Reader-Writer Locks for Network Attached Storage and Storage Area Networkš[1]

Arun Gandhi, Elizabeth Varki, Swapnil Bhatia
Department of Computer Science, University of New Hampshire
E-mail: {agandhi,varki,sbhatia}@cs.unh.edu

Abstract

Network Attached Storage and Storage Area Networks are emerging as the technology of choice for large-scale data storage. A key feature of such storage devices is that they can be directly accessed by applications without the intervention of file servers. Hence, issues such as data-sharing and security cannot be handled by file-servers. This paper addresses the issue of data-sharing by presenting a Reader-Writer locking mechanism that is controlled by the storage devices. The Reader-Writer synchronization refers to the classical problem of shared reader access and exclusive writer access to shared data. The proposed mechanism requires a simple data structure per data-item to be maintained at the 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 simple disks under the control of file servers to large, independent disk-array systems connected directly to the network using Network-Attached Storage (NAS) and Storage Area Network (SAN) technologies. A unique feature of these storage systems is that they are directly accessible by applications running on independent computers without the intervention of file servers. A downside, however, is that tasks such as data sharing, data security, and fault management which are done by host file servers in conventional server controlled storage systems must now be moved to the storage devices themselves. Currently, several research groups [1] [11] [4] [7] [12] are working on designing intelligent storage devices capable of handling some of these tasks. This paper addresses the issue of maintaining the coherence of data shared amongst multiple independent clients in a distributed environment by presenting a scalable, fault-tolerant algorithm for shared read locks and exclusive write locks.

The reader-writer synchronization is a classical problem that is inherent to all computer systems. While readers can share access to common data, writers must obtain an exclusive lock before accessing the shared data. In conventional storage systems, the reader-writer synchronization is handled by file-servers which control all access to storage data. In NAS and SAN environments, the reader-writer synchronization cannot be handled by file servers since clients can directly access storage. Therefore, we propose a locking mechanism that is directly implemented within the storage devices itself. However, the locking mechanism presented here is scalable since hundreds of clients could access a storage device in a NAS or SAN environment. Also, the proposed locking scheme is fault-tolerant and can recover from single node and multiple non-adjacent node failures.

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

The placement of locking mechanisms on storage devices is not a new concept. Currently, device locks are implemented on SCSI disks [3] and are used by distributed file systems such as the Global File System (GFS) [11]. The GFS uses device locks (called d-locks) as its mechanism for ensuring shared file consistency. A GFS user must acquire a d-lock before accessing storage data. The locks can be acquired either in exclusive write or shared read mode depending on the user's request type. When a client sends a read request to the device, it gets the lock immediately if the associated lock is not set in exclusive write mode. (For write requests, the lock must be free.) If the associated d-lock is locked, the request is turned down and a list of current holders of the lock is returned to the client (from the device). The requesting client then sends a 'call-back' to the current holder of the d-lock. When the current holder is done with the d-lock, it flushes its caches to the storage device and releases the d-lock to the next requester after incrementing the version number of the d-lock. The version number is used to inform others that their cached data is old and has to be reread from the device. The d-locks are fine-grained locks which are held for relatively small periods. This leads to spin-locking which increases the traffic on the network. Another drawback of d-locks is that they are not scalable since the client lists can grow very long in a distributed environment. All deadlock detection and avoidance must also be done by the client.

Most existing work on reader-writer locks have been developed in the context of shared memory systems. Mellor-Crummey and Scott [10] present a list-based reader-writer locking mechanism, called the MCS scheme, for a shared-memory system. All competing processors spin on a local memory location without going on the network. Markatos and LeBlanc [9] extend the MCS scheme by using a doubly linked list. In [2], Anderson presents a spin-lock based locking mechanism where processors spin on a remote flag which increases the network traffic. Johnson and Harathi [8] present an algorithm where a queue of requesting processes is maintained by the lock holder. All requesting processes spin on a local flag while waiting for the lock to be released. In [5], Fu and Tzeng implement tree of locks with locks at leaf, intermediate, and root levels. Only the holder of the root lock has access to the data. This algorithm removes the hot spot contention.

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 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 NAS and SAN environments where thousands of clients on the network could potentially access storage data.

In this paper we propose an algorithm to support multiple readers and writers in a distributed NAS and SAN environments. 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 data structures maintained on the device and client nodes are bounded and have to be maintained on a per-data-unit basis. A data-unit is implementation dependent, but may be of any granularity from a disk block to a complete file. Hence, the algorithm is flexible and can be easily incorporated by operating systems.

The rest of the paper is organized as follows: Section 2 presents the proposed locking mechanism. Section 3 discusses an example to illustrate the working of the algorithm. Finally, Section 4 presents the conclusions and future work.

2 The Reader-Writer Locking Scheme

This section presents our reader-writer locking mechanism developed for a distributed environment where client nodes have direct access to the storage devices. Associated with every "data-unit" stored on a device, is a lock which can be held in *shared-read* mode or *exclusive-write* mode. A data-unit is implementation dependent and can be of any granularity from a disk block to an entire file. The locking mechanism is controlled by the device storing the data-unit. Thus, all clients have to get a lock from the device before they can access the corresponding storage data. The mechanism presented here is based on the "writer priority" approach. As soon as a writer requests a lock from the device, all readers are informed and have to relinquish their shared read lock, at which point the device gives an exclusive-write lock to the writer. The shared-reader locks are long-term locks while exclusive-write locks are only given for a specified time duration.

The locking mechanism is implemented by maintaining separate queues for reader nodes and writer nodes. The queues are not stored at the storage device or at the client nodes. Rather, devices and client nodes maintain simple data structures associated with every data-unit. Figures 1 and 2 show the data structure maintained on each device and at each client node, respectively. The data-structures essentially maintain the position of a node in the reader or writer queue. The device is at the head and tail of both the reader and writer queues and keeps pointers to the first and second reader, the last and second-to-last reader, the first and second writer, and the last and second-to-last writer. Each client node maintains pointers to its next, next-to-next, previous, and previous-to-previous node. An example of a queue is shown in Figure 3. All messages (except the message granting exclusive-lock to a writer) are forwarded by the sending node to its neighbor (*next* or *previous*) and its neighbor's neighbor (*next_to_next* or *previous_to_previous*) depending on the direction of the propagation). The reason for maintaining these double pointers in each direction is to protect the locking mechanism from node failures. A queue will only get disconnected if 2 adjacent nodes in the queue go down. Hence, the mechanism is fault-tolerant, a key feature in a distributed algorithm. Another feature of the mechanism is that the size of the data-structure is not dependent on the number of readers and writers accessing (or waiting to access) a data-unit making the algorithm scalable and space efficient.

2.1 The Algorithm

All incoming clients request the device for a read or write lock to the requested data item. A read lock is granted if there are no writers on-line and a write lock is granted if there are no other readers and writers on-line. Regardless of whether the lock is granted or not, the device directs the incoming client to its location in the respective reader or writer queue. This incoming client then sends a message notifying both the client nodes, previous and previous_to_previous to it in the queue of its entry into the queue. These clients on receiving the message from this new client update their data structures. If a lock is not granted to the incoming client, it blocks after joining the tail of the appropriate queue.

	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

Figure 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)

Figure 2: Data Structure maintained at every client node.

Since the reader lock is shared, they are long-term locks and a reader can hold on to it till a writer arrives and requests an exclusive lock. The algorithm gives priority to writers, and all readers have to give up their lock once a writer requests a lock. As soon as a writer arrives, the device sends a message to the readers along all its reader links (i.e., *Reader.head, Reader.next_to_Head, Reader.Second_last, Reader.Last*) informing them of the writer. The message is propagated by the readers in the appropriate direction. The node which sees this message from both ends stops propagating the message any further and sends a message to the device informing it that all readers return their lock. This bidirectional propagation of the call-back message reduces the time it takes for the messages to reach all the readers. When the device receives the returned-lock message, it grants an exclusive lock to the writer. Thus, the proposed scheme implements the "delayed-write with call-backs" approach. The exclusive write lock is only granted for a specific time period. It is the responsibility of a writer client to periodically renew its lock before the expiration of this time period. If the lock is not renewed within the time period, the device grants the lock to the next writer in the queue. If there are no writers, the lock is granted to the readers.

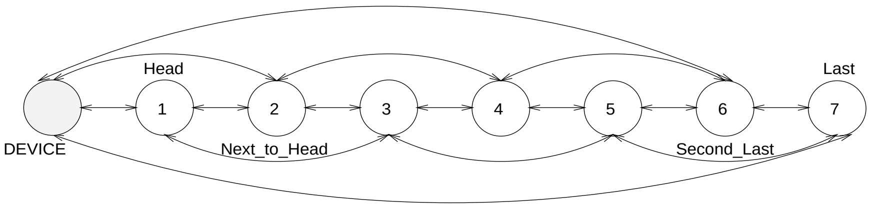

Figure 3: Structure of the queues maintained by the algorithm.

Message	**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 read lock is returned
Renew_Lock(source_node_id)	Writer renews its lock with the device before the expiration of lock time unit
Writer_Done(source_node_id, next, next_to_next)	Writer returns the exclusive lock to the device

Table 1: Messages exchanged in the algorithm.

Thus, all locks in the proposed scheme are based on blocking. All incoming clients request a lock from the device. The device informs the client of its position in the appropriate queue. After all the links in the queue are updated, the process in the client node will block if the lock is not granted. The client process will unblock when it gets the lock from the device. Table 1 enumerates the messages used in the algorithm. The algorithm assumes that the network is error-free and messages that are sent will be received. A detailed presentation of the algorithm, including the messages exchanged between the devices and nodes, is now given.

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

   The device notifies the new reader its position in the queue and the status of its lock request. This new reader updates its data structure and links up to the current tail nodes (*previous* and *previous_to_previous*). The node sends *Update_link* to its *previous* and *previous_to_previous* nodes. Both the nodes ahead of it in the queue too now update their data structures. The reader, once in the list, depending upon the status of its request either begins to read (*Device_Informs.status = no_writers*) or is waiting (*Device_Informs.status = writers_on_line*) to get a lock.

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

   The device notifies this new writer its position in the queue and the status of its lock request. The writer updates its data structure and links up to the current tail nodes (*previous and previous_to_previous*). The node sends *Update_link* to its *previous* and *previous_to_previous* nodes. If the incoming client is the second writer, then the device also updates its own field (*writers.next_to_head*) to the node id of the writer. Thus, the device always has an alternative link in case the head writer goes down.

   If the incoming client is the first writer and there are no readers reading, the write lock is immediately granted for a time period. It is the writers responsibility to periodically renew the lock before it expires. If there are readers reading, the write lock request is denied. However, the device, sends a *Device_Informs* message to the readers (along all its reader pointers) informing them that a writer is waiting and the lock should be returned. The first reader to get the *Device_Informs* message from both its left and right neighbors sends the *Readers_Return_Lock* message to the device. The device then grants the exclusive lock to the writer.

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

   The reader before leaving sends a message to the nodes ahead (*previous and previous_to_previous*) and behind (*next and next_to_next*) in the queue thereby updating the links. If a reader goes down, the queue is still maintained due to the presence of the double links in each direction. However, the reader queue has to be updated every time period to eliminate pointer to the failed nodes.

4. *When a writer finishes writing, or back off, or goes down*

   The writer returns the writer token to the device and notifies the device of the nodes behind it in the queue (*writer.next and writer.next_to_next*). The device updates its own fields (*writers.head and writers.next_to_head*) with these values. The device gives the token to the next writer (if any) in the writer queue making it the head of the queue. If it was the last writer, the device informs the readers (if any) that the writers are out (*Device_Informs*), and the readers can now continue to read. If a writer goes down while holding the lock, it will not renew the lock and the device grants the lock to the next node.

Summarizing, the key features of the algorithm are its scalability, fault-tolerance, and blocking locks, which makes the mechanism suitable for distributed environments. For the pseudo-code of the algorithm, an interested reader is referred to [6]. An example illustrating the working of the algorithm is presented in the next section.

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. Now consider a scenario where there are several readers and writers wanting to access the data-unit while the first writer is holding the lock as shown in Figure 4. Writer W1 has the token and is presently accessing the data-unit. The device grants this token only for a unit of time. W1 periodically renews the lock (*Renew_Lock(W1)*) before the expiration of the lock. The other writers wait in the queue for their turn. Figure 5 shows the data structures at the nodes W1 and R1, respectively.

	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

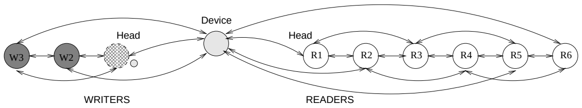

Figure 4: Writer W1 holds an exclusive lock while the other writers and readers are waiting

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 5: Data structures (i) At the node W1 (ii) At the node R1

When the writer W1 is done, it returns the token and notifies the device of the nodes W2 and W3 behind it in the queue (*Writer_Done(W1, W2, W3)*). The device updates its data structure with these values and then grants the token to the next writer W2 in the queue (*Device_Informs(token_granted, null,* t_{L2}*)*), making it the head of the queue. The new head, W2, now holds an exclusive lock to the shared data-unit for a time t_{L2}. Again, W2 periodically renews the lock (*Renew_Lock(W2)*) with the device before the expiration of time period t_{L2} until it gets done.

Now, suppose writer W2 goes down after renewing its lock. The device waits for the lock duration. Since W2 is down, there will be no further lock renewals nor will the device receive any message from W2. Hence, the device grants the token to the next writer, W3, in the queue (*Device_Informs(token_granted, null,* t_{L3}*)*).

While the writer W3 holds the lock, reader R4 goes down. Also, after waiting in the queue for a long time, reader R6 decides to back-off as shown in Figure 6. Before backing off from the reader list, R6 updates the links of the nodes. Node R6 sends an *Update_Link* message with *(Update_Link.next = device id)* to node R5 and with *(Update_Link.next_to_next = device id)* to node R4. Also an *Update_Link* message with *(Update_Link.second_last = R4)* and *(Update_Link.last = R5)* to the device. The number of active readers now waiting in the queue are reduced to four (R1, R2, R3, R5).

Now when the writer W3 is done, it returns the token to the device (*Writer_Done(W3, null,*

	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

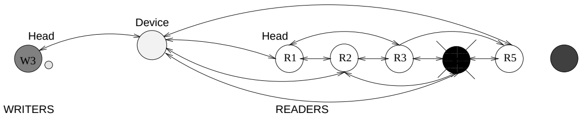

Figure 6: Writer W3 gets the lock after the W2 goes down. Reader R4 fails while reader R6 decides to back-off.

null)). The device knows this was the last node in the writer queue. It now sends a message to the readers R1 & R2 (*Device_Informs(no_writers, last)*) and R4 & R5 (*Device_Informs(no_writers, head,)*), informing that there are no writers in and the read lock is granted. The message is sent to the start and the tail nodes, and is propagated in either direction by the interim nodes depending upon direction attribute in the message. This forwarding of message by the intermediate nodes ceases once it reaches a reader in the list which has already seen the message from the other direction. Thus, the time taken to inform the readers that the read is granted is reduced.

A similar mechanism is implemented to inform the readers if a writer comes in while they are reading. When a reader sees the 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 Concluding Remarks

This paper presents a locking mechanism that addresses the Reader-Writer synchronization problem for the NAS and SANs technologies. The mechanism is based on the concept of shared read and exclusive write locks associated with each data-unit on the storage device. The data-units themselves are implementation dependent and can be of any granularity. The locks are controlled by the device. Simple data structures maintaining the position of a node in the reader or writer queue are kept by each node.

The algorithm presented here is scalable and space efficient since the complexity is not dependent on the number of readers and writers accessing (or wanting to access) the data-unit. The algorithm can handle multiple nonadjacent node failures due to the queues maintaining double links in both directions. All locks are based on blocking, so there is no spinning or busy waiting at local or remote flags. There is no hot spot contention since minimal messages are exchanged with the device. A drawback of the algorithm is the linked list maintenance. Since reader locks are long-term locks, the reader queues must be updated after a time period to ensure that failed nodes are removed from the queue. However, the presence of two pointers in each direction makes this task relatively easy. We are currently implementing this algorithm to evaluate its performance.

References

[1] Khalil Amiri, Garth Gibson, Richard Golding, "Scalable Concurrency Control and Recovery for Shared Storage Arrays", CMU-CS-99-111, February 1999.

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

[3] Andrew Barry, Matthew O'Keefe, Mike Tilstra, "Proposed SCSI Device Locks (and Device Lock Use in the Global File System)", *Technical Report, Version 0.9.5B*, March 14, 2000.

[4] EMC Corporation,
<http://www.emc.com/products/product_pdfs/wp/celerra/celerra_arch_high_avail.pdf>

[5] Shiwa S. Fu, Nian-Feng Tzeng, "A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 6, June 1997.

[6] Arun Gandhi, Swapnil Bhatia, Elizabeth Varki, "A Scalable Solution to the Readers-Writers Problem in a Distributed Environment", *Technical Report*, Department of Computer Science, University of New Hampshire, May 2000.

[7] Hewlett-Packard Company,
<http://www.enterprisestorage.hp.com/pdfs/bldgasan.pdf>

[8] Theodore Johnson, Krishna Harathi, "A Prioritized Multiprocessor Spin Lock", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 9, September 1997.

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

[10] John M. Mellor-Crummey, Michael L. Scott, "Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors", In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, pp. 106-113, April 1991.

[11] Steven R. Soltis, Thomas M. Ruwart, Matthew T. O'Keefe, "The Global File System", In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, MD, September 1996.

[12] VERITAS Software Corporation,
<http://eval.veritas.com/webfiles/docs/sandirect_whitepapers.doc>