Distributed Queue-based Locking using Advanced Network Features

Ananth Devulapalli
Ohio Supercomputer Center
1 South Limestone St., Suite 310
Springfield, OH 45502
ananth@osc.edu

Pete Wyckoff
Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

A Distributed Lock Manager (DLM) provides advisory locking services to applications such as databases and file systems that run on distributed systems. Lock management at the server is implemented using First-In-First-Out (FIFO) queues. In this paper, we demonstrate a novel way of delegating the lock management to the participating lock-requesting nodes, using advanced network primitives such as Remote Direct Memory Access (RDMA) and Atomic operations. This nicely complements the original idea of DLM, where management of the lock space is distributed. Our implementation achieves better load balancing, reduction in server load and improved throughput over traditional designs.

1 Introduction

Locking is essential for serialization of access to shared resources. In an uniprocessor environment, this is achieved using services such as mutexes, semaphores and monitors [7] that rely on the centralized management provided by the operating system. In shared-memory multiprocessor systems, the architecture frequently provides mechanisms to extend this centralized locking scheme across the domain of the machine. Distributed multiprocessor systems, however, rarely use networks that provide such features and hence resort to using a dedicated process to provide synchronization, called a Distributed Lock Manager (DLM).

DLM is a critical component in many applications that run on distributed systems, such as databases and file systems. Today, many vendors like Oracle [10], IBM [2] and HP [17] provide DLM as an integral component in their systems. Current DLM implementations achieve greater scalability and fault-tolerance over single-site lock serving models by distributing locks among all nodes in the cluster. Each node serves as a server for a subset of locks and as clients for the rest. A server uses FIFO queues for serializing requests for a shared resource.

This work was supported by the US DoE ASC program.

In this paper we show how to further augment load-balancing by distributing the queue-management among the lock-requesting clients. We focus particularly on the use of DLM in non-shared-memory distributed systems and we take advantage of atomicity features provided by modern high-speed communication networks. We start by explaining the current issues that motivate our work in Section 2. In the next section we explain the abstract design of how we distributed the queue-management. Then we explain our implementation of distributed FIFO queue using Remote Direct Memory Access (RDMA) based primitives. In Section 5 we describe the experiments and follow up with their results and analysis.

2 Motivation

In this section we describe traditional DLM designs and advanced capabilities of modern interconnects that motivate our work.

2.1 Locking in distributed systems

In multi-processor distributed systems, processes frequently need to share global resources such as files, memory buffers and access to storage or network devices. One approach to achieve synchronization among these processes is to concentrate the management of locks at a single site. Any node needing to lock a resource sends a request to the centralized manager. While this approach is simple, it suffers from two critical disadvantages, namely the presence of hot spots in the network and a single point of failure at the manager node. In the case of loosely coupled clusters that generally have higher latency networks and low-cost and hence low-reliability nodes, these problems are aggravated. As a result it is difficult to implement high performance shared services like file systems and database servers that rely on this model of centralized lock management.

As an alternative, lock management responsibility can be distributed across multiple nodes in two different fashions: various independent locks can be managed at different nodes, and aspects of the lock management protocol for a single lock can be handled by multiple nodes. The first aspect of spreading the management of multiple locks across

nodes has been implemented many times [3, 17, 10, 2]. By distributing lock management among multiple nodes, load is spread throughout the system. The topic of this paper, however, is the second aspect of distributed lock management, namely removing some of the centralized aspect of the management of a *single* lock.

2.2 Advanced interconnects

Within the last decade many new and interesting networking technologies have been implemented. VIA [8], Myrinet [22] and InfiniBand [14] introduced some of the advanced networking primitives that we see today.

2.2.1 Remote Direct Memory Access (RDMA)

RDMA is an efficient means of communication. It is asynchronous, involves minimal operating system overhead, transfers data with zero intermediate copies and consumes fewer CPU cycles, both at the sender and the receiver. To achieve these capabilities, intelligent communication coprocessors are required. Today RDMA appears in a variety of networks [8, 22, 14]. It can be considered analogous to DMA on a peripheral bus such as PCI, in that this capability has been extended to remote node. As DMA absolves the CPU of any involvement in memory transfers, RDMA removes it from the critical path of data transfer, thereby achieving lower latency and higher bandwidth.

2.2.2 Remote atomics

Specialized hardware primitives to assist in the manipulation of concurrent objects in multiprocessor systems have appeared in many different systems and in different forms, either in the processor instruction set [13, 23] or in an associated network component [6]. The classic primitives include atomic registers, *test&set*, *compare&swap*, and *fetch&add*. Algorithms to use these primitives to implement various synchronization have been well studied, for example, simple mutual exclusion, queuing, barriers, and wait-free versions of the same [6, 11, 12].

The interconnect used to perform communications required for concurrent operations is theoretically independent of the operations themselves, and can range from a tightly-coupled bus in a shared memory system to a general purpose wide-area network. Our focus in this work is to consider lock management strategies as implemented using modern networks that provide remote atomic primitives in hardware. In particular, InfiniBand offers both *compare&swap* and *fetch&add* operations with which we construct distributed queue algorithms.

2.3 Traditional DLM architecture

A DLM provides "advisory locking" services to higher level applications [25], meaning applications must cooperate in sharing the resource. Any node interested in locking an entity must first get a lock on a corresponding "lock resource". A lock resource is an abstract entity, identified by a descriptive name and corresponds to exactly one physical entity, for example a file. The management of locks is distributed among the nodes within cluster by distributing the lock namespace.

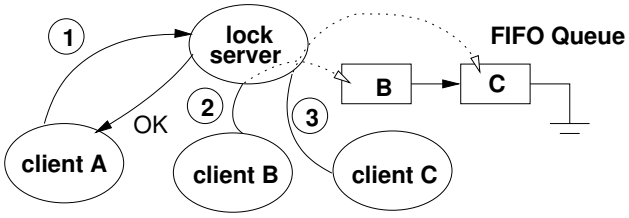

Figure 1. Locking Operation in DLM.

Figure 1 shows the process of locking in DLM. Client *A* is the first one to request the lock. Since the resource is unlocked, the server grants the lock to *A*. Then, clients *B* and *C* arrive with a request for same lock. But the resource is locked, so the server places them in a FIFO queue of waiting clients. When *A* is ready to unlock, it sends an unlock message to server, which upon its receipt, de-queues the first client, in this case *B*, and grants it the lock.

We described a simple version of DLM. Current implementations have many features, like multi-modal locks with conversion between different modes, hierarchical locks and fault-tolerance. But the fundamental concept in all the implementations, is that the lock manager achieves serialization by arranging the nodes in a queue. FIFO queues serialize access to the resource thereby breaking any deadlock cycles. It is important to note that these queues are per lock and queues are not shared among the locks. Also, these data structures are internal to the lock manager.

2.3.1 Critique of the current DLM design

Traditional DLM designs do not take advantage of modern network features, relying only on reliable message delivery with channel semantics. As a result, an active process is always required for lock management, which essentially means handling a queue of blocked processes. Another drawback is that locks are statically distributed, which may not necessarily match the dynamic application load, thereby creating hot-spots during execution. We can solve these problems by delegating the load of lock management to the lock requesting nodes themselves.

The target of our research is the distribution of the FIFO queue, thereby distributing the load. Note that this approach to distribute the data structure (queue), representing the state of a single lock, complements earlier DLM work that successfully distributed the lock space. Our work allows for more concurrency in the locking and unlocking algorithms and decentralizes the work to the lock-requesting clients, improving scalability.

3 Design of distributed FIFO queue

In this section we describe the essential design requirements and an abstract design for distributed FIFO queue based locking mechanism.

3.1 Design requirements

One advantage of traditional DLM designs is that the server has a global view of the state. This enables the server to send an acknowledgment to the requesting process, only when the server is ready to grant the lock. This results in minimum network traffic. Since this luxury is no longer available in the case of a distributed FIFO queue, care must be taken to minimize network traffic.

A FIFO queue by itself guarantees progress, but it does not guarantee freedom from starvation. Any design should try to allocate locks as fairly as possible.

In the case of a distributed FIFO queue, processes will be acting independently, trying to acquire and release locks. Effort must be made to minimize the state communication during each operation.

Since the reason for distributing the management of locks is reduction of server load, this must be achieved. The operation load should be shared among the participating processes. This also means that non-participating processes should be precluded from sharing the burden.

3.2 Distributing the FIFO queue

In the original implementation, which we will refer to as the "Basecase", the server needs to keep information about the current tail and head of the queue. To implement a distributed queue, one can use the fact that it is a *FIFO queue*. Each node in the queue needs information about its predecessor and successor nodes. To release the lock, the owning node sends a message directly to its successor; meanwhile, that node is waiting for a message from its predecessor, and so on down the queue. It is possible for a new node to get the information about its predecessor from the server. But it is not possible for a node to know about its successor beforehand, since there might not be a successor in the first place. The only way a node gets to know about its successor, is when the successor informs it about its presence.

3.2.1 Lock acquisition

For this chaining of the nodes to be possible, the only information that has to be maintained at the lock-server is the current tail of the queue. Figure 2 shows the interactions among three clients in the process of acquiring a single lock from a lock server. In step 1, client *A* queries the server about the current tail on the queue for the lock, and tries to install itself as the new tail. Because the lock is currently not held by any client, this operation succeeds and now *A* owns the lock. Next, client *B* attempts to acquire the lock, setting itself as the current tail at the server, but it must wait until *A* is finished with the lock. To obtain notification when this happens, *B* updates a value in the memory of *A* that declares itself as the next entry in the queue of waiters for the lock. Steps 4 and 5 show a third client also failing to acquire the lock and queuing itself behind the previous last waiting client.

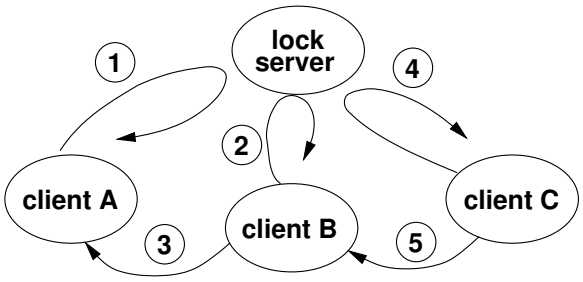

Figure 2. Three clients acquire a lock.

The effect of these operations is that all the processes are serialized in a FIFO queue that is no longer contained and managed at the server. The management load is now distributed among the clients. It is to be noted that only those clients that are participating in the locking process simultaneously are involved in queue management.

3.2.2 Releasing a lock

Figure 3, shows the interactions during a series of unlock operations. Client *A* finds that it has a successor and sends a grant message to it, passing lock ownership to client *B*. Similarly, *B* unlocks by transferring the lock to client *C*. As no other clients are waiting on the lock, *C* has an empty successor field. It updates the tail at the lock server to indicate that the lock is no longer held by any client.

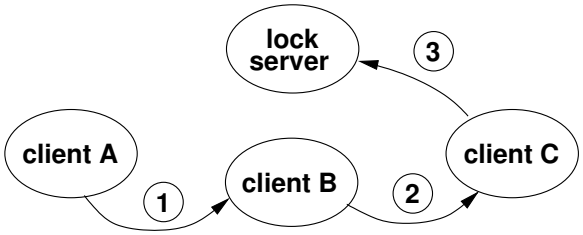

Figure 3. Passing lock ownership.

Compared to the Basecase, by distributing the queue among the processes, we have achieved distribution of the load. Each node takes care of informing the predecessor and successor, if present. The server simply fields requests that read and update the tail. In particular, the server is not involved at all in the unlock phase until the last waiting client has finished with the lock and finally clears the tail at the server.

4 Implementation

In the previous section, we laid out the basic framework for the design of distributed management of a FIFO queue.

In this section we describe the mapping between those abstract ideas and our implementation of FIFO queue based locking.

4.1 Enabling RDMA operations

In Section 3.2.1, we described how the distributed FIFO queue is constructed. By distributing the queue, the load at the server has been effectively reduced by half, since dequeue operations have now been off-loaded to the clients. However, further off-load is possible. This is where the advanced network primitives: RDMA and Remote atomics come into play.

To perform network operations that affect the memory of remote hosts, multiple pieces of information are required to ensure secure operation. In InfiniBand, an RDMA write requires the remote LID, QP number, virtual memory address, and a 32-bit opaque key [14]. These values are, however, static for the lifetime of a particular client, so we employ a caching mechanism at each client to reuse the values after a one-time initial lookup. As each client joins the lock domain, it registers these remote access keys with the lock server that in turn exports them to all other lock clients. If any cached value becomes invalid due to an exiting or initializing client, the network will return an error on the next attempted operation, invalidating the local cache and forcing a full lookup of the new values.

4.2 Lock acquisition

The state information in the tail must include the *identity* of the client that is currently at the tail of the queue. This information is sufficient for a client to know about its predecessor.

For lock acquisition we use atomic RDMA *compare & swap* (*CAS*) primitive. *CAS* [14] takes two 64-bit operands: *compare* and *swap*. The *compare* operand is compared with the value at remote address. Only if they match, the *swap* operand is written to the remote memory location. In either case, the original value at remote address is returned back as the result of the operation.

A client interested in a lock uses *CAS* to swap its identity into the tail atomically. If it succeeds, it has locked the resource. Otherwise, some other client has the lock. This client's identity is returned as a result of *CAS* operation. At this stage, the client does not know whether the current tail is the owner of the lock, or waiting for the lock. In any case, that tail would be this client's most likely predecessor. Therefore, the client tries to swap the current tail with itself. If it succeeds, it has the information about its predecessor. Otherwise it repeats the steps of swapping itself with the tail, until it is successful.

Once a client knows about its predecessor, it sends an alert to it, so that the predecessor is informed about it. It uses RDMA *Write with Immediate* (*RWI*) [14] to signal the predecessor. Since the tail is swapped atomically, each client will have at most one predecessor. Likewise each client will have at most one successor.

4.3 Unlock operation

When a client is ready to unlock, it looks for its successor message. If it finds the message, it sends an alert to the successor using *RWI* thus transferring the lock in the process. Otherwise, since this client does not know about the current state of the tail, it tries to get an update on the tail by swapping itself out with some value that stands for the "unlocked" state. Again, *CAS* is used for this operation. If it succeeds in swapping, it means the resource has been unlocked. For example, in Figure 3, when *B* is ready to unlock, it looks for its successor and finds a message from *C*. *B* then sends an alert using *RWI* to *C*. When *C* is done with the lock, it looks for its successor and finds it empty. Finally, *C* executes *CAS* against the tail and unlocks the resource.

4.3.1 Transient State Race Condition

We need to take care of a transient state during the unlock operation. Again, let us refer to Figure 3, when *B* is ready to unlock. Assume that the moment *B* is looking for its successor, *C* is busy trying to *CAS* the tail. Therefore, *B* thinks it is at the end of the tail and tries to unlock the resource, but finds *C* in the tail instead. Since *B* is no longer at the tail of the queue, it waits for a message from its successor. Also, *B* does not know whether *C* is its successor. The only thing that is certain from *B*'s perspective is that, it is no longer the tail. Therefore, it waits for an alert from its successor. As soon as it gets an alert, it sends an alert back to it, which in this case happens to be *C*.

4.4 Starvation freedom

We have a basic framework for distributing FIFO queue but there is no starvation freedom. The reason for this is *non-atomicity* of lock acquisition. The first client is able to acquire the lock atomically by executing *CAS* against the server. But subsequent clients repeatedly execute *CAS* operations until they are successful. For a slow client, these unbounded attempts can potentially lead to starvation.

We solved this problem, by an approach similar to the bakery algorithm [18]. Each client interested in updating the tail first gets a token from the server. Only when it is the client's turn, it gets a chance to replace the tail. A client interested in updating the tail, gets a number by executing atomic RDMA *fetch & add* (*FA*) against an always-increasing counter. *FA* is similar to *CAS*. *FA* [14] takes one 64-bit operand as its input and executes an unsigned addition with the value at remote location. The new value is stored in the location and the old 64-bit value from the location is returned as the result of this operation.

After executing *FA* against the counter, the client discards the most significant 32 bits of the result and keeps the least significant word as its counter. Since the remote

counter is always incremented, this counter will always be changing. Then the client constructs its 64-bit token by appending its *identity*, to its 32-bit counter. It uses this 64-bit token to swap into the tail. Only the client whose counter value is 1 greater than the current tail is allowed to swap into the tail. All other clients, whose counter is further away, will have to wait, for this client to proceed. This serialization ensures that there is no starvation. Every client gets a chance at replacing the tail. But it adds the cost of one additional remote atomic operation to the critical path of locking.

5 Experiments

5.1 Experimental setup

Our experiments were carried out on 32 nodes of Pentium 4 cluster at the Ohio Supercomputer Center. Each node has dual 2.4 GHz Intel P4 Xeon processors, 4 GB of RAM, 80 GB ATA100 hard drives, one 10 Gb/s InfiniBand interface and runs 2.6.6 version of Linux kernel. One of the 32 nodes was designated as lock-server and the rest as clients. For the experiments, we evaluated the following three algorithms:

Basecase: The original implementation of DLM, where the FIFO queue is managed by an active server.

Distributed: Distributed FIFO queue algorithm without starvation-free mechanism.

Starvation-free: Distributed FIFO queue algorithm with starvation-free mechanism.

In Basecase, communication between server and clients was implemented using InfiniBand *Send* and *Receive* primitives. Distributed and Starvation-free used *Send* and *Receive* for an initial look-up for lock information. For all other cases, they used RDMA based primitives as described in Section 4.

5.2 Operation latency

Latency for each primitive was timed in this experiment. Table 1 shows the median, mean and standard deviation of the timings. RDMA CAS, RDMA FA and *Send* were measured when a client executed those operations against a server. The size of send message was 100 bytes. RDMA Write Imm was timed between two clients without involving the server, since in distributed FIFO queue, *RWI* was used to send alert messages between clients.

	Median (μs)	Mean (μs)
RDMA CAS	11.005	11.455 ± 1.652
RDMA FA	11.027	11.467 ± 1.653
RDMA Write Imm	10.997	11.802 ± 2.141
Send	13.453	14.197 ± 2.183

Table 1. Average timings for primitive operations.

The numbers listed are higher than the numbers others have reported [9]. This is because we implemented each primitive to signal the host upon completion, which adds non-trivial overhead. The DLM application needs message delivery confirmation because of the dependencies of future actions on the past messages. The confirmation of the message delivery triggers the state change in the implementation.

	Lock (μs)	Unlock (μs)
Basecase	34.045 ± 5.388	32.818 ± 3.489
Distributed	11.889 ± 1.720	12.616 ± 1.629
Starvation-free	24.070 ± 3.140	12.350 ± 1.647

Table 2. Mean lock and unlock timings.

	Lock	Unlock
Basecase	$T_{send} + T_{oh} + T_{send}$	$T_{send} + T_{oh} + T_{send}$
Distributed	$T_{cas} + T_{oh}$	$T_{cas} + T_{oh}$
Starvation-free	$T_{fa} + T_{cas} + T_{oh}$	$T_{cas} + T_{oh}$

Table 3. Lock and unlock cost models.

In the second experiment, we measured uncontested, single client, lock and unlock operations against an unloaded server for each algorithm. The timings are shown in Table 2 and the corresponding cost models are listed in Table 3. Basecase uses *Send* for both lock and unlock messages, hence they have equal cost. Distributed uses *CAS* for both lock and unlock operations. Starvation-free also uses *CAS* for unlock, but it has a *FA* in critical path for lock operation. We note that the models explain the observed costs in Table 3, in terms of timings of primitive operations (Table 1).

5.3 Unlock cascade

In the scenario that one client has locked a resource for long enough that others have piled up behind it waiting to do comparatively short operations with the same lock, how long does it take for them to resume once the lock holder releases it? We conducted the experiment by making one client hold onto the lock long enough so that other clients pile up behind it. Next this client starts the timer, releases the lock and immediately sends a lock request. Other clients in the meanwhile are queued up behind this client waiting for their turn at the lock. Upon getting the lock, these clients immediately release the lock. So the lock propagates along the chain of queued processes finally arriving back at the first client, which stops the timer as soon as it gets the lock. The cost of operation, as observed by first client is $T_{cascade} \approx n * T_{transfer}$, where n is the number of clients.

In Basecase, $T_{transfer} \approx 2 * T_{send} + T_{oh}$. One send operation is consumed by an unlocking client and another is consumed by a grant message from the server to the next client. T_{oh} represents the processing cost at the server and clients. In case of Distributed and Starvation-free, $T_{transfer} \approx T_{rdma\_write\_imm} + T_{oh}$, since *RWI* is used to alert the successor. Here, T_{oh} represents the cost of a client looking for its successor's message, the cost of the successor processing an alert from predecessor and the final delivery of message to the user application.

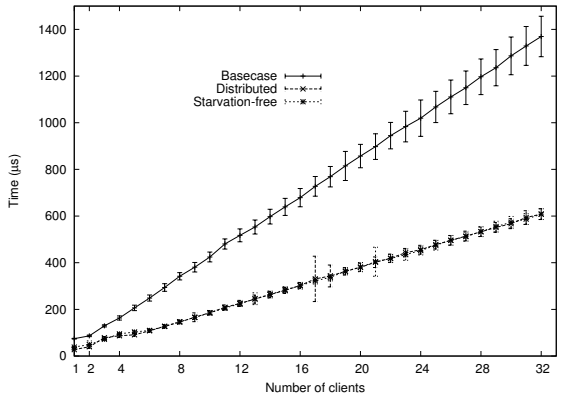

Figure 4. Time to propagate a lock along a queue of waiting clients.

Figure 4 plots the time to propagate the unlock request as the number of clients vary. The curves for Distributed and Starvation-free are overlapped since unlocking steps are similar in both cases. The slope of Basecase is twice that Distributed and Starvation-free, as explained by the cost model of cascade. Also, from the slopes it is evident that in Basecase $T_{oh} \approx 12\mu s$ and in case of Distributed and Starvation-free $T_{oh} \approx 6\mu s$. This experiment demonstrates the effectiveness of load-balancing in our design.

5.4 Throughput under different contention conditions

The scalability of different algorithms under different contention levels was tested under this experiment. Contention is defined as: $Contention = 1 - \frac{L}{N}$, where L is number of locks and N is number of clients. Three levels of contentions were used: 0%, 50% and 100%. For 0%, $L = N$, for 50%, $L = \frac{N}{2}$ and for 100%, $L = 1$. One node was designated as server, with number of clients varying between 1 and 32. The clients executed lock and unlock operations against server without any work between operations. Figure 5 shows the "cumulative throughput" of all clients for different contention levels and varying number of clients.

For 0% and 50% contention level, all curves reach a plateau. In 0%, Basecase, Distributed and Starvation-free level at 2, 4 and 4 clients respectively. Basecase is server-bound, while others are NIC-bound. In Basecase, the curve falls slightly as the number of clients increase due to increase in server processing overhead. Similar trends are also seen in 50% case. In case of Distributed, throughput is approximately *half* of 0% case since only half the attempts are successful. For Starvation-free, throughput drops by 25%. If we refer to cost model in Table 3, compared to 3 operations in 0% case, we have an additional unsuccessful *CAS* due to contention which results in 4 operations. For Basecase, the throughput *increases* relative to 0%, because server is able to overlap acknowledgment of "unlock" and "grant" messages, which was not possible in 0% case.

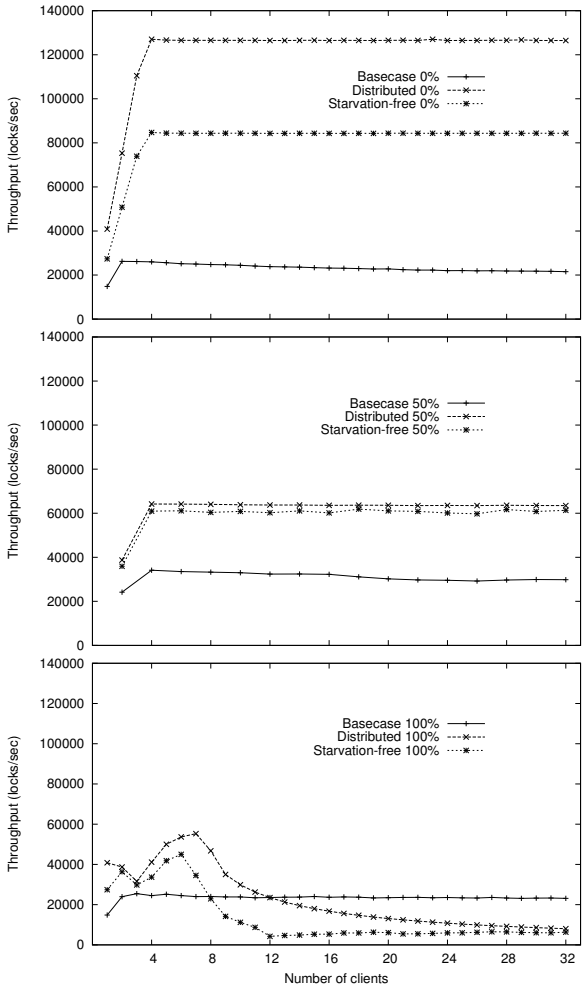

Figure 5. Lock throughput for varying number of clients at three different contention levels.

We observe interesting trends in 100% case. Basecase reaches a plateau and stays there, whereas Distributed and Starvation-free have bell-shaped curves. The curves drop at 3 clients due to an increase in number of unsuccessful *CAS* operations executed by lock-owner due to transient state as discussed in Section 4.3.1. Then with increase in number of clients, the chance for a client to enqueue increases, resulting in increase of successful *CAS* operations. The curves peak and then fall for different reasons. Distributed falls due

less per-capita *CAS* operations and an increase in failed *CAS* operations due to increase in transient states. Starvation-free falls more steeply. Since only the successor is allowed to replace the tail, *CAS* operations of remaining clients fail. But in the process they hold back the successor, while the predecessor seeing no successor unlocks the resource. This means there is no chance for queue-formation resulting in serialized lock-unlock operations by individual nodes. Both cases suffer since predecessor releases lock too quickly for successor to queue behind.

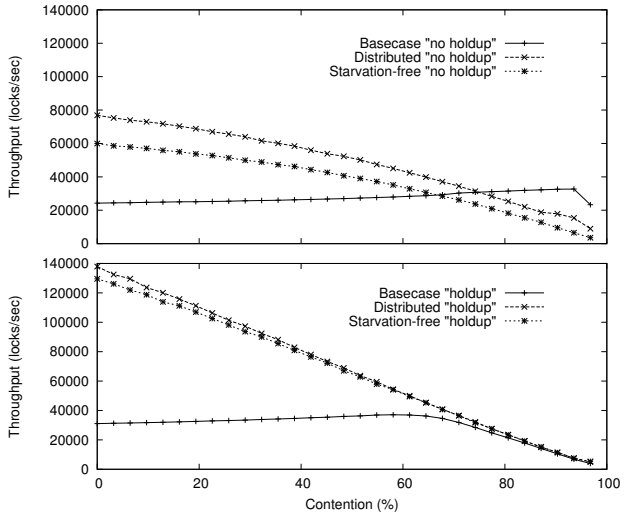

Figure 6. Lock throughput as a function of contention, for two different lock hold times.

To further investigate this issue, we conducted another experiment with some hold time between lock and unlock operations. Figure 6 shows the plot of throughput versus contention. Contention was calculated as in last experiment. Number of clients was fixed at 31 and number of locks was varied between 1 and 31. The top plot shows with no hold-up and the one below it shows when there is $\approx 270\mu s$ hold-up between lock and unlock operations. Distributed and Starvation-free fall below Basecase as contention increases due to failed *CAS* operations. When there is hold-up, throughput of Distributed and Starvation-free reaches the level of 0% case in Figure 5. Both curves stay above Basecase and finally all converge to a point. Basecase falls when contention increases since the lock-owner is not relinquishing locks fast enough to service the waiting clients. But this hold-up enables Distributed and Starvation-free to form queue and take the load from server. In a realistic scenario, one would expect clients to acquire a lock, do some amount of work and then release the lock. If the lock is highly contended, the work gives enough time to other clients to enqueue, thereby taking load off the server. More in-depth analysis of these experiments can be found in [5].

5.5 Loaded throughput

In this experiment, we measured uncontested, single-client, throughput of locks at server, under varying load conditions. Load is quantified by a single cpu-intensive process. Therefore, $load = n$, means there are n cpu-intensive processes running simultaneously with the server. Figure 7 shows two plots. The top plot is of throughput versus load. The plot below it shows the percentage CPU resources available to the server under varying load conditions.

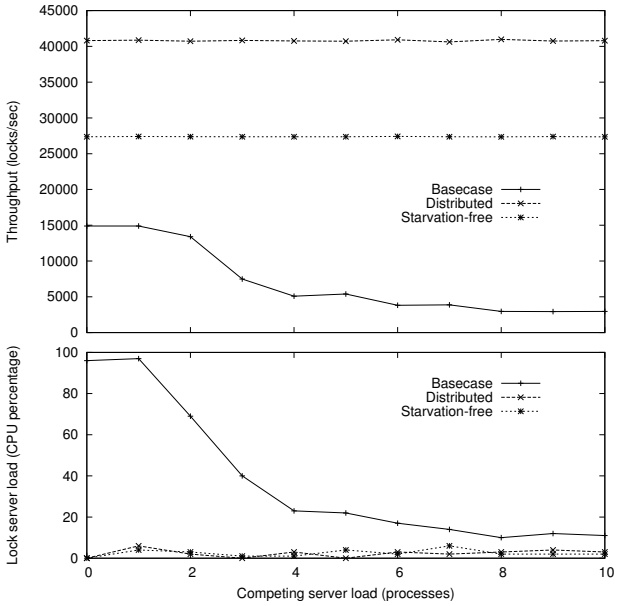

Figure 7. Throughput and utilization at lock-server for different loads.

We observe that Distributed and Starvation-free are agnostic to server-load, but the performance of Basecase takes a hit as the load increases. The reason is Basecase was implemented using polling server, while others were implemented with blocking server using interrupts. One can afford to use interrupts as there is very little involvement of server in the lock/unlock steps. However, in Basecase where server is in critical path, interrupts would have been prohibitively expensive. Even though Distributed and Starvation-free used blocking server, all algorithms used polling clients. We would like to remind that in DLM, each node plays the role of server and client. Therefore in Basecase, we will have two actively polling processes and hence vulnerable to addition of any cpu-intensive process. This experiment conclusively proves our hypothesis that offloading of lock-management to clients in indeed beneficial.

6 Related work

Several DLM alternatives were studied in [16], where they investigated the trade offs between static versus dy-

namic distribution of locks and gave analytic models for each alternative considered. SSDLM [15] investigated the use of DLM in clustered file-system. A variation of DLM [4], requiring $O(log(N))$ steps has been implemented.

Most of the work on distributed data structures was done in the context of Shared Memory Processors (SMPs). The work was motivated by the architecture of SMPs where cache consistency and coherence costs determined the designs. Basic axioms of lock-free data-structures were laid out by Barnes [1]. Valois [27, 28] further concertized those ideas in several primitive data-structures. Distributed versions of several primitive data-structures like FIFO queue [26, 21], hash-tables [20], priority-queues [24] exist for SMPs, but not for clustered environments, reason being, lack of primitives for efficient remote memory operations. In a related paper [19], Markatos et al. demonstrated the benefit of RDMA operations in building queues. However, the processes enqueued their data at a single site, unlike our implementation, where the queue was distributed.

7 Future Work

Our future work involves investigating ways of reducing unsuccessful *CAS* operations in distributed FIFO queue algorithms. We are also trying to extend our implementation by adding support for multi-modal locks, non-blocking locks and fault-tolerance.

8 Conclusion

In this paper, we presented an implementation of distributed FIFO queue based locking. We demonstrated a starvation-free algorithm which achieves better load-balancing by distributing the lock-management among the clients. Our design provides more throughput while consuming less CPU-resources at the server.

References

[1] G. Barnes. A method for implementing lock-free shared-data structures. In *SPAA '93*, pages 261–270, 1993.

[2] N. S. Bowen. A locking facility for parallel systems. *IBM Systems Journal*, 36(2), 1997.

[3] R. G. Davis. *VAXcluster Principles*. Digital Press, 1993.

[4] N. Desai and F. Mueller. A Log(n) Multi-Mode Locking Protocol for Distributed Systems. In *IPDPS '03*, 2003.

[5] A. Devulapalli and P. Wyckoff. Distributed queue-based locking using advanced network features. http://www.osc.edu/~ananth/papers/icpp05-tech-report.ps, 2005.

[6] A. Gottlieb et al. The NYU Ultracomputer—designing a MIMD shared memory parallel computer. *IEEE Transactions on Computers*, 32:175–189, February 1983.

[7] A. Silberschatz et al. *Operating System Concepts*. John Wiley & Sons, Inc., sixth edition.

[8] D. Dunning et al. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, 1998.

[9] J. Liu et al. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *Supercomputing Conference*, November 2003.

[10] R. Moran et al. *Oracle8 Parallel Server Concepts & Administration, Release 8.0*. Oracle Corporation, June 1997.

[11] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *Programming Languages and Systems*, 5(2):164–189, 1983.

[12] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 11(1):124–149, January 1991.

[13] IBM Corporation. System/370 principles of operation. Technical Report GA22-7000-8, IBM, 1981.

[14] *InfiniBand Architecture Specification, Release 1.1*. InfiniBand Trade Association, November 2002.

[15] H. Kishida and H. Yamazaki. SSDLM: architecture of a distributed lock manager with high degree of locality for clustered file systems. In *PACRIM '03*, volume 1, pages 9–12, August 2003.

[16] W.J. Knottenbelt, S. Zertal, and P.G. Harrison. Performance analysis of three implementation strategies for distributed lock management. In *Computers and Digital Techniques, IEE Proceedings*, volume 148, pages 176–187, Jul/Sep 2001.

[17] N.P. Kronenberg, H.M. Levy, and W.D. Strecker. VAXcluster: a closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, 1986.

[18] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM*, 17(8), 1974.

[19] E.P. Markatos, M. Katevenis, and P. Vatsolaki. The remote enqueue operation on networks of workstations. In *CANPC '98*, pages 1–14. Springer-Verlag, 1998.

[20] M.M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02*, pages 73–82.

[21] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*, pages 267–275.

[22] Myricom. Myricom Inc. http://www.myri.com.

[23] Sparc International. *The SPARC architecture manual, version 9*. Prentice Hall, 1994.

[24] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *IPDPS '03*, page 84.2, 2003.

[25] Kristin Thomas. *Programming Locking Applications*. IBM Corporation, 2001.

[26] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *SPAA '01*, pages 134–143.

[27] J.D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95*, pages 214–222, 1995.

[28] J.D. Valois. *Lock-free data structures*. PhD thesis, 1996.