS distributed file facility are very encouraging. From the design point of view, there is practically no limitation on the number of disks connected in the distributed environment of RHODOS. The design allows each disk to be as large as 2000 gigabytes. Furthermore, a file can be partitioned and therefore its contents can reside on more than one disk. Thus, the size of a file can be as large as the total space available on all the disks.

The use of fragments for the storage of control data and blocks for the storage of file data improves the performance of the file facility. Following this the syntax and semantics of the functions provided by the disk service are so designed that all the contiguous blocks/fragments can be transferred to and from the disk in one operation.

Provision of stable storage ensures that all the important data structures used for file management in the distributed file facility are recoverable.

The design of the file index table allows up to half a megabyte of file's data to be accessed directly. Thus, for files up to half a megabyte, the maximum number of disk references is two: one for the file index table and the other for file data. Furthermore, creation of

file index tables on a need basis, ensures that they do not accumulate in one place on the disk.

The transaction service is placed at the operating system level to form the basis of a uniform system-wide architecture for transaction service. Moreover, it avoids the proliferation of an ad hoc transaction mechanism both at the application level and system level. A transaction agent as an interface to the transaction service improves performance by allowing maximum processing of transactions at the client computer by intelligently caching the relevant information.

The presence of a transaction agent is event driven: it is invoked only when there is a need to perform file operations involving transactions. Separate implementation of file operations in the transaction service ensures that the usage of transaction primitives is absolutely optional.

The transaction service provides three optional levels of locking: record, page and file locking. Provision of record locking as a fine level of granularity maximizes the concurrent execution of transactions. Moreover, to support default level of locking it exploits the knowledge of how frequently a file is used. In order to maximize the performance when a transaction is committed tentative changes are made permanent using the write ahead logging if file's data are stored using contiguous disk blocks, otherwise it uses the shadow page technique.

Acknowledgement

We would like to thank Mr. Damien Depaoli for reading a draft of this paper and his helpful comments.

References

[1] Bernstein, P.A., Hadzilacos, V., Goodman, N. "Concurrency control and recovery in database systems", Addison-Wesley Publishing Company, 1978.

[2] Braban, B. and Schlenk, P. "A Well Structured Parallel File System for PM," Operating Systems Review 23, No. 2, pp. 25-38, 1989.

[3] Brown, M.R., Kolling, K.N. and Taft, E.A. "The Alpine File System", ACM Transactions on Computer Systems 3, No. 4, pp. 261-293, 1985.

[4] Douglis, F., Ousterhout, J. K., Kaashoek, M. F. Tanenbaum, A.S. "A Comparison of Two distributed systems: Amoeba and Sprite," Computing Systems, Vol. 4, No. 4, pp. 353-384, Fall 1991.

[5] Gifford, D.K., Needham, R.M. and Schroeder, M.D. "The Ceder File System," Communications of the ACM 31, No. 3, pp. 288-298, 1988.

[6] Goscinski, A. "Distributed Operating Systems — The Logical Design," Addison Wesley, 1991.

[7] Kung, H.T. and Papadimitriou, C.H. "An optimal theory of concurrency control for databases", In Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 116-126, 1979.

[8] Israel, J.E., Mitchell, J.G., and Sturgis, H.E. "Separating data from function in a distributed file system", in Operating Systems: Theory and Practice, (Ed. Lanciaux, D.), North-Holland, Amsterdam, pp. 17-27, 1978.

[9] Leach, P.J., Levine, P.H., Hamilton, J.A. and Stumpf, B.L. "The File System of an Integrated Local Network," ACM Computer Science Conference New Orleans, 1985.

[10] Mullender, S.J. and Tanenbaum, A.S. "A Distributed File Service Based on Optimistic Concurrency Control," Proceedings of the Tenth ACM Symposium on Operating Systems Principles Orcas Island, Washington, pp. 51-62, 1985.

[11] Nelson, M., Welch, B. and Ousterhout, J.K. "Caching in the Sprite Network File System," ACM Transactions on Computer Systems 6, No. 1, 1988.

[12] Osadzinski, A. "The Network File System (NFS)," Computer Standards & Interfaces 8, pp. 45-48, 1988.

[13] Ousterhout, J.K., Cherensen, A.R., Douglis, F., Nelson, M.N. and Welch, B.B. "The Sprite Network Operating System," IEEE Computer 21, No. 2, pp. 23-36, 1988.

[14] Popek, G. and Walker, B. (Editors) "The LOCUS Distributed System Architecture," MIT Press, Cambridge, Mass: MIT Press, 1985.

[15] Satyanarayanan, M., Howard, J.H., Nichols, D.A., Sidebotham, R.N., Spector, A.Z. and West, M.J. "ITC Distributed File System: Principles and Design," Proceedings of the 10th Symposium on Operating Systems Principles Orcas Island, pp. 35-50, 1985.

[16] Schroeder, M.D., Gifford, D.K. and Needham, R.M. "A Caching File System for a Programmer's Workstation," Proceedings of the 10th Symposium on Operating Systems Principles Orcas Island, pp. 25-32, 1985.