On the Performance of Distributed Lock-Based Synchronization

Yuval Lubowich[*] Gadi Taubenfeld[*]

March 18, 2012

Abstract

Distributed mutual exclusion locks are the de facto mechanism for concurrency control on distributed data structures. A process accesses the data structure only while holding the lock, and hence the process is guaranteed exclusive access. The popularity of this approach is largely due to the apparently simple programming model of such locks and the availability of efficient implementations. We study the relation between classical types of distributed locking mechanisms and several distributed data structures which use locking for synchronization. Our objectives are:

1. To determine which one of the two classical locking techniques – token-based locking or permission-based locking – is more efficient. Our strategy to achieve this objective is to implement several locks and to compare their performance.
2. To propose, implement and test several distributed data structures, namely, two different types of counters, a queue, a stack and a linked list; and for each one of the data structures to determine what is the preferred mutual exclusion lock to be used as the underling locking mechanism.
3. To determine which one of the two proposed counters is better to be used either as a stand-alone data structure or when used as a building block for implementing other high level data structures.

Our testing environment is consisting of 20 Intel XEON 2.4 GHz machines running the Windows XP OS with 2GB of RAM and using a JRE version 1.4.2_08. All the machines were located inside the same LAN and were connected using a 20 port Cisco switch.

Keywords: Locking, synchronization, distributed mutual exclusion, distributed data structures, message passing, performance analysis.

[*]The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel. ylubowich@gmail.com ; tgadi@idc.ac.il

1 Introduction

1.1 Motivation and Objectives

Simultaneous access to a data structure shared among several processes, in a distributed message passing system, must be synchronized in order to avoid interference between conflicting operations. Distributed mutual exclusion locks are the de facto mechanism for concurrency control on distributed data structures. A process accesses the data structure only while holding the lock, and hence the process is guaranteed exclusive access. The popularity of this approach is largely due to the apparently simple programming model of such locks and the availability of efficient implementations.

Over the years several techniques have been proposed for implementing distributed mutual exclusion locks. These locks can be grouped into two main classes: *token-based* locks and *permission-based* locks. In token-based locks, a single token is shared by all the processes, and a process acquires the lock (i.e., is allowed to enter its critical section) only when it possesses the token. Permission-based locks are based on the principle that a process acquires the lock (and may enter its critical section) only after having received "enough" permissions from other processes. Our first objectives is:

> **Objective one.** To determine which one of the two locking techniques – token-based locking or permission-based locking – is more efficient. Our strategy to achieve this objective is to implement one classical token-based lock (Suzuki-Kasami's lock [40]), and two classical permission-based locks (Maekawa's lock [18] and Ricart-Agrawala's lock [31]), and to compare their performance.
>
> The worst-case message complexity of one of the permission-based locks (i.e., Maekawa's lock) is better than the worst-case message complexity of the other two locks. It would be interesting to find out whether this theoretical result be reflected in our performance analysis results.

It is possible to trivially implement a lock by letting a single pre-defined process (machine) to act as an "arbiter" or even to let all the data structures reside in the local memory of a single process and letting this process impose a definite order between concurrent operations. Such a centralized solution might be preferred in some situations, although it limits the degree of concurrency, imposes an extra load on the arbiter, and is less robust. In this work, we focus on fully distributed implementations of locks. Locks are just a tool used when implementing various distributed applications, thus, our second objective has to do with implementing lock-based data structures.

> **Objective two.** To propose, implement and test several distributed data structures, namely, two different types of counters, a queue, a stack and a linked list; and for each one of the data structures to determine what is the preferred mutual exclusion lock to be used as the underling locking mechanism.

In a shared memory implementation of a data structure, the shared data is usually stored in the shared memory. But, who should hold the data in a distributed message passing system? one process? all of them? In particular, when implementing a distributed counter, who should hold the current value of the counter? one process? all of them?

To address this question, for the case of a shared counter, we have implemented and compared two types of shared counter: A *find&increment* counter, where only the last process to update the counter needs to know its value; and an *increment&publish* counter, where everybody should know the value of the counter after each time the counter is updated. We notice that in the find&increment counter, once the counter is updated there is no need to "tell" its new value to everybody, but in order to update such a counter one has to find its current value first. In the increment&publish counter the situation is the other way around.

> **Objective three.** To determine which one of the two proposed counters is better to be used either as a stand-alone data structure or when used as a building block for implementing other high level data structures (such as a queue, a stack or a linked list).

We point out that, in our implementations of a queue, a stack, and a linked list, the shared data is distributed among all the processes; that is, all the items inserted by a process are kept in the local memory of that process.

1.2 Experimental Framework and Performance Analysis

A *process* corresponds to a given computation. That is, given some program, its execution is a process. Sometimes, for convenience, we will refer to the program code itself as a process. A process runs on a *processor*, which is the physical hardware. Several processes can run on the same processor although in such a case only one of them may be active at any given time. Real concurrency is achieved when several processes are running simultaneously on several processors. In our setting, each process runs on a different uni-processor machine.

We assume a fully connected, reliable physical network, where communication delays are determined by network contention. In order to measure and analyze the performance of the proposed five data structures and of the three locks, we have implemented and run each data structure with each one of the three implemented distributed mutual exclusion locks as the underling locking mechanism. We have measured each data structure's performance, when using each one of the locks, on a network with one, five, ten, fifteen and twenty processes, where each process runs in a different node. A typical simulation scenario of a shared counter looked like this: Use 15 processes to count up to 15 million by using a *find&increment* counter that employs Maekawa's lock as its underling locking mechanism.

The queue, stack, and linked list were also tested using each one of the two counters as a building block in order to determine which of the two counter performers better when used as a building block for implementing other high level data structures.

The dependencies between the various implementations of the data structures and locks are illustrated in Figure 1.

Special care was taken to make the experiments more realistic by preventing runs which would display an overly optimistic performance; for example, preventing runs where a process completes several operations while acquiring and holding the lock once.

Our testing environment consisted of 20 Intel XEON 2.4 GHz machines running the Windows XP OS with 2GB of RAM and using a JRE version 1.4.2_08. All the machines were located inside the same LAN and were connected using a 20 port Cisco switch.

1.3 Our Findings

The experiments, as reported in Section 5, lead to the following conclusions:

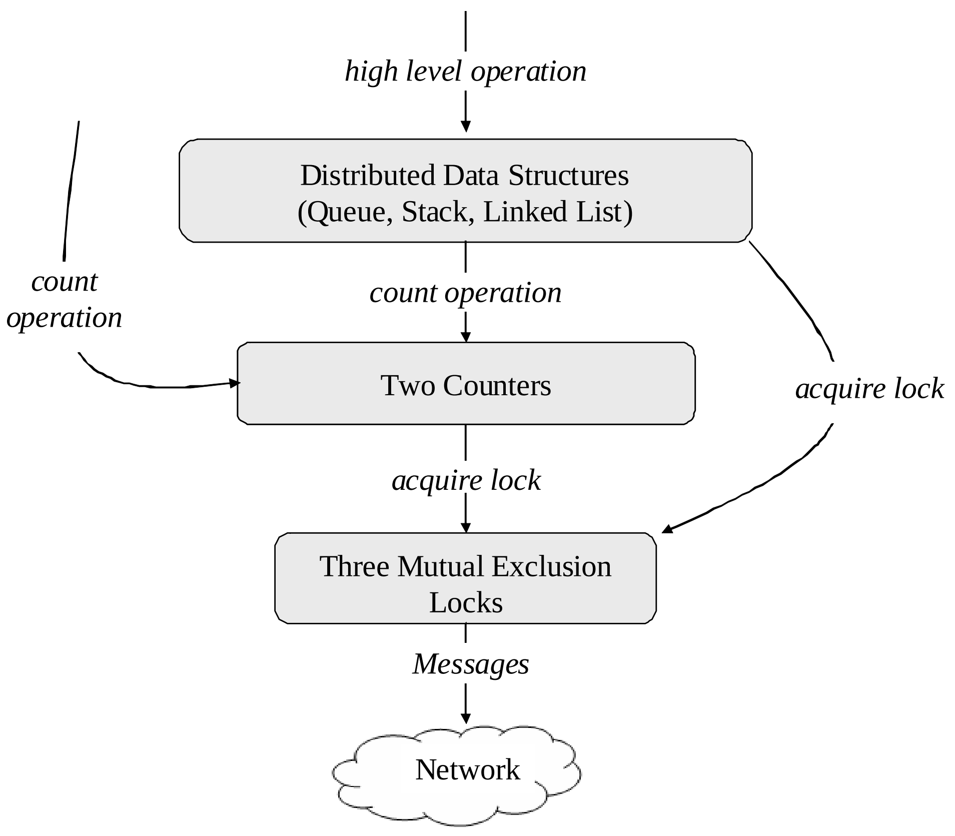


Figure 1: The dependencies between the various implementations of the data structures and locks.

1. One of the two permission-based locks always outperforms the token-based lock.

   A possible explanation for this result is that, in our experimental setting, when implementing distributed data structures, it may be better to take the initiative and search for information (i.e., ask for permissions) when needed, instead of waiting for your turn.

2. Maekawa's permission-based lock *always* outperforms Ricart-Agrawala and Suzuki-Kasami locks, when used as the underling locking mechanism for implementing the find&increment counter, the increment&publish counter, a queue, a stack, and a linked list. Put another way, for each one of the five data structures, the preferred lock to be used as the underling locking mechanism is always Maekawa's lock.

   The worst-case message complexity of Maekawa's lock is better than the worst-case message complexity of both Ricart-Agrawala and Suzuki-Kasami locks; thus, the performance analysis supports and confirms the theoretical analysis.

3. The find&increment counter *always* outperforms the increment&publish counter, either as a stand-alone data structure or when used as a building block for implementing a distributed queue and a distributed stack.

   A possible explanation for this result is that, in our experimental setting, when implementing distributed data structures, it may be more efficient to actively search for information only when it is needed, instead of distributing it in advance.

As expected, all the data structures exhibit performance degradation as the number of processes grows.

2 Mutual Exclusion

The *mutual exclusion* problem, which was first introduced by Edsger W. Dijkstra in 1965, is the guarantee of mutually exclusive access to a single shared resource when there are several competing processes [4]. The problem arises in operating systems, database systems, parallel supercomputers, and computer networks, where it is necessary to resolve conflicts resulting when several processes are trying to use shared resources. The problem is of great significance, since it lies at the heart of many interprocess synchronization problems.

2.1 The Problem

The problem is formally defined as follows: it is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section* and *exit*. Thus, the structure of a mutual exclusion solution looks as follows:

```
loop forever
        remainder code;
        entry code;
        critical section;
        exit code
end loop
```

A process starts by executing the remainder code. At some point the process might need to execute some code in its critical section. In order to access its critical section a process has to go through an entry code which guarantees that while it is executing its critical section, no other process is allowed to execute its critical section. In addition, once a process finishes its critical section, the process executes its exit code in which it notifies other processes that it is no longer in its critical section. After executing the exit code the process returns to the remainder.

The Mutual exclusion problem is to write the code for the *entry code* and the *exit code* in such a way that the following two basic requirements are satisfied.

Mutual exclusion: *No two processes are in their critical sections at the same time.*

Deadlock-freedom: *If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.*

The deadlock-freedom property guarantees that the system as a whole can always continue to make progress. However deadlock-freedom may still allow "starvation" of individual processes. That is, a process that is trying to enter its critical section, may never get to enter its critical section, and wait forever in its entry code. A stronger requirement, which does not allow starvation, is defined as follows.

Starvation-freedom: *If a process is trying to enter its critical section, then this process must eventually enter its critical section.*

Although starvation-freedom is strictly stronger than deadlock-freedom, it still allows processes to execute their critical sections arbitrarily many times before some trying process can execute its critical section. Such a behavior is prevented by the following fairness requirement.

First-in-first-out (FIFO): *No beginning process can enter its critical section before a process that is already waiting for its turn to enter its critical section.*

The first two properties, mutual exclusion and deadlock freedom, were required in the original statement of the problem by Dijkstra. They are the minimal requirements that one might want to impose. Of all the problems in interprocess synchronization, the mutual exclusion problem is the one studied most extensively. This is a deceptive problem, and at first glance it seems very simple to solve.

A *concurrent* system is a collection of processors that communicate by reading and writing from a shared memory. A *distributed* system is a collection of processors that communicate by sending messages over a communication network. Concurrent mutual exclusion algorithms are discussed in details in [41]. In this work we consider only distributed systems.

2.2 Distributed Mutual Exclusion Algorithms

We consider a system which is made up of n reliable processes, denoted $p_1, \ldots, p_n$, which communicate via message passing. The three mutual exclusion algorithms (i.e., locks) implemented in this work satisfy the *mutual exclusion* and the *starvation-freedom* requirements, as defined in Section 2.1.

The first published distributed mutual exclusion algorithm, due to Lamport [16], is based on the notion of logical clocks. Over the years a variety of techniques have been proposed for implementing distributed mutual exclusion locking algorithms [28, 29, 37, 38]. These algorithms can be grouped into two main classes: token-based algorithms[22, 27, 40, 39] and permission-based algorithms [3, 16, 18, 21, 31, 36].

- In token-based algorithms, a single token is shared by all the processes, and a process is allowed to enter its critical section only when it possesses the token. A process continues to hold the token until its execution of the critical section is over, and than it may pass it to some other process.
- Permission-based algorithms, are based on the principle that a process may enter its critical section only after having received "enough" permissions from other processes. Some permission-based algorithms require that a process receives permission for all of the other processes whereas others, more efficient algorithms, require a process to receive permissions from a smaller group.

Below we describe the basic principles of the three known distributed mutual exclusion algorithm that we have implemented. We assume that: (1) the underlying communication network is error-free; (2) transit times may vary; (3) messages may not be delivered in the order sent; and (4) processes (i,e nodes) are operated correctly.

2.3 Suzuki-Kasami's Token-based Algorithm

In Suzuki and Kasami's algorithm [40], the privilege to enter a critical section is granted to the process that holds the PRIVILEGE token (which is always held by exactly one process). Initially process p_1 has the privilege . A process requesting the privilege sends a REQUEST message to all other processes. A process receiving a PRIVILEGE message (i.e. the token) is allowed to enter its critical section repeatedly until the process passes the PRIVILEGE to some other process.

A REQUEST message of process p_j has the form REQUEST(j, m) where j is the process identifier and m is a sequence number which indicates that p_j is requesting its $(m+1)$'th critical section invocation. Each process has an array RN of size n, where n is the number of processes. This array is used to record the largest sequence of number ever received from each one of the other processes. When a REQUEST(j, m) message is received by p_i, the process updates RN by executing $RN[j] = \max(RN[j], m)$.

A PRIVILEGE message has the form of PRIVILEGE(Q, LN) where Q is a queue of requesting processes and LN is an array of size n such that, $LN[j]$ is the sequence number of the request of p_j granted most recently. When p_i finishes executing its critical section, the array LN, contained in the last PRIVILEGE message received by p_i, is updated by executing $LN[i] = RN[i]$, indicating that the current request of p_i has been granted. Next, every process p_j such that $RN[j] = LN[j] + 1$, is appended to Q provided that p_j is not already in Q. When these updates are completed, if Q is not empty then PRIVILEGE$(tail(Q), LN)$ is send to the process at the head of Q. If Q is empty then p_i retain the privilege until some process requests it.

The algorithm requires, in the worst case, n message exchanges per mutual exclusion invocation: $(n-1)$ REQUEST messages and one PRIVILEGE message. In the best case, when the process requesting to enter its critical section already holds the privilege token, the algorithm requires no messages at all.

2.4 Ricart-Agrawala's Permission-based Algorithm

The first permission-based algorithm, due to Lamport [16], has $3(n-1)$ message complexity. Ricart and Agrawala had modified Lamport's algorithm and were able to achieve $2(n-1)$ message complexity [31]. In this algorithm, when a process, say p_i, wants to enter its critical section, it sends a REQUEST(m, i) message to all other processes. This message contains a sequence number m and the process' identifier i, which are then used to define a priority among requests.

Each process has a local variable in which it records the highest sequences number seen in any REQUEST message sent or received. When process p_i sends a REQUEST(m, i) message, the value of the sequence number m in the message, equals to the value of the highest sequences number seen so far, plus 1.

Process p_j, upon receipt of a REQUEST message from process p_i, sends an immediate REPLY message to p_i if either p_j itself has not requested to enter its critical section, or p_j's request has a lower priority than that of p_i. Otherwise, process p_j defers its REPLY (to p_i) until its own (higher priority) request is granted.

The priority order decision is made by comparing a sequence number present in each REQUEST message. If the sequence numbers are equal, the identifiers of the processes are compared to determine which will enter first.

Process p_i enters its critical section when it receives REPLY messages from all the other $n-1$ processes. When p_i releases the critical section, it sends a REPLY message to all deferred requests. Thus, a REPLY message from process p_i implies that p_i has finished executing its critical section. This algorithm requires only $2(n-1)$ messages per critical section invocation: $n-1$ REQUEST messages and $n-1$ REPLY messages.

2.5 Maekawa's Permission-based Algorithm

In Maekawa's algorithm [18], process p_i acquires permission to enter its critical section from a set of processes, denoted S_i, which consists of at most $\sqrt{n}$ processes that act as arbiters. The algorithm uses only $c\sqrt{n}$ messages per critical section invocation, where c is a constant between 3 for light traffic and 5 for heavy traffic. Thus, the message complexity of Maekawa's algorithm is $O(\sqrt{n})$, while the message complexity of each one of the other two algorithms is $O(n)$.

Each process can issue a request at any time. In order to arbitrate requests, any two requests from different processes must be known to at least one *arbitrator* process. Since process p_i must obtain permission to enter its critical section from every process in S_i, the intersection of every pair of sets S_i and S_j must not be empty, so that processes in $S_i \cap S_j$ can serve as arbitrators between conflicting requests of p_i and p_j.

There are many efficient constructions of the sets S_1,..., S_n (see for example [15, 35, 24]). The construction used in our implementation is as follows: Assume that $\sqrt{n}$ is a positive integer (if not then few dummy processes can be added). Consider a matrix of size $\sqrt{n} \times \sqrt{n}$, where the value of an entry (i, j) in the matrix is $(i-1) \times \sqrt{n}+j$. Clearly, for every $k \in \{1, ..., n\}$ there is exactly one entry, denoted (i_k, j_k), whose value is k. The unique entry (i_k, j_k) is $(\lceil k/\sqrt{n} \rceil, k \pmod{\sqrt{n}}+1)$.

For each $k \in \{1, ..., n\}$, a subset S_k is defined to be the set of values on the row and the column passing through (i_k, j_k). Clearly, $S_i \cap S_j \neq \emptyset$ for all pairs i and j (and the size of each set is $2\sqrt{n} - 1$). Thus, whenever two processes p_i and p_j try to enter their critical sections, the arbiter processes in $S_i \cap S_j$ will grant access to only one of them at a time, and thus the mutual exclusion property is satisfied. By carefully designing the algorithm deadlock is also avoided.

3 Distributed Data Structures

Simultaneous access to a data structure shared among several processes must be synchronized in order to avoid interference between conflicting operations. There are several approaches for the construction of distributed (message passing) and concurrent (shared memory) data structures.

3.1 Lock-based Synchronization

As already pointed out, mutual exclusion locks are the de facto mechanism for concurrency control on distributed data structures: a process accesses the data structure only inside a critical section code, within which the process is guaranteed exclusive access. The popularity of this approach is largely due to the apparently simple programming model of such locks and the availability of implementations which are efficient and scalable.

When using locks, the *granularity* of synchronization is important. Using a single lock to protect the whole data structure, allowing only one process at a time to access it, is an example of *coarse-grained* synchronization. In contrast, *fine-grained* synchronization enables "small pieces" of a data structure to be locked, allowing several processes with non-interfering operations to access it concurrently. Coarse-grained synchronization is easier to program but is less efficient and is not fault-tolerant compared to fine-grained synchronization. Our implementations of distributed data structures, as discussed in the sequel, are all examples of coarse-grained synchronization.

Using mutual exclusion locks to protect the access to a shared data structure may degrade the performance of applications, as it forces processes to wait for a lock to be released. Moreover, slow or stopped processes may prevent other processes from ever accessing the data structure.

Various lock-based data structures have been proposed in the literature mainly for use in databases, see for example [2, 5, 6, 7, 8, 17, 32]. A distributed dictionary structure is studied in [25].

3.2 Lock-free Synchronization

In the case of *concurrent* data structures, where communication is done by reading and writing from shared memory, concurrent updates of simple data structures such as queues, stacks, heaps, linked lists, and counters, locking may be avoided by using *lock-free* data structures. Although we are interested in this paper in distributed (message-passing) systems, to place our results in perspective, we briefly discuss below the lock-free synchronization approach.

Several progress conditions have been proposed in the literature for lock-free data structures. The two most important conditions are non-blocking [14] and wait-freedom [11].

1. A data structure is *non-blocking* if it guarantees that *some* process will always be able to complete its pending operation in a finite number of its own steps regardless of the execution speed of other processes (admits starvation).

2. A data structure is *wait-free* if it guarantees that *every* process will always be able to complete its pending operations in a finite number of its own steps regardless of the execution speed of other processes (does not admit starvation).

Advantages of using non-blocking algorithms (i.e., algorithms that satisfy the non-blocking property) are that they are not subject to deadlocks or priority inversion, they are resilient to process failures (no data corruption on process failure), and they do not suffer significant performance degradation from scheduling preemption, page faults, or cache misses. Non-blocking algorithms are still not used in many practical applications as such algorithms are often complex (each variant of a non-blocking queue is still a publishable result).

While non-blocking has the potential to significantly improve the performance of concurrent applications, the wait-free property (although desirable) imposes too much overhead upon the implementation. Wait-free algorithms are often very complex and memory consuming, and hence considered less practical than non-blocking algorithms. Furthermore, starvation can be efficiently handled by collision avoidance techniques such as exponential backoff.

The term *lock-free* algorithms refers to algorithms that do not use locking in any way. Lock-free algorithms are designed under the assumption that synchronization conflicts are rare and should be handled only as exceptions; when a synchronization conflict is noticed the operation is simply restarted from the beginning. Non-blocking algorithms are, by definition, also lock-free, but lock-free algorithms are not necessarily non-blocking.[1]

Various other progress conditions weaker than non-blocking have been proposed in the literature. For example, a data structure is *obstruction-free* if it guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" long enough [12]. Efficient lock-free algorithms usually require the use of powerful synchronization primitives such as compare-and-swap or load-linked/store-conditional. Lock-free implementations of various concurrent data structures have appeared in many papers; few examples are [9, 10, 20, 34, 42].

[1]In the literature, the terms lock-free and non-blocking are sometimes used as synonymous, or even with opposite meaning to the way they are defined in this paper. We find it is useful to distinguish between algorithms that do not require locking (i.e., lock-free algorithms) and those that actually satisfy the non-blocking progress condition.

General methodologies for implementing concurrent data structures and algorithms have been proposed in the literature. Such implementations are usually less efficient compared to specialized algorithms. *Transactional memory* is one such methodology which has gained momentum in recent years as a simple way for writing concurrent programs [13, 33]. It is based on the idea of transactions from databases. Transactional memory allows programmers to define customized read-modify-write operations and to apply them to multiple, independently-chosen words of memory as one indivisible step.

The underline implementation of transactional memory may use both locks and lock-free objects, but the complexity is hidden from the programmer. Several lock-based and lock-free implementations have be proposed in the last few years. The transactional memory programming paradigm is also seriously considered by the industry. A list of citations can be found at http://www.cs.wisc.edu/trans-memory/.

3.3 Our Implementations: Two counters, a queue, a stack and a linked list

We have proposed and implemented five distributed data structures: two types of counters, a queue, a stack and a linked-list. Each one of these data structures implementations makes use of an underlying locking mechanism. As already mentioned, we have implemented the three mutual exclusion algorithms described in the previous section, and for each of the five data structures determined what is the preferred mutual exclusion algorithm to be use for locking.

Below we describe the five distributed data structures that we have studied. All the data structures are linearizable. Linearizability means that, although several processes may concurrently invoke operations on a linearizable data structure, each operation appears to take place instantaneously at some point in time, and that the relative order of non-concurrent operations is preserved [14].

Two counters. A shared counter is a linearizable data structure that supports the single operation of incrementing the value of the counter by one and returning its previous value. We have implemented and compared two types of shared counter:

1. A *find&increment* counter. In this type of a counter only the last process to update the counter needs to know its value. In the implementation, a single lock is used, and only the last process to increment the counter knows its current value. A process p that tries to increment the shared counter first acquires the lock. Then p sends a FIND message to all other processes. When the process that knows the value of the counter receives a FIND message, it replies by sending a message with the value of the counter to p. When p receives the message, it increments the counter and releases the lock. (We notice that p can keep on incrementing the counter's value until it gets a FIND message.)

2. An *increment&publish* counter. In this counter everybody should know the value of the counter each time it is updated. In the implementation, a single lock is used. A process that tries to increment the shared counter first acquires the lock. Then, it raises the counter value, sends messages to all other processes informing them of the new counter value, gets acknowledgements, and releases the lock.

A queue. A distributed queue is a linearizable data structure that supports enqueue and dequeue operations, by several processes, with the usual queue semantics. We have implemented a distributed queue which consists of local queues residing in the individual processes participating in the distributed queue. A single lock and a shared counter are used for the implementation. Each element in a queue has a timestamp that is generated using the shared counter. An ENQUEUE operation is carried out by raising the counter's value by one and enqueueing an element in the local queue along with the counter's value. A DEQUEUE operation is carried out by first acquiring the lock, locating the process that holds the element with the lowest timestamp, removing this element from this process' local queue, and releasing the lock.

A stack. A distributed stack is a linearizable data structure that supports push and pop operations, by several processes, with the usual stack semantics. We have implemented a distributed stack which is similar to the distributed queue. A single lock and a shared counter are used for the implementation. It consists of local stacks residing in the individual processes participating in the distributed stack. Each element in the stack has a timestamp that is generated by the shared counter. A PUSH operation is carried out by incrementing the counter value by one and pushing the element in the local stack along with the counter's value. A POP operation is carried out by acquiring the lock, locating the process that contains the element with the highest timestamp, removing this element from its local stack, an releasing the lock.

A linked list. A distributed linked list is a linearizable data structure that supports insertion and deletion of elements from any point in the list. We have implemented a list which consists of a sequence of elements, each containing a data field and two references ("links") pointing to the next and previous elements. Each element can reside in any process. The distributed list also supports the operations "traverse list"; and "size of list". The list contains a head and a tail "pointers" that can be sent to requesting processes. Each pointer maintains a reference to a certain process and a pointer to a real element stored in that process. Manipulating the list requires that a lock be acquired.

A process that needs to insert an element to the head of the list acquires the lock, and sends a request for the "head pointer" to the rest of the processes. Whenever a process that holds the "head pointer" receives the message, it immediately replies by sending the pointer to the requesting process. Once the requesting process has the "head pointer" inserting the new element is purely a matter of storing it locally and modifying the "head pointer" to point to the new element (the new element of course now points to the element previously pointed by the "head pointer"). Deleting an element from the head of list is done much the same way. Inserting or deleting elements from the list requires a process to acquire the (single) lock, traverse the list and manipulate the list's elements. A process is able to measure the size of the list by acquiring the lock and then querying the other processes about the size of their local lists.

4 The Experimental Framework

Our testing environment consisted of 20 Intel XEON 2.4 GHz machines running the Windows XP OS with 2GB of RAM and using a JRE version 1.4.2_08. All the machines were located inside the same LAN and were connected using a 20 port Cisco switch.

We measured each data structure's performance, using each one of the distributed mutual exclusion algorithms, by running each data structure on a network with one, five, ten, fifteen and twenty processes, where each process runs in a different node of the network. For example a typical simulation scenario of a shared counter looked like this: Use 15 processes to count up to 15 million by using a find&increment counter that employs Maekawa's algorithm as its underling locking mechanism.

All tests were implemented using Coridan's messaging middleware technology called MantaRay. MantaRay's is a fully distributed server-less architecture where processes running in the network are aware of one another and as a result are able to send messages back and forth directly. We have tested each of the implementations in hours, and sometimes days long, of executions on various number of processes (machines).

5 Performance Analysis and Results

All the experiments done on the data structures we have implemented, start with an initially empty data structure (queue, stack etc.) to which processes have performed a series of operations. For example, in the case of a queue, the processes performed a series of enqueue/dequeue operations. Each process enqueued an element, did "something else" and repeated for a million times. After that, the process dequeued an element, did "something else" and repeated for a million times again. The "something else" consisted of approximately 30 mSeconds of doing nothing and waiting. As with the tests done on the locking algorithms, this served in making the experiments more realistic in preventing long runs by the same process which would display an overly optimistic performance, as a process may complete several operations while holding the lock. The time a process took to complete the "something else" is *not* reported in our figures. The experiments, as reported below, lead to the following conclusions:

- Maekawa's permission-based algorithm always outperforms Ricart-Agrawala and Suzuki-Kasami algorithms, when used as the underling locking mechanism for implementing the find&increment counter, the increment&publish counter, a queue, a stack, and a linked list;
- The find&increment counter always outperforms the increment&publish counter, either as a stand-alone data structure or when used as a building block for implementing a distributed queue and a distributed stack.

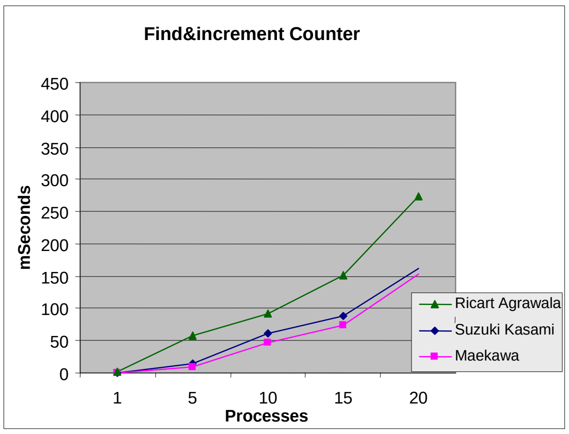
As expected, the data structures exhibit performance degradation as the number of processes grows.

5.1 Counters

The two graphs in Figure 2, show the time one process spends performing a single count up operation averaged over one million operations for each process using each of the three locking algorithms implemented. As can be seen, the counters perform worse when using Ricart-Agrawala algorithm and perform best when using Maekawa's algorithm. As for comparing the two counters, it is clear that the find&increment counter behaves and scales better than the increment&publish counter when the number of processes grows. The observation that the find&increment counter is better than the increment&publish counter will become also clear when examining the results for the queue and stack implementations that make use of shared counters as building blocks.

5.2 A Queue

The two graphs in Figure 3, show the time one process spends performing a single *enqueue* operation averaged over one million operations for each process using each of the three locks. Similar to the performance analysis of the two counters, the queue performs worse when using Ricart-Agrawala



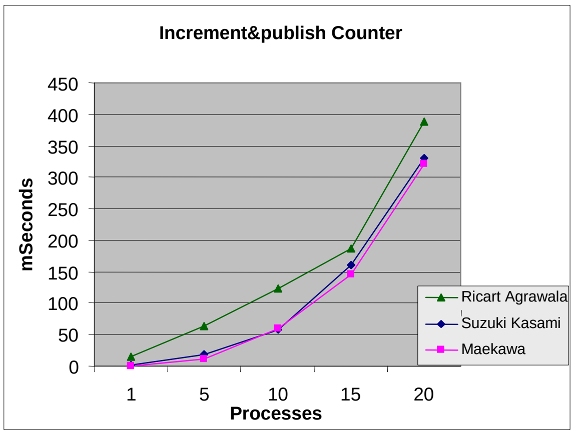


Figure 2: The time one process spends performing a single count up operation averaged over one million operations per process, in the find&increment counter and in the increment&publish counter.

algorithm and performs best when using Maekawa's algorithm. It is clear that the queue performs better when using the find&increment counter than when using increment&publish counter.

The dequeue operation does not make use of a shared counter. Figure 4bb shows the time one process spends performing a single *dequeue* operation averaged over one million operations for each process using each of the three locks. Similar to the performance analysis of the enqueue operation, the dequeue operation is the slowest when using Ricart-Agrawala algorithm, and is the fastest when using Maekawa's algorithm.

5.3 A Stack

As expected, the graphs of the performance analysis results for a stack are almost the same as those presented in the previous subsection for a queue.

The two graphs in Figure 5, show the time one process spends performing a single *push* operation averaged over one million operations for each process using each of the three locking algorithms implemented.

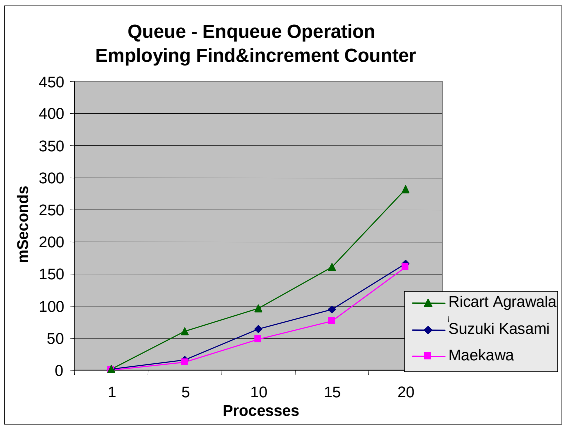
As in all previous examples, the stack performs worse when using Ricart-Agrawala algorithm and performs best when using Maekawa's algorithm. As for comparing the two counters, it is clear that the stack performs better when using the find&increment counter than when using increment&publish counter.

The pop operation does not make use of a shared counter. The graph in Figure 6, shows the time one process spends performing a single *pop* operation averaged over one million operations for each process using each of the three locking algorithms implemented.

Similar to the performance analysis of the push operation, the pop operation is the slowest when using Ricart-Agrawala algorithm, and is the fastest when using Maekawa's algorithm.

5.4 A Linked list

The linked list we have implemented does not make use of a shared counter. Rather it uses the locking algorithm directly to acquire a lock before manipulating the list itself. The graphs in Figure 7,



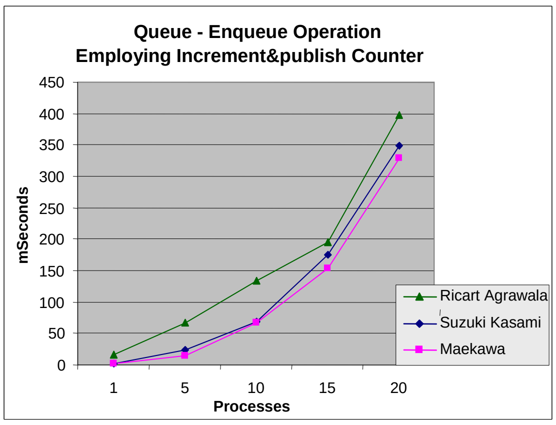


Figure 3: The time one process spends performing an *enqueue* operation averaged over one million operations per process, in a queue employing the find&increment counter and in a queue employing the increment&publish counter.

show the time one process spends performing a single *insert* operation or a single *delete* operation, averaged over one million operations for each process using each of the three locking algorithms implemented. As in all previous examples, the linked list performs worse when using Ricart-Agrawala algorithm and performs best when using Maekawa's algorithm as the underling locking mechanism.

6 Discussion

Data structures such as shared counters, queues and stacks are ubiquitous in programming concurrent and distributed systems, and hence their performance is a matter of concern. Rather surprisingly, while the subject of data structures is a very hot research topic in recent years in the context of concurrent (shared memory) systems, this is not the case for distributed (message passing) systems. In this work, we have studied the relation between classical locks and specific distributed data structures which use locking for synchronization.

The experiments consistently revealed that the implementation of Maekawa's lock is more efficient than that of the other two locks, and that the implementation of the find&increment counter is consistently more efficient than that of the increment&publish counter. The fact that Maekawa's lock performs better is, in part, due to the fact that its worst-case message complexity is better. The results suggest that, in our experimental setting, it is more efficient to actively search for information (or ask for permissions) only when it is needed, instead of distributing it to everybody in advance. It would be interesting to find out whether similar results hold also for different experimental set ups.

For our investigation, it is important to implement and use the locks as completely independent building blocks, so that we can compare their performance. In practice, various optimizations are possible. For example, when implementing the find&increment counter using a lock, a simple optimization would be to store the value of the counter along with the lock. Thus, when a process requests and obtains a lock, it obtains the current value of the counter along with the lock, thereby eliminating the need for any find messages.

Future work would entail implementing and evaluating other locking algorithms [3, 23], and fault tolerant locking algorithms that do not assume an error-free network [1, 19, 27, 30, 26]. It

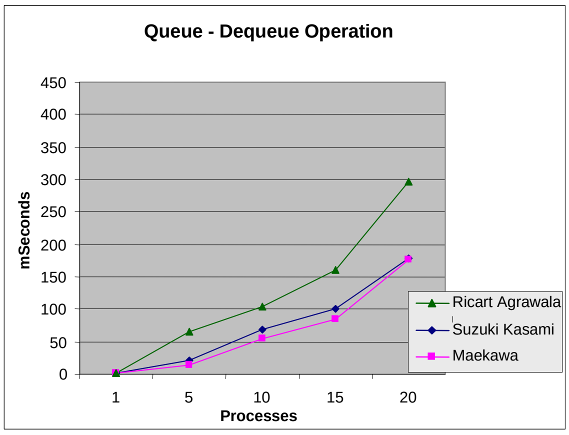


Figure 4: The time one process spends performing a *dequeue* operation averaged over one million operations per process.

would also be interesting to consider additional distributed lock-based data structures, and different experimental set ups.

By implementing more locking algorithms and data structures more insightful knowledge can be gained about their relationships. This knowledge can be used to improve the performance of distributed applications.

References

[1] D. Agrawal and A. El-Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, 1991.

[2] R. Bayer and M. Schkolnick. Concurrency operations on B-trees. *Acta Informatica*, 1(1):1–21, 1977.

[3] O. S. F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–147, 1983.

[4] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

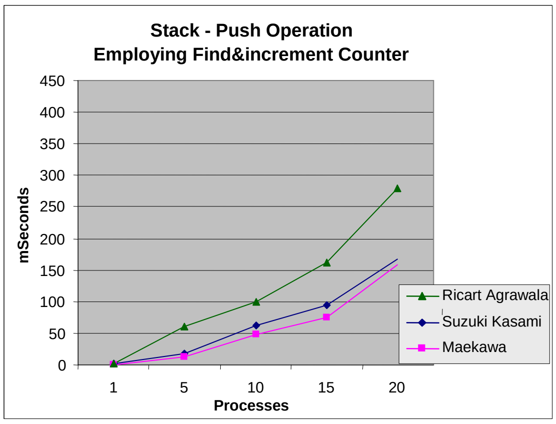
[5] C. S. Ellis. Concurrency search and insert in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

[6] C. S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, c-29:811–817, 1980.

[7] C. S. Ellis. Extendible hashing for concurrent operations and distributed data. In *Proc. of the 2nd ACM symposium on Principles of database systems*, pages 106–116, 1983.

[8] C. S. Ellis. Distributed data structures: A case study. *IEEE Transactions on Computers*, c-34(12):1178–1185, 1985.

[9] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing*, pages 50–59, 2004.



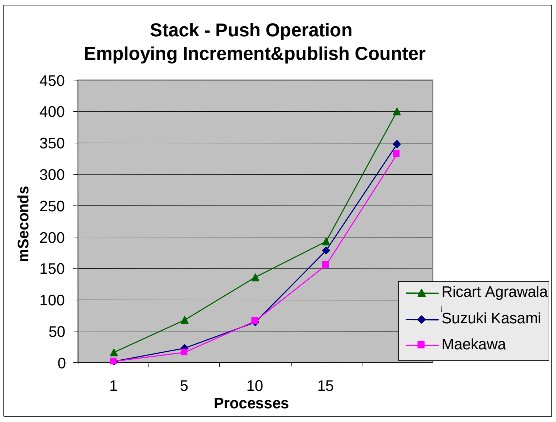


Figure 5: The time one process spends performing an *push* operation averaged over one million operations per process, in a stack employing the find&increment counter and in a stack employing the increment&publish counter.

[10] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. 15th international symp. on distributed computing*, *LNCS* 2180:300–314, 2003.

[11] M. P. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, 1991.

[12] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.

[13] M. P. Herlihy and J.E.B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[15] T. Ibaraki and T. Kameda. A theory of coteries: Mutual exclusion in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):779–794, 1993.

[16] L. Lamport. Time, clocks, and the order of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[17] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[18] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985.

[19] S. Mishra and P. K. Srimani. Fault-tolerant mutual exclusion algorithms. *Journal of Systems and Software*, 11(2):111–129, 1990.

[20] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

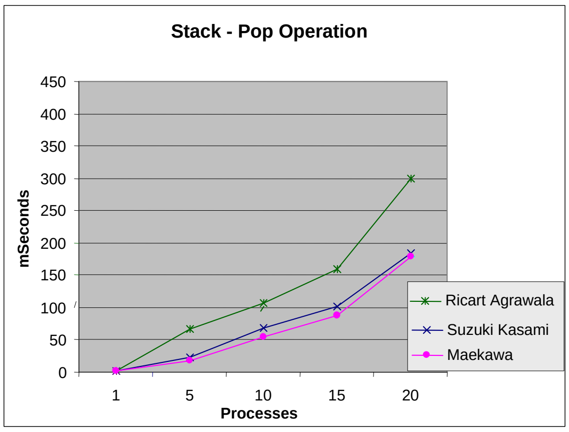


Figure 6: The time one process spends performing a *pop* operation averaged over one million operations per process.

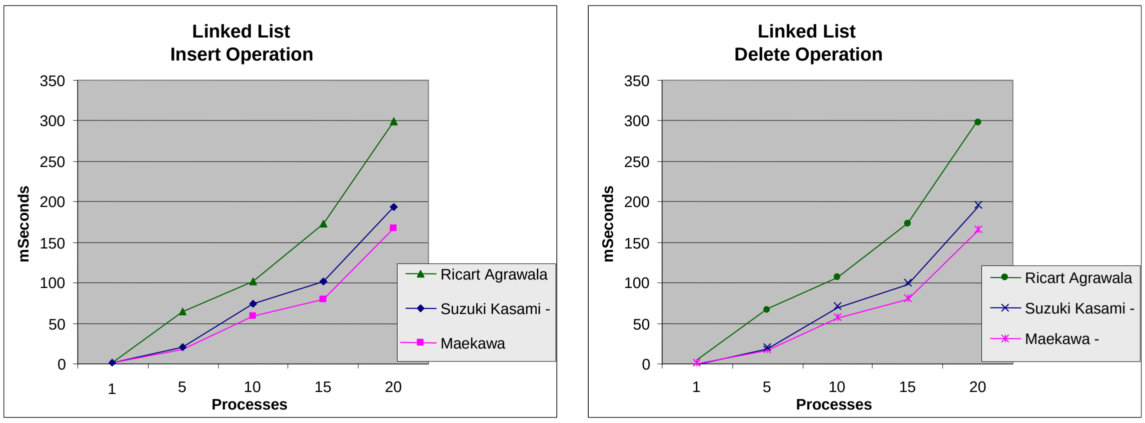


Figure 7: The time one process spends performing an *insert* operation or *delete* operation averaged over one million operations per process in a linked list.

[21] M. Mizuno, M. Mesterenko, and H. Kakugawa. Lock-based self-stabilizing distributed mutual exclusion algorithm. In *Proc. 17th Inter. Conf. on Dist. Comp. Systems*, 708–716, 1996.

[22] M. Naimi and M. Trehel. An improvement of the $\log n$ distributed algorithm for mutual exclusion. In *Proc. 17th Inter. Conf. on Dist. Comp. Systems*, 371–375, 1987.

[23] M. L. Neilsen and M. Mizuno. A DAG-based algorithm for distributed mutual exclusion. In *Proc. 17th Inter. Conf. on Dist. Comp. Systems*, 354–360, 1991.

[24] M. L. Neilsen and M. Mizuno. Coterie join algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):582–590, 1992.

[25] D. Peleg. Distributed data structures: A complexity-oriented structure. In *4rd International Workshop on Distributed Algorithms*, 1990. LNCS, vol. 486, 1990, pages 71–89.

[26] S. Rangarajan and S. K. Tripathi. A robust distributed mutual exclusion algorithm. *Lecture Notes in Computer Science*, 579:295–308, 1992.

[27] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[28] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.

[29] M. Raynal. Simple taxonomy for distributed mutual exclusion algorithms. *Operating Systems Review (ACM)*, 25(2):47–50, 1991.

[30] R. L. N. Reddy, B. Gupta, and P. K. Srimani. New fault-tolerant distributed mutual exclusion algorithm. In *Proc. of the ACM/SIGAPP Symp. on Applied Computing*, pages 831–839, 1992.

[31] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *CACM*, 24(1):9–17, 1981. Corrigendum in *CACM*, 24(9):578,1981.

[32] Y. Sagiv. Concurrent operations on B-trees with overtaking. In *ACM Principles of Dadabase Systems*, pages 28–374, 1985.

[33] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, 1995.

[34] H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. In *8th International Conference on Principles of Distributed Systems*, 2004.

[35] D. Shou and S. D. Wang. A new transformation method for nondominated coterie design. *Information Sciences*, 74(3):223–246, 1993.

[36] M. Singhal. A dynamic information-structure mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):121–125, 1992.

[37] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, 1993.

[38] M. Singhal and N.G. Shivaratri. *Advanced concepts in operating systems: distributed, database and multiprocessor operating systems*. McGraw-Hill, Inc., 1994.

[39] J.L.A. van de Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2:113–115, 1987.

[40] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, 1985.

[41] G. Taubenfeld. Synchronization Algorithms and Concurrent Programming. *Pearson / Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.

[42] J. D. Valois. Implementing lock-free queues. In *Proc. of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 212–222, 1994.