Reader-Writer Locks for Network Attached Storage and Storage Area Networks*

Arun Gandhi, Elizabeth Varki, and Swapnil Bhatia
Department of Computer Science
University of New Hampshire
Durham, NH 03824, USA
Email: {agandhi,varki,sbhatia}@cs.unh.edu

Abstract

Emerging large-scale data-storage technologies such as *Storage Area Networks* (SAN) and *Network Attached Storage* (NAS), being server-less, trade data- sharing and security semantics for high throughput and availability. This paper addresses the issue of data-sharing by presenting a Reader-Writer locking mechanism that is controlled by the storage device. Shared-read and exclusive-write access to shared data is guaranteed by maintaining simple data structures per data-item on storage devices and client nodes. Since requests are issued across the network, all locks are based on blocking. The algorithm presented here is scalable and fault-tolerant.

1 Introduction

Storage systems have evolved from traditional application-level file servers to large, independent, network-aware, disk-array systems such as SANs and NAS. By virtue of being server-less, although these systems can provide high throughput, scalability and transparency, issues such as data- sharing, security and fault management remain unaddressed. Currently, several research groups [1, 2] are working on designing intelligent storage devices capable of handling these tasks. This paper addresses the issue of maintaining coherence of data shared by multiple clients in a distributed environment. Classical reader-writer synchronization allows readers shared access to common data whereas writers must obtain an exclusive-write lock before accessing the shared data. Traditionally, it was the task of the file server to ensure this data coherence. For server-less systems, this task may be shared by clients and the storage device itself.

The placement of locking mechanisms on storage devices is not a new concept. Currently, device locks (d-locks) are implemented on SCSI disks [3] and are used by distributed file systems such as the *Global File System* (GFS) [2]. D-locks are fine-grained locks that are held for relatively small periods. A GFS user must acquire a d-lock in exclusive-write or shared-read mode before accessing storage data. However, d-locks are not scalable and lead to spin-locking which increases traffic on the network. Other existing work on reader-writer locks has been done in the context of shared memory systems. The Mellor-Crummery and Scott (MCS) scheme [4] is a list-based reader-writer locking mechanism where all competing processors spin on a local memory location. Markatos and LeBlanc [5] extend the MCS scheme by using a doubly linked list. In [6], Anderson presents a spin-lock based locking mechanism where processors spin on a remote flag which increases network traffic. Johnson and Harathi [7] present an algorithm where all requesting processes spin on a local flag in a queue while waiting for the lock to be released. In [8], Fu and Tzeng implement a tree of locks to remedy hot-spot contention. Only the holder of the root lock has access to the data.

Prior work on reader-writer synchronization is largely for shared memory environment where the number of users trying to access data are typically much smaller than

*This work was supported in part by NSF under Next Generation Software grant EIA-9974992 and Information Technology Research grant JIS-0082399.

in a distributed environment. Most of the solutions are based on spin locks since the requests are sent across internal buses and the wait time is likely to be short. Issues such as scalability, network delay, network traffic, node failures, and backing-off are not critical in such systems. Unlike distributed systems, fault-tolerance is not a key issue in shared memory environments. Consequently, these approaches are difficult to map to the SAN and NAS environments where thousands of clients on the network could potentially access storage data.

2 The Reader-Writer Locking Scheme

In this scheme, associated with every "data-unit" on a device, is a lock which can be held in *shared-read* or *exclusive-write* mode. A data-unit is implementation dependent and can be of any granularity from a disk block to an entire file. When a writer requests a lock from the device, all readers must relinquish their shared-read lock to implement "writer priority". The device then grants an exclusive-write lock to the writer. The shared-reader locks are long-term while exclusive-write locks are only granted for a requested duration. The locking mechanism is implemented by the device by maintaining separate queues for reader and writer nodes as shown in Figure 1. Simple data structures associated with every data-unit maintain the position of a client in the reader or writer queues as shown in Tables 1 and 2. All messages (except the message granting exclusive-lock to a writer) are forwarded by each node to both neighbors in each direction. Double pointers in each direction protect the locking mechanism from multiple non-adjacent node failures. The size of the data-structure is independent of the number of readers and writers accessing (or waiting to access) a data-unit.

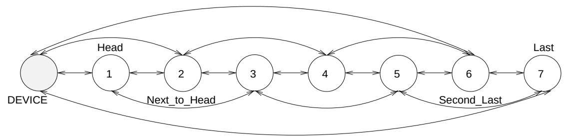

Figure 1: Structure of queues maintained in the scheme.

2.1 The Algorithm

All incoming clients request the device for a read or write lock on shared data. A read lock is granted if there are no writers on-line and a write lock is granted if there are no other readers or writers on-line. If a lock is not granted, the client blocks after joining the tail of the appropriate queue. Regardless of whether the lock is granted or not, the device directs the incoming client to its location in the reader or writer queue. This incoming client then establishes two-way links with its two previous nodes in the queue. Fixed duration write-locks require periodic renewal and are granted higher priority. Read locks are shared, long-term locks which can be held till an exclusive-write lock is requested and thus implement the "delayed-write with call-back" approach. If a write-lock expires, the next writer (if any) or readers are granted access. Table 3 enumerates the messages used in the algorithm. The algorithm assumes that the network is reliable. The steps of the algorithm are explained below. A detailed pseudo-code of this algorithm is presented in [9].

	FIELDS	DESCRIPTION
	Node id: Device Id	The device identification number
READERS	Node id: Head	First in the reader queue
	Node id: Next_to_Head	Second in the reader queue
	Node id: Second_Last	Second last in the reader queue
	Node id: Last	Last in the reader queue
WRITERS	Node id: Head	First in the writer queue
	Node id: Next_to_Head	Second in the writer queue
	Node id: Second_Last	Second last in the writer queue
	Node id: Last	Last in the writer queue

Table 1: Data Structure maintained at the device.

FIELDS	DESCRIPTION
Request Type	READ / WRITE
Node id: Next	Node behind it in the queue
Node id: Next_to_Next	Node next to the one behind it in the queue
Node id: Previous	Node ahead of it in the queue
Node id: Previous_to_Previous	Node previous to the one ahead of it in the queue
Status	For clients requesting write lock, it is either, I. Token_Granted (if no writers in and no readers are presently reading) II. Token_Not_Granted (if writer(s) in or reader(s) presently reading) For clients requesting read lock, it is either, I. No_Writers (if no writers are presently writing or waiting) II. Writers_On_Line (if any writer(s) come in while the readers are reading)

Table 2: Data Structure maintained at every client node.

1. ***When a reader arrives with a read request***

   A new reader, on receiving its position in the queue and the status of its lock request from the device, updates its data structure. It then establishes

MESSAGES	DESCRIPTION
Lock_Request(source_node_id, request_type)	Node requesting a read/write lock from the device
Update_Link(previous, previous_to_previous, next, next_to_next)	Update link to the other node
Device_Informs(status, direction, lock_duration)	Device granting read or write lock to a node, or calling for the read lock from the readers
Readers_Return_Lock(source_node_id)	Reader informs the device that the read lock is returned
Renew_Lock(source_node_id)	Writer renews its lock with the device before the expiration of lock unit time
Writer_Done(source_node_id, next, next_to_next)	Writer returns the exclusive lock to the device

Table 3: Messages exchanged in the algorithm.

two-way links with its (`previous` and `previous_to_previous`) nodes (which also update their own data structures) through an `Update_link` message. Once in the list, the reader either begins to read (`Device_Informs.status = no_writers`) or waits for the lock (`Device_Informs.status = writers_on_line`).

2. ***When a writer arrives with a write request***

   A new writer, on receiving its position in the queue and the status of its lock request from the device, updates its data structure. It them establishes two-way links with its (`previous` and `previous_to_previous`) nodes (which also update their own data structures) through an `Update_link` message. If the incoming client is the second writer, then the device also updates its own (`writers.next_to_head`) field to the node id of the new writer so as to have an alternative link in case the head writer fails. If there are no readers reading, the write lock is immediately granted for a requested duration and must be periodically renewed. If there are readers reading, the write-lock request is blocked and the device sends a `Device_Informs` message to both ends of the reader queue. Alternative links ensure that the message is forwarded to all the readers in the queue. The first reader to receive this message from both its neighbors sends a `Readers_Return_Lock` message to the device which then grants the exclusive-lock to the writer.

3. ***When a reader finishes reading, or backs off, or fails***

   A finishing reader establishes links between its (`previous` and `next_to_next`), and (`previous_to_previous` and `next`) nodes in the queue with an `Update_link` message. Failed readers are discovered and deleted when the queue is next updated.

4. ***When a writer finishes writing, or backs off, or fails***

   A finishing writer returns the write-token to the device and notifies it of its neighbors (`writer.next` and `writer.next_to_next`). The device updates its (`writers.head` and `writers.next_to_head`) fields and grants the write-token to the next writer (if any) making it the head of the writer queue. If there are no more writers, the device invites the readers (if any) to continue reading with a `Device_Informs` message. A lock held by a failed writer will eventually expire and a new lock will be granted to the next writer.

3 An Example

The data structures for each data-unit on the device are initially set to the device id, and at each client node are set to NULL. Consider a scenario as shown in Figure 2. Writer W1 possesses the token to the shared data and renews it periodically. Figure 3 shows the data structures at the nodes W1 and R1, respectively.

	FIELDS	VALUES
	Node id: Device Id	ID #
READERS	Node id: Head	R1
	Node id: Next_to_Head	R2
	Node id: Second_Last	R5
	Node id: Last	R6
WRITERS	Node id: Head	W1
	Node id: Next_to_Head	W2
	Node id: Second_Last	W2
	Node id: Last	W3

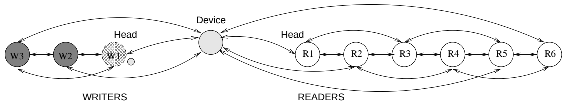

Figure 2: W1 holds an exclusive lock.

When W1 finishes, it returns the token and notifies the device about its neighbors W2 and W3 (`Writer_Done(W1, W2, W3)`). The device updates its data structure and grants the token to W2. The new head (W2) now holds an exclusive lock to the shared data which it renews periodically. Now, W2 fails after renewing its lock. The device waits till lock expiration and then grants a new lock to the next writer, W3.

FIELDS	VALUES
Request Type	WRITE
Node id: Next	W2
Node id: Next_to_Next	W3
Node id: Previous	Device Id
Node id: Previous_to_Previous	Device Id
Status	Token Granted

FIELDS	VALUES
Request Type	READ
Node id: Next	R2
Node id: Next_to_Next	R3
Node id: Previous	Device Id
Node id: Previous_to_Previous	Device Id
Status	Writers_On_Line

Figure 3: Data structures at (i) W1 (ii) R1

While W3 holds the lock, reader R4 fails and R6 decides to back-off as shown in Figure 4. R6 updates the links between its neighbors. The readers now waiting in the queue are R1, R2, R3, R5.

FIELDS		VALUES
	Node id: Device Id	ID #
READERS	Node id: Head	R1
	Node id: Next_to_Head	R2
	Node id: Second_Last	R4
	Node id: Last	R5
WRITERS	Node id: Head	W3
	Node id: Next_to_Head	Device Id
	Node id: Second_Last	Device Id
	Node id: Last	W3

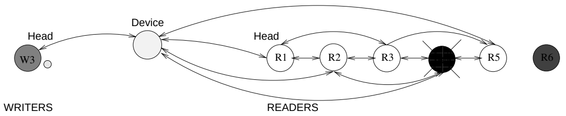

Figure 4: W3 obtains the lock when W2 fails. R4 fails while R6 decides to back-off.

When W3 finishes, it returns the token to the device. Since no more writers exist, the device invites readers to continue reading by sending `Device_Informs` messages to R1, R2, R4, and R5. (A similar mechanism is implemented to inform the readers if a writer comes in while they are reading). On failing to receive the same message from R4 as from R3, R5 concludes that R4 has failed and initiates a recovery procedure which deletes R4 from the queue.

When a reader sees a `Device_Informs` message from both ends, it stops forwarding the message and sends message `Readers_Return_Lock(source_node_id)` to the device notifying that the readers have given up their locks. On receiving this message from a reader, the device now grants the lock to the new writer.

4 Summary

This paper proposes a scalable, fault-tolerant, hot-spot free scheme for data sharing in a server-less environment. Since requests are issued across the network, spin-locks are eliminated and all locks are based on blocking. Locks are assigned to incoming requests based on the order-of-arrival. All the data structures on the device and client nodes have to be maintained on a per-data-unit basis. A data-unit is implementation dependent, but may be of any granularity and can be easily incorporated into an operating system. The algorithm can handle multiple non-adjacent node failures. A drawback of the algorithm is the complexity of list operations. The algorithm is currently being implemented to evaluate its performance.

References

[1] K. Amiri, G. Gibson, R. Golding, “Scalable Concurrency Control and Recovery for Shared Storage Arrays,” CMU-CS-99-111, February 1999.

[2] S. R. Soltis, T. M. Ruwart, M. T. O’Keefe, “The Global File System,” In *Proc. of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, MD, September 1996.

[3] A. Barry, M. O’Keefe, M. Tilstra, “Proposed SCSI Device Locks (and Device Lock Use in the Global File System),” *Technical Report, Version 0.9.5B*, March 14, 2000.

[4] J. M. Mellor-Crummey, M. L. Scott, “Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors,” In *Proc. 3rd ACM Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, pp. 106-113, April 1991.

[5] E.P. Markatos, T.J. LeBlanc, “Multiprocessor Synchronization Primitive with Priorities,” In *Proc. IEEE Workshop Real-Time Operating Systems and Software*, pp. 148-157, 1991.

[6] T.E. Anderson, “The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors,” In *Proc. IEEE Transactions Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 6-16, January 1990.

[7] T. Johnson, K. Harathi, “A Prioritized Multiprocessor Spin Lock,” In *Proc. IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 9, September 1997.

[8] S. S. Fu, N. Tzeng, “A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors,” In *Proc. IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 6, June 1997.

[9] A. Gandhi, S. Bhatia, E. Varki, “A Scalable Solution to the Reader-Writer Problem in a Distributed Environment,” *Technical Report*, Department of Computer Science, University of New Hampshire, May 2000.