A File System Interface for Concurrent Access

Andrea Skarra and Herman Rao

AT&T Bell Laboratories
Murray Hill, NJ 07974
{ahs,herman}@research.att.com

Abstract

The paper introduces a transparent service for synchronized replication across loosely-connected, heterogeneous file systems, and it focuses on the part of the service that supports transactional synchronization. The transaction paradigm is useful for grouping together the file accesses in a program that are logically connected, such that each group of accesses is synchronized as an atomic unit. The service, Parrot*, is implemented as a library that an application links and runs in its own address space. The library redefines and extends the file system interface* open/close/read/write*, while preserving its syntax and semantics.* Parrot *implements a read-one write-all synchronization protocol that (1) guarantees consistency across all copies of each replicated file at the granularity of individual read and write operations and (2) supports transactional synchronization for sequences of read and write operations on one or more files.*

1 Introduction

When concurrent processes access the same locations in primary or secondary memory, they typically use a synchronization protocol to preserve the integrity of both the memory locations and the access return values. In the operating system domain, the unit of synchronization is a single access to a memory location [3, 8, 10]. That is, a protocol for concurrent processes guarantees data consistency at *access boundaries* (i.e., the point at which an access terminates or the next access begins). For higher-level, application-defined synchronization schemes, an operating system (OS) typically supports a set of locking primitives.

In the original UNIX interface for time-sharing systems, open/close/read/write, write operations do not overlap with other reads or writes, and readers always receive the most recent data. In distributed environments, however, the consistency guarantees are typically weaker due to delays in cache updates (i.e., a reader may receive data from the cache that is slightly outdated) [11], and in replicated settings, the presence of multiple copies of the same logical file further complicates consistency maintenance (i.e., additional synchronization is required to guarantee that the order of updates is the same at all replicas). The replication service Parrot (Section 3) manages concurrent access to the files it replicates, such that the access semantics are the same as when the processes execute on the same host and access a single local copy of the data.

In contrast to the operating system domain, the synchronization protocols in database systems support a unit of concurrency that is larger than a single access, namely a *transaction.* A transaction is a sequence of operations on one or more data objects (i.e., a sequence of accesses to memory locations), and it represents a single logical action on the database. A protocol for concurrent transactions guarantees database consistency at *transaction boundaries*, and it does so by maintaining the isolation of each transaction in a concurrent execution [1]. As a result, a developer can reason about the state of an object over several operations in a transaction without regard to other transactions that may or may not be concurrent at run-time.

The transaction paradigm is also useful for synchronizing processes that share data managed by the OS (e.g., programs that read and write the same files). Suppose each record r in a file f contains an attribute *readnum* that stores the number of times that a program P reads r. P increments *readnum* in the following pseudocode, where the functions' parameter lists are abbreviated:

```
for every r in f                                    Sequence 1.1
    read(f); increment_readnum(); write(f);
```

Access-level synchronization alone is not sufficient to guarantee consistency in f's records when several instances of P execute concurrently. As soon as some P reads r, no other program should read or write r until after P increments *readnum* and writes r to f. The read-increment-write sequence is a single logical action on r (i.e., a transaction) that should be synchronized as an atomic unit.

In the absence of a separate transaction mechanism, a programmer can implement the isolation property of transaction synchronization for such sequences with OS locking primitives. The programmer simply brackets the sequence with primitives, such as exclusive_lock(f)...unlock(f). In the general case, a computation involves multiple files, and the program must request multiple locks at a time to ensure isolation. For example, a computation depends on the contents of two files *f1* and *f2*, while its result is stored in another file *f3*,

```
read(f1); read(f2); computation(); write(f3);       Sequence 1.2
```

and the program brackets the sequence with lock requests as follows:

```
share_lock(f1); share_lock(f2); exclusive_lock(f3) ... unlock(f3); unlock(f2); unlock(f1).
```

Deadlock can occur, however, when processes request multiple locks at a time. A synchronization protocol must either prevent or detect/resolve deadlock, whether it is provided by a transaction mechanism or defined by an application using OS locks.

Both the Network File System (NFS) and the Andrew File System [11] support primitives for locking (remote) files within a local area network. In a distributed setting, however, locking is more expensive due to higher communication costs, and deadlock detection in particular is more difficult because it requires a global algorithm. Indeed, we found that NFS does not detect deadlock across files stored on different hosts, and the average lock request time for a remote file is six to nine times that for a local file on a Sun SPARCstation 1+ under SunOS 4.1.1 [13]. We believe that the currently available OS locking primitives do not adequately support transactional synchronization in a distributed setting.

TX is a service that supports deadlock-free transactional synchronization of program sequences that access Parrot-replicated files across a local- or wide-area network. It implements a protocol that minimizes the number of messages required to synchronize concurrent processes. Section 2 describes TX and the relevant parts of Parrot, and Section 3 gives a brief overview of Parrot.

2 TX

Parrot is a transparent service for synchronized replication across loosely-connected, heterogeneous file systems. The service includes a token-based synchronization protocol that guarantees (1) atomicity of read and write operations with respect to other write operations and (2) sequentiality of writes to each logical file (i.e., the order of writes is the same at all physical copies of a single logical file). A process requests a write-token for a file f before it modifies f, and it obtains the token only when (1) no other process holds a read- or a write-token for f and (2) no updates are pending for the local copy of f. Parrot replicates only a portion of the file system: it manages only the subtrees that applications designate explicitly.

Parrot generates a *current sequence number* for each file f that it manages ($curSeqN_f$). It initializes $curSeqN_f$ to zero when a process creates f in a Parrot-managed subtree (or moves f to such a subtree), and it increments $curSeqN_f$ when a process modifies f with a write() operation. A process P receives $curSeqN_f$ as part of acquiring a read- or write-token for f. The value is global: $curSeqN_f$ reflects all the changes to f, whether the processes responsible for the changes are local to P or remote.

TX is the part of Parrot that uses the current sequence number to support transactional synchronization. To use TX, a process P opens a Parrot-managed file f with the flag O_TX. When P first reads or writes f, TX saves $curSeqN_f$ in a local variable $locSeqN_f$. When P next reads or writes f, TX compares $locSeqN_f$ to $curSeqN_f$. If they are not the same, another process has changed f in the interim, and TX returns an error. Otherwise, it invokes the OS read or write function and returns the result.

Parrot defines a total order over the files that it replicates to avoid deadlock: when a TX function requests multiple tokens, it does so according to the predefined order.

2.1 Interface

The TX control structures are encapsulated within a functional interface that redefines and extends the UNIX interface open/close/read/write, while preserving its signature and return values. TX defines an additional value for errno, ETX, that signifies an error due to transactional synchronization, and it defines two new functions, readtx() and writetx(), that access a set of files and have parameters of type fd_set* (as in the system call select) and tx_t*, where struct tx_t {char *buf; int nbyte;}. A program invokes the functions with bits set in the fd_set for the files to be read or written, and on an ETX error return, the functions clear all the bits in the fd_set other than those for files whose *curSeqN* has changed. The interface and its pseudocode are as follows:

- open (char *path; int flags; mode_t mode)
  If flags includes O_TX, initialize the local variable $locSeqN_{path}$ to zero, and remove O_TX from flags. Invoke the system call open and return the result.

- read (int fd; char *buf; int nbyte)
  Obtain a read-token for the file f opened at fd. If O_TX is *not* set for fd, then invoke the system call read, release the read-token, and return. Otherwise, if $locSeqN_f == 0$, then assign $curSeqN_f$ to $locSeqN_f$. If $locSeqN_f \neq curSeqN_f$, then set errno = ETX, else invoke the system call read. Release the read-token and return.

- write (int fd; char *buf; int nbyte)
  Obtain the write-token for the file f opened at fd. If O_TX is *not* set for fd, then invoke the system call write on the local copy of f, and if successful, send the update to the remote copies; release the write-token and return. Otherwise, if $locSeqN_f == 0$, then assign $curSeqN_f$ to $locSeqN_f$. If $locSeqN_f \neq curSeqN_f$, then set errno = ETX, else update f's copies as above, and if O_TX, increment $locSeqN_f$ (and $curSeqN_f$). Release the write-token and return.

- readtx (fd_set *readfds; struct tx_t *txp)
  For each set bit, readfds[fd], and in the order defined by Parrot, get a read-token for the file f opened at fd, and if O_TX is set for fd do: if $locSeqN_f == 0$, then mark f *is_null*, else if $locSeqN_f \neq curSeqN_f$, then mark f *changed*. If any f is *changed*, then set errno = ETX and clear all bits in *readfds except the ones for changed files, else for every file f, invoke the system call read and assign $curSeqN_f$ to $locSeqN_f$ if f *is_null*. Release all tokens and return. The function reads no files if any of them are *changed*; otherwise it reads them all.

- writetx (fd_set *depends_on,*writefds; struct tx_t *txp)
  For each set bit, depends_on[fd] or writefds[fd], and in the order defined by Parrot, get a read- or write-token respectively for the file f opened at fd, and if O_TX is set for fd do: if $locSeqN_f \neq 0$ and $locSeqN_f \neq curSeqN_f$, then mark f *changed*. If any f is *changed*, then set errno = ETX and clear all bits in *depends_on and *writefds except the ones for changed files, else for each fd in *writefds invoke the system call write; if successful, send the update to the remote copies of f, and if O_TX, set $locSeqN_f$ to $curSeqN_f+1$ (and increment $curSeqN_f$). Release all tokens and return. The function performs no writes if any of the files are *changed*; otherwise it performs all the writes.

- resettx (fd_set *fds)
  For each set bit, fds[fd], reinitialize $locSeqN_f$ to zero for the file f opened at fd.

2.2 Usage

TX supports application-defined retry protocols. To illustrate, recall Sequence 1.1 as an example of an access that begins a new transaction with each new record. The pseudocode for using TX is as follows for each record r in file f:

Sequence 2.1

```
if (fd = open (f, O_RDWR | O_TX)) < 0
    exit;
FD_ZERO (&fdset);
FD_SET (fd, &fdset);
for every r in f
    resettx (&fdset);
    for (try = TRY_NUM; try > 0; try−−) {
        if read (fd, buf, nbyte) < 0
            exit;
        increment_readnum();
        if write (fd, buf, nbyte) ≥ 0
            break;
        if errno ≠ ETX
            exit;
        /*  else reset and retry — some other process has modified f since read()  */
        resettx (&fdset);
    }
```

The program reinitializes $locSeqN_f$ with resettx() each time it begins a new read-increment-write sequence, since it constitutes a single logical action that does not depend on f's (previous) state.

In contrast, the pseudocode using TX for Sequence 1.2 on files *f1*, *f2*, and *f3* resets $locSeqN_{f1}$ and/or $locSeqN_{f2}$ only when it retries after failure of readtx() or writetx():

Sequence 2.2

```
/*  Open f1-f3 with O_TX flag at fd1-fd3  */
/*  Initialize tx_t array with buffers  */
FD_ZERO (&readset);
FD_ZERO (&writeset);
for (try = TRY_NUM; try > 0; try−−) {
    FD_SET (fd1, &readset);
    FD_SET (fd2, &readset);
    if readtx (&readset, txp) < 0 {
        if errno ≠ ETX
            exit;
        resettx (&readset);
        continue;
    }
    Computation();
    FD_SET (fd3, &writeset);
    if writetx (&readset, &writeset, txp) ≥ 0
        break;
    if errno ≠ ETX
        exit;
    /*  else reset and retry — some other process has modified f1 or f2 since readtx()  */
    resettx (&readset);
}
```

If the program later accesses *f1*, *f2*, or *f3* without invoking resettx() on the file, the access will fail if another process has modified the file in the meantime.

2.3 Discussion

TX supports a form of transactional synchronization that is similar to the optimistic, timestamp-based algorithms in database systems. In an optimistic scheme, a transaction reads data and records the timestamps, does its computation, and prepares a list of database updates. At commit, the transaction submits its updates together with the timestamps to the database system. If the combination of updates and timestamps satisfies the isolation property of transaction synchronization, the database system commits the transaction. Otherwise the transaction aborts (and restarts).

In a pessimistic lock-based scheme, a transaction T obtains permission (i.e., a lock) on an object before each read or write; the lock precludes any changes in the object due to other transactions. If the lock request conflicts with another transaction's lock, T simply waits until the other transaction releases the lock. In an optimistic scheme, T would proceed directly to commit, whereupon the database system would discover the conflict and abort T. A pessimistic approach may be better for applications in which transactions perform intensive and long-lived computations on database objects, since less work is lost due to abort and restart.

We believe that an optimistic scheme is better than a pessimistic scheme for the OS domain. Commonly, a database system is dedicated to a single application, but OSs usually serve a number of applications at once, so that fairness among applications is an issue. The OS should not allow one application to sequester resources, causing other applications to starve.

Indeed, the Parrot architecture could just as easily support a pessimistic protocol. We could define primitives with which programs explicitly request and release tokens. We believe, however, that an optimistic approach is better. As compared to transaction monitors such as Encina [4, 6], ours is a lighter-weight and simpler approach; the extension to the UNIX file system interface is small and consonant with the rest of the interface.

3 Parrot

Parrot is a file replication service that is layered on top of existing physical file systems; it increases the availability of the underlying systems by transparently propagating the modifications of files and directories from one file system to another. Designed for a dual architecture, the replication service is performed on both file systems symmetrically and with synchronization. As a result, Parrot guarantees file system users a consistent view of data, physically replicated on both file systems. Parrot is implemented as part of an extensible file system, called the Multiple Dimensional File System (n-DFS) [5], that allows services such as versioning, viewpathing, and process monitoring to be added to an underlying system without any modification to the kernel or the applications.

The physical file systems under Parrot may be located on a local- or wide-area network. The applications on each host read from the local copy of a replicated file, and they write to both the local and remote copies. Parrot introduces a logical layer between applications and the kernel. It intercepts an application's system calls that access files and directories, such as read, write, chmod, creat, and truncate, and it passes them to the underlying file system. Moreover, it propagates the calls that modify files to an external server, which in turn executes them on the remote file system.

Parrot supports sequential write sharing. That is, file access under Parrot has exactly the same semantics as if all of the processes on both hosts were executing on a single time-sharing system. In particular, a consistency control mechanism within Parrot maintains and issues read and write tokens on a per-file basis. Note that like the replication function, the consistency control mechanism is transparent to applications. The Parrot code requests the proper token before the process reads or writes the file and releases it after the read or write operation.

To intercept system calls, Parrot replaces the standard C library, libc, with a new library. For systems that support dynamic shared libraries, such as SunOS 4.1 and Solaris, applications may invoke the new library simply by defining the library search path. For systems without dynamic shared libraries, applications must be re-linked with the new library.

Many distributed file systems, such as HA-NFS [2], Ficus [7], Echo [9], and Deceit [12], implement replication mechanisms on file servers embedded in the operating system. Parrot, on the other hand, is implemented as a library that an application links and runs in its own address space. It requires no change in either the kernel or an application: it is transparent to both, and it preserves the syntax and semantics of system calls. Finally, due to the library implementation, Parrot is more portable than those built from scratch, modified from particular kernels, or implemented as new drivers.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[2] A. Bhide, E. Elnozahy, and S. Morgan. A Highly Available Network File Server. In *Proceedings of Winter USENIX Conference*, January 1991.

[3] P. Brinch Hansen. Structured Multiprogramming. *Communications of the ACM*, 15(7), July 1972.

[4] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

[5] G. S. Fowler, D. G. Korn, and H. C. Rao. n-DFS: Multiple Dimensional File System. In W. Tichy, editor, *Trends in Software.* John Wiley & Sons Ltd., 1994.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[7] R. Guy, J. Heidemann, W. Mak, T. Page, G. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *Proceedings of Summer USENIX Conference*, June 1990.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. Technical Report CMU-CS-88-120, Department of Computer Science, Carnegie Mellon University, March 1988.

[9] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and Consistency Trade-offs in the ECHO Distributed File System. In *Proceedings of Second Workshop on Workstation Operating Systems*, September 1989.

[10] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10), October 1974.

[11] M. L. Kazar. Synchronization and Caching Issues in the Andrew File System. In *Proceedings of Winter USENIX Conference*, 1988.

[12] A. Siegel, K. Birman, and K. Marzullo. Deceit: A Flexible Distributed File System. In *Proceedings of Summer USENIX Conference*, July 1990.

[13] A. H. Skarra. Using OS Locking Services to Implement a DBMS: An Experience Report. In *Proceedings of Summer USENIX Conference*, June 1994.

Trademarks