Lock Based Distributed Data Structures

M.Sc dissertation

Submitted by Yuval Lubowich

Under the supervision of Prof. Gadi Taubenfeld (IDC)

October, 2006.
<ResourceList>
  <Resource type="book">
    <Title>מבני נתונים מבוזרים</Title>
    <Subtitle>مبוסמי بغילה</Subtitle>
    <Authors>
      <Author>יד יובל לובוביץ-על</Author>
      <Author>גדי טאובנפלד' העבודה בוצעה בהנחיית פרופ</Author>
    </Authors>
    <Publisher>מרכז הבינתחומי בהרצליה</Publisher>
    <PublicationDate>אוקטובר 2006</PublicationDate>
  </Resource>
</ResourceList>
Lock Based Distributed Data Structures

Yuval Lubowich
Department of Computer Science
Interdisciplinary Center Herzliya
ylubowich@idc.ac.il

Done under the supervision of Prof. Gadi Taubenfeld {tgadi@idc.ac.il}

Abstract

In this work we study the relation between classical types of distributed mutual exclusion locks and several simple distributed data structures which use locking for synchronization. We have designed and implemented several data structures and for each data structure we have determined what is the preferred type of mutual exclusion algorithm to be used for locking. The data structures we have implemented include: two types of shared counters, a queue, a stack and a linked list.

Our findings show: (1) that one of the locking algorithms we have tested outperforms the other locking algorithms. (2) Out of the two types of shared counters we have suggested and implemented, one type clearly outperforms the other one. (3) The data structures we have implemented exhibit performance degradation as the number of processes grows.

The methodology used for performance analysis was to implement and run each one of the proposed data structures with each one of the mutual exclusion algorithms as the underlying locking mechanism, and to measure the performance for different number of participating processes. All these tests were implemented using Coridan's commercial messaging middleware technology, which is known as MantaRay.
1. **Introduction**

Concurrent access to a data structure shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion locks are the de facto mechanism for concurrency control on concurrent and 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 module of such locks and the availability of implementations that are said to be efficient and scalable.

In this work we have: (1) suggested and implemented several lock-based distributed data structures, namely, two different types of counters, a queue, a stack and a linked list; (2) for each one of these data structures we have determined what is the preferred mutual exclusion algorithm to be used as the underlying locking mechanism; and (3) for the two counter algorithms we have determined which is better to use either as a stand-alone data structure or when used as a building block for implementing other high level data structures.

Two communication models are generally considered for distributed computing: message-passing and shared memory. In the former model, it is usually assumed that processes are connected by communication channels and that the processes communicate by sending and receiving messages. The later model abstracts a hardware-based shared memory comprised of registers. The processes exchange information by means of read and write operations exported by the registers. This work focuses solely on message passing algorithms.

Message-passing algorithms, as well as shared memory algorithms, vary in many respects: the degree of distribution of control, the degree of parallel operation, network traffic, applicable network topologies, and reliability. The algorithms discussed in this paper 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 data structures, we have implemented and run each data structure with each one of the three implemented mutual exclusion algorithms as the underlying locking mechanism, while measuring the performance for different number of participating processes.
All the experimental tests were implemented using Coridan's messaging middleware technology which is known as MantaRay. MantaRay is a reliable messaging framework that uses multicasting to discover other processes running MantaRay in a network, allowing new machines (processes) to be added or removed from the topology. Once a process joins the network it can immediately start to send and receive messages from other processes already running MantaRay[23].

Over the years a variety of techniques have been proposed for implementing mutual exclusion locks efficiently by employing message-passing. These distributed locking algorithms can be grouped into two main classes; Token based and permission based ones.

(1) We show that out of all the locking algorithms we examined, one permission based algorithm (Maekawa) is a much better choice than the rest of the algorithms when the number of processes is larger than 5. When the number of participating processes is small (usually smaller than 5 processes) none of the algorithms we tested exhibited significant performance difference. However, another well known permission based algorithm (Ricart – Agrawala) is a poor choice for a locking algorithm. Whereas Maekawa's algorithm always performs at least as fast as any other algorithm we tested. (2) We show that one type of shared counter (token counter) always outperforms the type of counter (notification counter). (3) As expected, as a result of using locking the data structures tested exhibit performance degradation as the number of processes grows.

2. Distributed Mutual Exclusion Algorithms

The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among a number of competing processes [3]. It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the reminder, entry, critical and exit. The problem is to write the code for the entry and the exit sections in such a way that the following two basic requirements are satisfied (assumed a process always leaves its critical section).

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.
A stronger liveness requirement than deadlock freedom is,

**Starvation freedom:** If a process is trying to enter its critical section, than the process must eventually enter its critical section.

All the locking algorithms discussed in this work satisfy mutual exclusion and starvation freedom.

Over the years a variety of techniques have been proposed for implementing mutual exclusion locks efficiently by employing message-passing. These distributed locking algorithms can be grouped into two main classes:

♦ **Token based** algorithms [11, 12, 16, 20, 21], where a single token is shared by all the processes. 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.

♦ **Permission based** algorithms [2, 6, 7, 13, 17]. These algorithms are based on the principle that a process may enter its critical section only after having received “enough” permission from other processes. Some permission based locking algorithms require that a process receives permission for all of the other processes [13] whereas others, more efficient algorithms, require a process to receive permissions from a smaller group [7].

In this paper, we focus on the message passing model. We have implemented one token based algorithm [20] and two non-token based algorithms [7, 13] which are described in the following sections (this is in addition to the specific data structures, described later, that we have suggested and implemented).

Message traffic is the main complexity measurement used to evaluate the algorithms.
2.1 Suzuki-Kasami’s Token-based Algorithm

The Suzuki-Kasami algorithm [20] is one of the first token based algorithms conceived. The algorithm's message complexity is: exactly N messages for each entrance to the critical section in the worst case. N denotes the number of processes.

In Suzuki and Kasami’s algorithm, the privilege to enter a critical section is granted by holding a PRIVILEGE token. Initially the process who's ID is ‘1’ has the privilege which is always held by exactly one process. 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 j has the form REQUEST (j, m) where j is the process identifier and m is a sequence number which indicates that process j is now requesting its (m+1)th critical section invocation. Each process has an array RN the size of N, where N is the number of processes, used to record the largest sequence of numbers ever received from each one of the other processes. When a REQUEST message is received by process i, the process updates RN by 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 the size of N such that, LN[j] is the sequence number of the request of process j granted most recently. When process i finishes executing its critical section, the array LN, contained in the last PRIVILEGE message received by process j is updated by LN[i] = RN[i], to indicate that the current request of process i has been granted. Next all process identifiers j such that RN[j] = LN[j] + 1, is appended to Q provided that j is not already found in Q. When these updates are complete, if Q is not empty then PRIVILEGE (tail(Q), LN) is send to the process found at the front of Q. If Q is empty then process i retain the privilege until a requesting process is found by arrivals of REQUEST messages.

The algorithm requires, in the worst case, at most N message exchanges per mutual exclusion invocation; (N-1) REQUEST messages and one PRIVILEGE message, or in the best case no message at all if the process requesting to enter its critical section already has the privilege token.
2.2. Ricart-Agrawala Permission-based Algorithm

The first permission based algorithm proposed by Lamport [6], has 3N message complexity. Ricart and Agrawala’s later modified Lamport's algorithm and were able to achieve 2N message complexity in their new algorithm [13].

The algorithm reduces message traffic by using an implicit RELEASE message. In this algorithm, when a process S wants to enter its critical section, it sends a REQUEST message to all other processes. This message contains a sequence number and the process's identifier (S), which are then used to define a priority among requests.

Process Y, upon receipt of a REQUEST message from process S, sends an immediate REPLY message to process S if Y itself not requesting or if Y’s request has a lower priority than that of S. Otherwise, process Y defers its REPLY until its own (higher priority) request is granted.

Process S enters its critical section when it receives REPLY message from all other N-1 processes. When process S releases the critical section, it sends a REPLY message to all deferred requests. Thus, a REPLY message from process S implies that it has finished execution of the critical section. This algorithm requires only 2*(N-1) messages per critical section execution as opposed to Lamport's algorithm which uses 3*(N - 1) messages per CS invocation: (N - 1) REQUEST, REPLY, and RELEASE messages are sent.

During our testing we have discovered and fixed a race condition that can occur in the original Ricart-Agrawala algorithm. This race condition can cause the algorithm to deadlock.

Correction to the Ricart-Agrawala Algorithm

Ricart and Agrawala describe the proposed algorithm and note that; “…Each node has three processes to implement the mutual exclusion.

(1) One is awakened when mutual exclusion is invoked on behalf of this node.
(2) Another receives and processes REQUEST messages.
(3) The last receives and processes REPLY messages.
The three processes run asynchronously but operate on a set of common variables."

Deadlocks occur when a process, say P, is releasing the critical section and at the same time a REQUEST message is received by process P. Releasing the critical section requires P to traverse the list of processes and for each deferred process send a REPLY message.

```java
acquireCriticalSection{
    acquire critical section{
        critical section{do something}
        for (int j = 1; j <= N; j++) {
            if (replyDeferred[j - 1]) {
                replyDeferred[j - 1] = false;
                // Send a reply to process j
                sendReplyMessage(j);
            }
        }
    }
}
```

**Pseudo code 1 Releasing a critical section**

When a REQUEST message is received by a process it is either answered by a REPLY message (if the process approves the REQUEST) or deferred to a later time (if the process is currently executing its critical section or is in the process of starting the execution). The decision to defer the REPLY message is calculated by evaluating the following equation:

```java
// K is the sequence number this process got from the process issuing the REQUEST message
int k = message.getIntProperty("ourSequenceNumber");

boolean deferIt = requestingCriticalSection && ((k > ourSequenceNumber) ||
(k == ourSequenceNumber && j > myID));
```

**Pseudo code 2 Evaluating the deferIt variable**

By evaluating the deferIt variable the process either marks the REQUEST as deferred or if sends a REPLY message.

```java
if (deferIt)
    replyDeferred[j - 1] = true;
else
    sendReplyMessage(j); // send the reply only to j
```

**Pseudo code 3 Handling REQUEST message**

---

The deadlock might occur if a REQUEST message is received by the process while it is in the process of releasing the critical section. More precisely, while executing the "for" loop in the release section, the replyDeferred\[j - 1\] variable might be equal to 'false' causing the algorithm not to send a REPLY message to the waiting process. At the same time the REQUEST message (received by the process) is deemed to be deferred because the critical section has not been released yet. Thus no REPLY message is ever sent to the process originating the REQUEST message.

We have modified the algorithm and removed the race condition by adding another locking semaphore to the code. In a sense serializing the execution of the process; either the process releases the critical section or it handles a REQUEST message but not both at the same time.

Whenever a REQUEST message is received by the process, the process handling the REQUEST tries to acquire the semaphore. The same semaphore is used to restrict access to the part of the code responsible for releasing the critical section. In other words, either the critical section is released or a REQUEST message is handled by the process. By restricting the access to these sections of the process we have removed the race condition and eliminated the possibility of deadlocks from happening.

### 2.3. Maekawa’s Permission-based Algorithm

Maekawa algorithm [7] is a quorum based algorithm. Processes acquire permission to enter the critical section from a quorum called the voting set which consists of processes that act as arbiters. The algorithm uses only \(c\sqrt{N}\) messages to guarantee mutual exclusion, where \(N\) is the number of processes and \(C\) is a constant between 3 for light traffic and 5 for heavy traffic. This algorithm requires \(c\sqrt{N}\) messages to send a request, \(c\sqrt{N}\) messages to receive a permission, \(c\sqrt{N}\) messages to inquire the requesting process from the locked processes (the INQUIRE message is sent to a process \(i\) that has requested a process \(j\) if \(j\) receives another request that predates that of \(i\).), \(c\sqrt{N}\) messages from the requesting process if it fails to lock all its members (i.e. its quorum), and \(c\sqrt{N}\) messages to release the critical section.
As with any other distributed system, each process can issue a mutual exclusion request at an arbitrary time. In order to arbitrate these requests, any pair of two requests must be known to at least one "arbitrator". Since processes themselves must serve as arbitrators, any pair of two requests must reach a certain common process. The algorithm assumes that process i must obtain permission enter its mutual exclusion from every member of a subset Si of the processes of the network. Thus there must exist at least one common process in the intersection of every pair of sets of processes Si and Sj for any i and j so that the common process can serve as an arbitrator. Formally: for any combination of i and j, 1≤i,j≤N, S_i∩S_j≠0.

The selection of Si’s is not unique. There exists a number of ways to select a set of Si’s. The choice of Si’s affects the number of messages required to acquire mutual exclusion. It is desirable to have Si’s that are symmetric and of which the size of each subset is small. The problem of finding a set of Si’s is equivalent to finding a finite projective plane of N points². If a corresponding finite projective plane does not exist or if N is not expressed as |S_i| (|S_i| - 1) + 1 the algorithm presents other methods of generating S_i. One such method uses a grid the size of L x L, numbering the L² grid points from 1 to L². A subset S_i is defined to be the set of grid points on the row or the column passing through point i. Then it is clear that S_i ∩ S_j ≠ 0 for any i and 1 ≤ j, i ≤ L². The set of S_i’s is symmetric in the sense that |S_i| = 2L – 1 for any i and that any i is contained in (2L – 1) subsets. In other words, whenever two processes i and j want to enter the critical section, the arbiter process will grant access to only one of them at a time, and thus the mutual exclusion property is satisfied.

Table 1. Message complexity required to acquire the critical section by each algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>A Fully Centralized Algorithm</th>
<th>Suzuki – Kasami Token based algorithm</th>
<th>Ricart Agrawala Permission based algorithm</th>
<th>Maekawa Permission Based Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>The total number of messages required to acquire permission to enter the critical section</td>
<td>O(1)</td>
<td>N</td>
<td>2^(N-1)</td>
<td>(C\sqrt{N}) (C ∈ {3,4,5})</td>
<td></td>
</tr>
<tr>
<td>Details</td>
<td>1 Request</td>
<td>N-1 REQUEST 1 PRIVILEGE</td>
<td>N-1 REQUEST N-1 REPLY</td>
<td>(C\sqrt{N}) REQUEST (C\sqrt{N}) PERMISSION (C\sqrt{N}) RELEASE Potentially (C\sqrt{N}) INQUIRE (C\sqrt{N}) FAIL</td>
<td></td>
</tr>
</tbody>
</table>

² Although it is known that a finite projective plane of order |S_i| exists if |S_i| is a power of a prime, very little is known about general finite projective planes for other values of |S_i|. The Bruck-Ryser theorem [1] is the only result in this direction, and states that there exists no finite projective plane of order |S_i| if either |S_i| - 1 or |S_i| - 2 is divisible by 4 and if |S_i| cannot be expressed as the sum of two integral squares (|S_i| ≠ a² + b²) for a and b nonnegative integers.
3. Distributed Data Structures

We have 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 underline locking mechanize. The locking used can be implemented using any one of the three mutual exclusion algorithms described in the previous section. In addition, 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 best mutual exclusion algorithm to be use for locking.

A shared counter is a distributed object that provides a fetch-and-increment operation in a distributed system, where fetch-and-increment is defined as an atomic operation that atomically increments by one the value of a variable and returns its previous value. We present two different implementations of a shared counter: notification based and token based.

The basic premise employed in the notification based implementation of a shared counter is that after every increment of the counter every process is always updated with the latest value of the counter. A process that tries to increment the shared counter is able to do so only when it enters its critical section. Whenever a process raises the counter value it sends a notification message to the rest of the processes, informing them of the new counter value. Thus in a quiescent state, a state in which nobody tries to increment the counter, every process knows the value of the counter.

In a token based implementation of a shared counter, the counter's value is stored within a token object. A process wishing to raise the counter's value needs to acquire the token by entering its critical section and sending a GET_TOKEN message to the rest of the processes in the network. Whenever a process that holds the token receives the message, it immediately replies by sending the token to the requesting process. A process that receives the token can increment the counter's value as long as the counter remains in its possession.

A distributed queue is a linearizable data structure that supports enqueue and dequeue operations, by several processes, with the usual queue semantics. Linearizability implies that each operation appears to take place instantaneously at some point in time, and that the relative order of non-concurrent operations is preserved. We have implemented a distributed
FIFO queue which consists of local queues residing in the individual processes participating in the distributed queue. In addition a shared counter is used. Each element in a queue has a timestamp that is generated using the shared counter. An ENQUEUE operation is carried out by raising the distributed queue's shared counter value by one and enqueuing an element in the local queue along with the counter's value. A DEQUEUE operation is carried out by acquiring mutual exclusion using one of the locking algorithms and locating the process that holds the element with the lowest timestamp and removing this element from this process' local queue.

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 LIFO stack which is similar to the distributed queue. It consists of local queues residing in the individual processes participating in the distributed stack and a shared counter. Each element in the stack has a timestamp that is generated by the shared counter. A PUSH operation is carried out by raising the distributed stack's shared counter value by one and enqueuing the element in the local queue along with the counter's value. A POP operation is carried out by acquiring mutual exclusion using one of the locking algorithms and locating the process that contains the element with the highest timestamp and removing it from its local queue.

A distributed Linked List represents a form of non linear data structure. In other words, elements can be inserted and removed from any point in the list. The essence of a linked list is to represent sequences of elements, all having the same type. We have implemented a list which consists of a sequence of elements, each containing an arbitrary data field and two references (“links”) pointing to the next and previous elements. Each element of the list can reside in any process in the network. The distributed list also support other operations such as “traverse list”; “remove element” 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. As with the other distributed data structures presented above, manipulating the list requires that mutual exclusion be acquired. A process that needs to add an element to the head of the list acquires the mutual exclusion using one of the locking algorithms and sends a request for the “head pointer” to the rest of the processes in the network. 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” adding 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”). Removing an element from the head of list is done much the same way. Adding or removing elements from the list requires a process to acquire mutual exclusion, traverse the list and manipulate the elements its elements. A process is able to measure the size of the list by entering its critical section and querying the other processes about the size of their local lists.

4. The Experiments Framework

As already mentioned, we have implemented five lock-based distributed data structures (two types of counters, a queue, a stack and a linked list) as well as three distributed mutual exclusion locks. Each one of the five data structures was run using the three distributed locks. We have determined what is the preferred mutual exclusion algorithm to be used as the underling locking mechanism.

More precisely we measured each data structure's performance (using separately each one of the distributed mutual exclusion algorithms implemented) by running each data structure on a network with one, five, ten, fifteen and twenty processes (i.e. nodes). 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 shared token counter that employs Maekawa as its underling locking mechanism.

![Execution Flow](image)

**Figure 1. Execution Flow: a layered description of our implementation.**
All tests were implemented using Coridan's messaging middleware technology which is known as MantaRay. MantaRay is a reliable messaging framework that uses multicasting to discover other processes running MantaRay in a network, allowing new machines (processes) to be added or removed from the topology. Once a process joins the network it can immediately start to send messages and receive messages from other processes already running. MantaRay's server-less architecture was designed specifically for heterogeneous, distributed, and high-traffic environments such as the one used in the experiments.

MantaRay communication mechanism can be broadly classified as either point-to-point or publish-subscribe. Publish and subscribe (Pub/Sub) processes address messages to some process in a content hierarchy. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a process’s multiple publishers to its multiple subscribers. Point-to-point (PTP) products are built around the concept of message queues. Each message is addressed to a specific queue; processes extract messages from the queue(s) established to hold their messages. Our experiments use both modes of communication (Pub/Sub and PTP).

Peer-to-Peer versus Broker Communication in a network
With a broker-based architecture, processes in the network are only able to send messages to one other process; namely the broker process. The broker process on the other hand is able to send messages to all other processes. When process A needs to send a message to process B, it sends it to the broker and the broker sends it to process B, and vice versa.

![Figure 2. Processes communication in a centralized environment.](image_url)

MantaRay's fully distributed architecture does not use a broker process to facilitate message transfer among processes. Rather processes running in the network are aware of one another and as a result are able to send messages back and forth directly. Processes discover each other automatically by using a multicast channel which is used to send relevant discovery information, such as IP, port, and even process ID on a common multicast address. All processes listen to this channel and receive information about other running processes.
Thus for example, after process A has discovered process B, it is able to open a TCP connection to it and start communicating immediately.

![Diagram showing communication between Process A and Process B](image)

**Figure 3. Communication in a distributed environment**

MantaRay is extensively used by various commercial companies and academic institutes around the world. Current commercial implementations range from Aircraft controlling systems in the DOD to stock trading systems to VoIP providers to healthcare organizations. MantaRay is also used in academic projects such as Dragon Soft’s Try Again application [24].

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

5. Results

To insure the accuracy of the experimental results, we have used the testing machines exclusively and prevented other users from accessing them during the experiments. The algorithms were compiled at the highest optimization level, and were carefully optimized. We tested each of the implementation in hours (sometimes days long) executions on various number of processes (machines). It was during this process that we discovered the race condition mentioned in section 2.2.

All the experiments done on the data structures we have implemented employed an initially empty data structure (queue, stack etc.) to which processes performed a series of operations. 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 due to low process context switching. The time a process took to complete the “something else” was not reported in our figures.

In sections 5.1, 5.2, 5.3, 5.4 we present the result of the experiment done on our shared counters, distributed queues, distributed stacks and linked lists respectively. In section 5.5 we summarize our conclusions.

**5.1 Counters**

Our two shared counter implementations are optimized towards either a read or a write operations (i.e. update operation). The notification counter gives priority to read operations as each process is always updated with the latest values of the counter by the updating process. The Token counter on the other hand gives priority to write operations as processes that possess the token can continuously update the counter value. Reading the counter's value on the other hand requires the process to acquire the token.

Figure 4 shows the average net execution time (when omitting the 30 mSecods delay) for one count up operation. More precisely, for n processes, the graph shows the time one process spends performing a single count up operation averaged over 1 million operations for each process using each of the three locking algorithms implemented.

Looking at the graph we see that notification counter when using the Ricart-Agrawala locking algorithm perform worse than when using either the Suzuki-Kasami or Maekawa algorithms. We also see that the Suzuki-Kasami algorithm performs almost as well as Maekawa’s algorithm when the number of process is less than 20 although Maekawa’s algorithm is always slightly better. We believe that as the number of processes gets larger Maekawa’s algorithm will out perform the Suzuki–Kasami algorithm as the amount of network communication will be noticeably lower. Figure 5 plots the same quantity as figure 4, but for the token counter. Figure 5 shows that the token counter behaves and scales better than the notification counter when the number of processes grows.
In order to further verify the above results for our shared counters experiments, we have also implemented and tested two more additional distributed data structures (Queue and Stack) that made use of a shared counter.

5.2 Queues

Figure 6 shows the average net execution time (when omitting the 30 ms delay) for one enqueue operation in a distributed queue which uses a notification counter in order to generate the unique timestamp associated with each object which is enqueue in the queue. More precisely, for n processes, the graph shows the time one process spends performing an enqueue operation averaged over 1 million operations for each process using each of the three locking algorithms implemented.

Figure 7 shows the average net execution time (when omitting the 30 ms delay) for one million enqueue operations in a distributed queue which uses a token counter in order to generate the unique timestamp associated with each object which is enqueue in the queue. More precisely, for n processes, the graph shows the time one process spends performing an enqueue operation averaged over 1 million operations for each process using each of the three locking algorithms implemented.

As with the counter tests we run, queues using the Ricart-Agrawala locking algorithm perform worse than queues that use either the Suzuki-Kasami or Maekawa algorithms. We also see that the Suzuki-Kasami algorithm performs almost as well as Maekawa’s algorithm.
when the number of process is less than 20 although Maekawa’s algorithm is always slightly better. We believe that as the number of processes gets larger Maekawa’s algorithm will out perform the Suzuki–Kasami algorithm as the amount of network communication will be noticeably lower.

![Figure 6. Time one process spends performing a single enqueue operation averaged over 1 million operations in a queue employing a notification counter.](image)

![Figure 7. Time one process spends performing a single enqueue operation averaged over 1 million operations in a queue employing a token counter.](image)

Figure 8 shows the average net execution time (when omitting the 30 mSecoddns delay) for one million dequeue operations in a distributed queue. More precisely, for n processes, the graph shows the time one process spends performing a dequeue operation averaged over 1 million operations for each process using each of the three locking algorithms implemented.

![Figure 8. Time one process spends performing a single dequeue operation averaged over 1 million operations from a queue.](image)
5.3 Stacks

The results of our distributed stack experiments, which are presented below, are identical to the ones we got in the distributed queue experiments. Figure 9 shows the average net execution time (when omitting the 30 mSecodns delay) each process spends pushing one million objects into a distributed stack which uses a notification counter in order to generate the unique timestamp associated with each object which is pushed to the stack. More precisely, for n processes, the graph shows the time one process spends performing a push operation averaged over 1 million operations for each process using each of the three locking algorithms implemented.

Figure 10 shows the average net execution time (when omitting the 30 mSecodns delay) for one million push operations in a distributed stack which uses a token counter in order to generate the unique timestamp associated with each object which is pushed to the stack. More precisely, for n processes, the graph shows the time one process spends performing a push operation averaged over 1 million operations for each process using each of the three locking algorithms implemented.

As with the counter tests we run, stacks using the Ricart-Agrawala locking algorithm perform worse than stacks that use either the Suzuki-Kasami or Maekawa algorithms. We also see that the Suzuki-Kasami algorithm performs almost as well as Maekawa’s algorithm when the number of process is less than 20 although Maekawa’s algorithm is always slightly better. We believe that as the number of processes gets larger Maekawa’s algorithm will out perform the Suzuki–Kasami algorithm as the amount of network communication will be noticeably lower.

![Figure 9. Time one process spends performing a single push operation averaged over 1 million operations in a stack employing a notification counter.](image)

![Figure 10. Time one process spends performing a single push operation averaged over 1 million operations in a stack employing a token counter.](image)
Figure 11 shows the average net execution time (when omitting the 30 mSecodns delay) for one million pop operations in a distributed stack. More precisely, for n processes, the graph shows the time one process spends performing a pop operation averaged over 1 million operations for each process using each of the three locking algorithms implemented.

![Stack - Pop Operation](image)

Figure 11. Time one process spends performing a single pop operation averaged over 1 million operations from a stack.

5.4 Linked Lists

The Linked list we have implemented does not make use of a shared counter. Rather it uses the locking algorithm directly to acquire a mutual exclusion before manipulating the list itself. Figure 12 shows the net execution time (when omitting the 30 mSecodns delay) for one million add operations in a distributed linked list. More precisely, for n processes, the graph shows the time one process spends performing an add operation averaged over 1 million operation for each process using each of the three locking algorithms implemented.

![Linked List - Add / Remove Operations](image)

Figure 12. Time one process spends performing a single add operation averaged over 1 million operations to a linked list.

21
5.5 Conclusions

The experiments we run lend us to the following conclusion: (1) Generally speaking the Ricart - Agrawala mutual exclusion algorithm is a poor choice for a locking algorithm. Whereas Maekawa's algorithm always performs at least as fast as any other locking algorithm we have tested (figures 8, 11 and 12). (2) when the number of processes is small (usually less than 5 processes) the locking algorithms we tested exhibited minor performance differences. Thus when the number of processes using the locking algorithm is small it is advisable to use an algorithm with which is easier to implement and debug than a complex one. When the number of processes larger than 5, Maekawa's algorithm is a much better choice (figures 4 through 12). (3) When more than ten processes try to use a shared counter in a mostly write oriented application the Token Counter is always the counter of choice (figures 4 and 5). (4) As expected all the lock based data structures we tested exhibited increasing performance degradation as the number of number of processes participating in the simulation grew (figures 4 through 12).

6 Related Work

Edsger Dijkstra first raised the question of synchronization between processes where more than one process is competing for access to shared resource in 1971. Locking a resource by achieving mutual exclusion is a common technique to ensure the resource is accessed by only one process or piece of code at a time. The creation of mutual exclusion in a computer network under distributed control is not trivial. While many papers have considered implementation of mutual exclusion [2,6,7,11,12,13,16,20,21], the earliest algorithm for mutual exclusion in a computer network was proposed by Lamport [6] and is based on Lamport’s notion of logical clocks. It requires approximately 3* (N - 1) messages to be exchanged per critical section invocation, where N is the number of processes. Ricart and Agrawala [13] propose a symmetrical, fully distributed mutual exclusion algorithm which was said to be optimal in the sense that a symmetrical, distributed algorithm cannot use fewer messages if requests are processed by each process concurrently. The algorithm uses only 2* (N - 1) messages between processes. As each process is required to gain permission from all processes the algorithm is based on a unanimous consensus rule. Suzuki and Kassami [20] propose an algorithm that realizes mutual exclusion among N processes. The algorithm requires at most N messages to be exchanged for one mutual exclusion invocation. The proposed algorithm is a counterexample to the optimal algorithm Ricart and Agrawala.
proposed. A drawback of the algorithm is that the sequence numbers contained in the messages are unbounded. A way to overcome this limitation (at the cost of increased message traffic) is discussed.

Thomas’s voting technique used in [14] is based on a majority consensus rule and requires that a process requesting mutual exclusion obtain a permission vote from only a majority of the processes. Maekawa [7] proposes a distributed, symmetric algorithm that is said to require only C*SQRT (N) messages to create mutual exclusion in a computer network, where N is the number of processes and C a constant between 3 and 5. The approach taken by Maekawa parallels the voting technique used in Thomas. It also uses deferral, the technique used in Ricart and Agrawala. An additional technique, relinquishment, is used, however, to avoid deadlocks. Ye-In [22] shows that under certain circumstances when requests arrive out of order in the timestamp sequence most of the time, more than N messages may be needed to resolve the possible deadlocks in Maekawa’s algorithm.

Raymond [12] presents an algorithm that uses a spanning tree of the computer network where the number of messages exchanged per critical section depends on the topology of this tree. A process trying to access the critical section is only required to communicate with neighboring processes of the spanning tree. Typically, under light load, the number of messages exchanged is O(logN) and it reduced to approximately four messages under heavy load.

Singhal [18,19] presents a taxonomy of mutual exclusion in distributed systems where all communication is done using message passing. Singhal classifies these algorithms based on their distinct features and commonalities such as token based, non token based or hybrid solutions.

In a token based algorithm, a unique token is shared among the processes. A process is allowed to enter its critical section if it possesses the token. Token based algorithms are further classified into broadcast-based and logical-structure-based algorithms. Both types are further classified into two sub categories; static and dynamic. Static algorithms are then broken down into ring, tree and graph classifications. [8] is an example of a static algorithm in which processes are organized in a ring structure. A token circulates the ring serially from one process to the other. A process must wait to capture the token before entering its critical section. In dynamic algorithms the logical structure changes as processes requests and
executes the critical section. Niami-Trehel [11] is such an algorithm. Each process maintains a variable “last” which records the ID of the last process that had sent the request message to this process. The tree structure of the system is defined by the last variable at each process. Singhal [16] is another example of a dynamic token based algorithm, which uses heuristics to help each process maintain information about the state of other processes in the system. Processes then use that information to select a set of processes that are likely to have the token. Other examples of token based algorithms are [21]

Non token based algorithms require one or more successive rounds of message exchange among the processes in order to obtain the permission to execute the critical section. Non token based algorithms are classified as either “Ricart-Agrawala”, “Maekawa” or “generalized” types. Singhal finds that token based mutual exclusion algorithms are in general more message-efficient that permission based ones and that there is a trade off between speed and message complexity of the algorithm. Singhal [17] has been the first to propose a dynamic Ricart-Agrawala type algorithm whose information-structure evolves with time as processes learn about the state of the system. Other examples of non token based algorithms are [2, 6].

FIFO Queues and LIFO stacks are among the most fundamental concurrent data structures that exist today. These Data structures can be classified into two classes; locked-based data structures and wait-free ones. The problem of constructing a wait-free implementation of one data structure lies at the heart of much recent work in concurrent algorithms. Maurice [10] shows that simple universal objects exist from which, one can construct a wait-free implementation of any sequential data structure. He also derives a hierarchy of data structures such that no data structure at one level has a wait-free implementation in terms of data structures at lower levels. Michael and Scott’s lock-free FIFO queue algorithm [9] is one of the most effective and practical concurrent queue implementation. The algorithm is included in the standard Java concurrency package [25]. Mozes and Shavit [5] present an optimistic approach to lock-free FIFO queues which is said to perform consistently better than Michael and Scott’s queue.

7 Conclusion

Distributed data structures such as shared counters, queues and stacks are ubiquitous in parallel programs and their performance is a matter of concern. We have presented the
relation between classical distributed mutual exclusion locks and specific distributed data structures which use locking for synchronization; for example a distributed queue implementation using a token based shared counter and Suzuki-kasami’s mutual exclusion algorithm.

Our findings show: (1) that Maekawa's locking algorithm outperforms any of the other locking algorithms we have tested. (2) Out of the two types of shared counters we suggested and implemented, the token counter clearly outperforms the other one. (3) As a result of using locking, the data structured tested exhibit performance degradation as the number of processes grows.

8 Future Work

We have laid down the foundations for a testing suit which can be used to measure the relation between distributed mutual exclusion locks and specific distributed data structures. Future work would entail designing and implementing modern locking algorithms such as Carvalho-Roucairol Mutual Exclusion algorithm [2], Neilsen-Mizuno algorithm for mutual exclusion [8] and fault tolerant algorithms that do not assume an error free network [4,15]. More complex lock-based distributed data structures such as hash tables, heaps and different implementations of the existing data structures we have already implemented as well as data structures that use fine grained locks as opposed to the coarse grained locks presented here.

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


