CAS-Based Lock-Free Algorithm for Shared Deques

Maged M. Michael

IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA
magedm@us.ibm.com

Abstract. This paper presents the first lock-free algorithm for shared double-ended queues (deques) based on the single-address atomic primitives CAS (Compare-and-Swap) or LL/SC (Load-Linked and Store-Conditional). The algorithm can use single-word primitives, if the maximum deque size is static. To allow the deque's size to be dynamic, the algorithm employs single-address double-width primitives. Prior lock-free algorithms for shared deques depend on the strong DCAS (Double-Compare-and-Swap) atomic primitive, not supported on most processor architectures. The new algorithm offers significant advantages over prior lock-free shared deque algorithms with respect to performance and the strength of required primitives. In turn, lock-free algorithms provide significant reliability and performance advantages over lock-based implementations.

1 Introduction

The double-ended queue (deque) object type [11] supports four operations on an abstract sequence of items: PushRight, PudhLeft, PopRight and PopLeft. PushRight (PushLeft) inserts a data item onto the right (left) end. PopRight (PopLeft) removes and returns the rightmost (leftmost) data item, if any. A shared object is *lock-free* [8] if it guarantees that whenever a process executes some finite number of steps towards an operation on the object, *some* process (possibly a different one) must have completed an operation on the object, during the execution of these steps. Therefore, unlike conventional lock-based objects, lock-free objects are immune to deadlock even with process (fail stop) failures, and offer robust performance even with arbitrary process delays.

Prior lock-free algorithms for shared deques [1,4,5] depend on the strong DCAS (Double-Compare-and-Swap) atomic primitive, which is not supported on most current processor architectures, and its simulation using weaker widely-supported primitives such as CAS (Compare-and-Swap)[1] or LL/SC (Load-Linked and Store-Conditional)[2] entails significant performance overhead.

[1] CAS(addr,exp,new) performs {*if (*addr* $\neq$ *exp) return false; *addr* $\leftarrow$ *new; return true;*} atomically. DCAS is similar to CAS, but operates on two independent memory locations.

[2] LL(addr) returns *addr. SC(addr,new) performs {*if (*addr was written by another process since the last LL(addr) by the current process) return false; *addr* $\leftarrow$ *new;*

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 651–660, 2003.
© Springer-Verlag Berlin Heidelberg 2003

This paper presents the first CAS-based lock-free algorithm for shared deques. The general structure of our implementation is a doubly-linked list, where each node contains pointers to its right and left neighbors, and includes a data field. The two ends of the doubly-linked list are pointed to by two anchor pointers. The two anchor pointers along with a three-value status tag occupy one memory block that can be manipulated atomically by CAS or LL/SC. The status tag indicates whether the deque is stable or not. When a process finds the deque in an unstable state, it must first attempt to take it to a stable state before attempting its own operation. The algorithm can use single-word CAS or LL/SC, if the maximum size of the deque is static. To allow the deque's size to be dynamic, the algorithm uses single-address double-width versions of these primitives.

```
// Code in bold type is for memory management.
// Types and structures
define StatusType = {STABLE, RPUSH, LPUSH}
structure NodeType { Right,Left:⟨*NodeType, integer⟩; Data: DataType; }
define AnchorType = ⟨*NodeType,*NodeType,StatusType⟩
// Shared variables
Anchor : AnchorType; // initially Anchor=⟨null,null,STABLE⟩
```

Fig. 1. Types and structures.

2 The Algorithm

2.1 Structures

Figure 1 shows the data structures used by the algorithm. The deque is represented as a doubly-linked list. Each node in the list contains two link pointers, Right and Left, and a data field. A shared variable, Anchor, holds the two anchor pointers to the leftmost and rightmost nodes in the list, if any, and a three-value status tag. Anchor must fit in a memory block that can be read and manipulated using CAS or LL/SC, atomically. Initially both anchor pointers have null values and the status tag holds the value STABLE, indicating an empty deque.

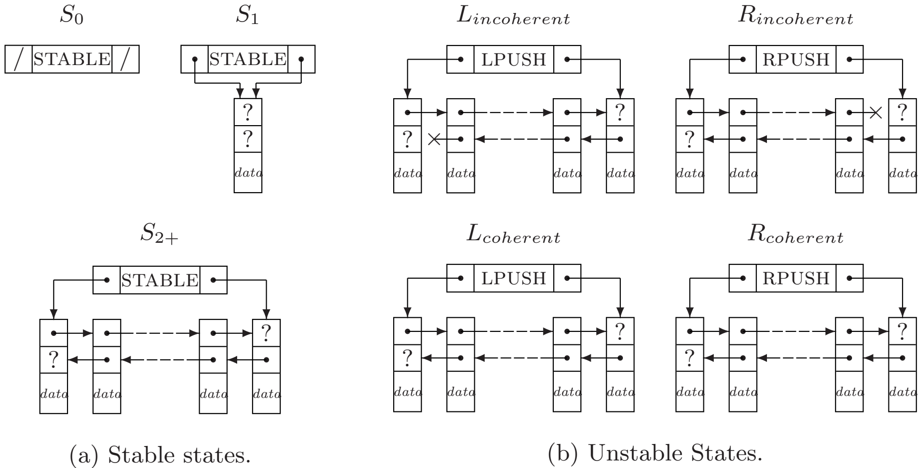
The status tag serves to indicate if the deque is in an unstable state. When a process finds the deque in an unstable state, it must first attempt to take it to a stable state before attempting its own operation.

The algorithm can use single-word CAS or LL/SC. Nodes can be preallocated in an array, which length can be chosen such that both anchor pointers and the status tag can fit in a single-word. However, in order to allow the deque to be completely dynamic, with arbitrary size, we employ single-address double-width CAS or LL/SC.

return true;} atomically. However, due to practical considerations, processor architectures that support LL/SC prohibit nesting or interleaving LL/SC pairs, and occasionally but not infinitely often, allow SC to fail spuriously.

2.2 Valid States

Figure 2(a) shows the three stable deque states. A deque is stable only if the status tag holds the value STABLE. A deque can be stable only if it is coherent. That is, for all nodes $x\hat{}$ in the deque's list $x\hat{}.Right\hat{}.Left = x$, unless $x\hat{}$ is the rightmost node; and $x\hat{}.Left\hat{}.Right = x$, unless $x\hat{}$ is the leftmost node. Empty and single-item deques are always stable (states S_0 and S_1, respectively). A deque is in the S_{2+} state if it is stable and contains two or more items.



(a) Stable states.

(b) Unstable States.

Fig. 2. Valid states.

Figure 2(b) shows the four unstable but valid deque states. A deque can be unstable only if it contains two or more items. It is acceptable for one end of the deque to be incoherent, as long as the status tag indicates an unstable deque on the same end. The list, excluding the end nodes, is always coherent. The deque is right-incoherent if $r\hat{}.Left\hat{}.Right \neq r$, where $r\hat{}$ is the rightmost node. The deque is left-incoherent if $l\hat{}.Right\hat{}.Left \neq l$, where $l\hat{}$ is the leftmost node. The deque is in state $R_{incoherent}$ ($L_{incoherent}$) when the deque is righ-incoherent (left-incoherent), the rest of the list is coherent, and the status tag is RPUSH (LPUSH). At most one end of the deque can be incoherent at a time. Unstable states result from push operations, but never from pop operations, hence the naming of the unstable status tag values. It is also acceptable for the status tag to indicate an unstable deque even when the deque is in fact coherent. Such is the case for the $R_{coherent}$ and $L_{coherent}$ states.

2.3 Operations

For clarity, we assume for now perfect memory management. However, due to the limited available space and the importance of memory management we include

code (in bold type) related to other memory management methods, that we discuss only in the following subsection.

```
    PushRight(data:DataType) {
        // For simplicity, assume NewNode() always returns a new node.
        node ← NewNode();
        node^.Data ← data;
        while true {
1:          ⟨l,r,s⟩ ← Anchor;
            if r = null {
2:              if CAS(&Anchor,⟨l,r,s⟩,⟨node,node,s⟩) return;
            } else if s = STABLE {
3:              node^.Left ← ⟨r,anyvalue⟩;
4:              if CAS(&Anchor,⟨l,r,s⟩,⟨l,node,RPUSH⟩)
                    {StabilizeRight(⟨l,node,RPUSH⟩); return;}
            } else Stabilize(⟨l,r,s⟩);
        }
    }
```

Fig. 3. PushRight algorithm.

Figure 3 shows the algorithm for the PushRight operation. If the deque is empty (i.e., in state S_0), the operation is completed in one atomic step (line 2) by setting both anchor pointers to the address of the new node containing the new item. Otherwise, if the deque is unstable, it must be stabilized first using the Stabilize routine in Figure 4, before attempting the push operation.

If the deque is stable and contains one or more items (i.e., in states S_1 or S_{2+}), the push operation is completed in multiple steps. The first step (line 4) is to swing the right anchor pointer to the new node and to indicate a right-unstable deque in the status tag, atomically, after setting the Left pointer of the new node to point to the rightmost node in the deque (line 3). After this step, the deque is unstable and most likely right-incoherent. It is possible for a successful CAS in line 4 to take a stable deque in state S_1 or state S_{2+} to state $R_{coherent}$ directly, if the Right pointer of the old rightmost node in the list happens to hold the address of the newly pushed node. The next step of a PushRight operation is to stabilize the deque by calling StabilizeRight.

If either of the CAS operations in lines 2 and 4 fails, the process restarts the push attempt by reading a fresh snapshot of Anchor in line 1. Throughout the algorithm, the state of the deque changes only on the success of CAS operations.

The success of the CAS in line 2 is the linearization point of a RightPush into an empty deque, and the success of the CAS in line 4 is the linearization point of a RightPush into a non-empty deque.

The StabilizeRight routine (Figure 4) involves two main stages. The first stage (lines 5–9) serves to guarantee that the Right pointer of the old rightmost node (i.e., the left neighbor of the newly pushed node) points to the new node.

At the end of that code segment, either the deque is in state $R_{coherent}$, or it must have been in that state at some time since the success of the latest push in line 4 by the current process. The final stage in StabilizeRight is to attempt to set the status tag to STABLE in line 10 and take the deque to state S_{2+}.

```
Stabilize(⟨l,r,s⟩:AnchorType) {
    if s = RPUSH
        StabilizeRight(⟨l,r,s⟩);
    else // s = LPUSH
        StabilizeLeft(⟨l,r,s⟩);
}

StabilizeRight(⟨l,r,s⟩:AnchorType) {
        // hp0, hp1 and hp2 are private pointers to three of the process'
        // hazard pointers.
        *hp0 ← l; // for memory management only.
        *hp1 ← r; // for memory management only.
        if Anchor ≠ ⟨l,r,s⟩ return; // for memory management only.
5:      ⟨prev,dontcare⟩ ← r^.Left;
        *hp2 ← prev; // for memory management only.
6:      if Anchor ≠ ⟨l,r,t⟩ return;
7:      ⟨prevnext,t⟩ ← prev^.Right;
        if prevnext ≠ r {
8:          if Anchor ≠ ⟨l,r,s⟩ return;
9:          if ¬CAS(&prev^.Right,⟨prevnext,t⟩,⟨r,t+1⟩) return;
        }
10:     CAS(&Anchor,⟨l,r,s⟩,⟨l,r,STABLE⟩);
}
```

Fig. 4. Stabilize and StabilizeRight routines.

In order to avoid race conditions that may corrupt the object, whenever a change in the deque is detected, the process ends its stabilization attempt. The algorithm requires any process that finds the deque in an unstable state to stabilize it first before attempting its intended operation. Thus, any changes in Anchor detected in line 6 or line 8 guarantee that some other processes must have already stabilized the deque.

The order of steps in the algorithm is very delicate, and especially in the StabilizeRight (and StabilizeLeft) routine. The condition in line 6 guarantees that the pointer value *prev* read in line 5 is a valid pointer as it guarantees that at that time, the deque contained two or more nodes, and that the node r^ was part of the deque.

The condition in line 8 is needed to prevent the process from corrupting the deque if, for instance, between executing lines 6 and 7, some other process stabilized the deque, popped the node pushed by the original process, and then completed another PushRight operation taking the deque to state S_{2+}. Without

the condition in line 8, if the original process resumes execution, it will reach line 9 and its CAS operation will succeed, resulting in the corruption of the deque. The deque would be corrupted as its status tag becomes STABLE while its list is right-incoherent. This is not a valid state. A succession of PopLeft operations can cause the left anchor pointer to point to a node that is no longer part of the deque's list.

```
PopRight():DataType {
    while true {
11:    ⟨l,r,s⟩ ← Anchor;
       if r = null return EMPTY;
       if r = l {
12:       if CAS(&Anchor,⟨l,r,s⟩,⟨null,null,s⟩) break;
       } else if s = STABLE {
          *hp0 ← l; // for memory management only.
          *hp1 ← r; // for memory management only.
          if Anchor ≠ ⟨l,r,s⟩ continue; // for memory management only.
13:       ⟨prev,dontcare⟩ ← r^.Left;
14:       if CAS(&Anchor,⟨l,r,s⟩,⟨l,prev,s⟩) break;
       } else Stabilize(⟨l,r,s⟩);
    }
    data ← r^.Data;
    RetireNode(r); // definition depends on memory management method
    return data;
}
```

Fig. 5. PopRight algorithm.

Figure 5 shows the PopRight algorithm. If the deque is empty (i.e., in state S_0) a value EMPTY is returned. Pop operations on non-empty deques take exactly one successful CAS operation to complete. If the deque contains one item (i.e., in state S_1), the operation is completed in one atomic step (line 12) by setting both anchor pointers to null. Otherwise, if the deque is unstable it is stabilized first before attempting the pop. If the deque is stable and contains two or more items (i.e., in state S_{2+}), the pop operation is completed in one atomic step (line 14) by swinging the right anchor pointer to the left neighbor of the rightmost node.

Reading Anchor in line 11 is the linearization point of a RightPop operation on an empty deque. The success of the CAS in line 12 is the linearization point of a RightPop from a single-item deque, and the success of the CAS in line 14 is the linearization point of a RightPop from a multiple-item deque.

The deque algorithm is symmetric. The PushLeft, PopLeft, and StabilizeLeft routines are similar to the corresponding right side routines.

On architectures that support LL/SC (PowerPC, MIPS and Alpha) but not CAS, implementing *CAS(addr,exp,new)* using the following routine suffices for

the purposes of this algorithm. { *repeat* { *if LL(addr)* $\neq$ *exp return false;* } *until SC(addr,new); return true;* }

2.4 Memory Management

The freeing and reuse of nodes removed (popped) from a deque object involve two related but different issues: memory reclamation and ABA prevention. The memory reclamation problem is how to allow the arbitrary reuse of removed nodes, while still guaranteeing that no process accesses free memory [13]. The ABA problem can occur if a process reads a value A from a shared location, then other processes change that location to B and then back to A, later the original process performs an atomic comparison (e.g., CAS) on the location and the comparison succeeds where it should fail, leading to corrupting the object [9]. For this algorithm, the ABA problem occurs only if a node is popped and then reinserted while a process holds a reference to it with the intent of using that reference as an expected value of an ABA-prone comparison.

The simplest method for preventing the ABA problem is to include a tag with each pointer to dynamic nodes that is prone to the ABA problem, such that both are manipulated atomically, and the tag is incremented when the pointer is updated [9]. In such a case, an otherwise ABA-prone CAS succeeds only if the tag has not changed since the current process last read the location (assuming that the tag has enough bits to make full wraparound between the read and the CAS practically impossible). This solution allows removed nodes to be reused immediately, but if used by itself it prevents memory reclamation for arbitrary reuse.

Unless atomic operations on three or more words are supported, the inclusion of ABA tags with the Anchor variable in an atomic block prevents the anchor pointers from holding arbitrary pointer values, thus limiting the maximum size of the deque. In such a case, nodes available for insertion in the deque can be allocated in an array. The size of the array can be chosen such that both anchor pointers, status tag, and ABA-prevention tag fit in an atomic block.[3]

Other memory management methods do not require the inclusion of an ABA-prevention tag with Anchor and thus allow dynamic deque size using double-width single-address primitives.

If automatic garbage collection is available, it suffices to nullify the contents of removed nodes to prevent accidental garbage cycles from forming, in case the fields of the removed node hold the addresses of other memory blocks that would be otherwise eligible for recycling. Garbage collection techniques prevent the ABA problem for this algorithm.

The hazard pointer method [13] allows the safe reclamation for arbitrary reuse of the memory of removed nodes and provides a solution to the ABA-problem for pointers to dynamic nodes without the use of per pointer tags or

[3] For example, if the atomic block size is 64 bits, then 32 bits can be dedicated to the ABA-prevention tag, two bits are occupied by the status tag, and each anchor pointer can occupy up to 15 bits. Thus, the deque size is at most $2^{15} - 1$ nodes.

per node reference counters. The method requires the target lock-free algorithm to associate a small number of *hazard pointers* (three for this algorithm) with each participating process. The method guarantees that no removed node is reused or freed as long as some process' hazard pointer has been pointing to it continuously from a time when it was in the object. By preventing the ABA problem for Anchor without using tags, it allows the anchor pointers to hold arbitrary (two-byte-aligned) pointer values, and hence allows the size of the deque and its memory use to grow and shrink arbitrarily, using double-width single-address CAS or LL/SC. The figures in the previous subsections show (in bold type) code related to memory management that uses hazard pointers, and for simplicity uses ABA tags for the side pointers of each dynamic node. Combined with hazard pointers, ABA tags do not prevent removed nodes from being freed for arbitrary reuse.

2.5 Complexity

In the absence of contention, each operation on the deque takes a constant number of steps. Under contention, we use the total work complexity measure. Total work is defined as the worst-case total number of primitive steps executed by P processes, during any execution of an arbitrary sequence of r operations on the shared object, divided by r [10]. The total work of any operation of the new algorithm, under contention by P processes is $O(P)$. Intuitively, the useful work to complete an operation is constant, and at most constant work for each of the other $P-1$ concurrent operations is deemed useless as a result of the success of the completed operation.

3 Discussion

3.1 Related Work

Universal lock-free methodologies (e.g., [8,17]) can be used to transform sequential object implementations including deques into shared lock-free implementations. However, the resulting lock-free implementations for almost all object types, including deques, are extremely inefficient. This motivated the development of object-specific algorithms.

The first lock-free shared deque algorithms were the linear (array-based) algorithms of Greenwald [5] using DCAS. Agesen *et. al.* [1] presented two DCAS-based algorithms. The first algorithm is array-based, offering improvements over Greenwald's. The other algorithm is dynamic and requires three DCAS operations per push and pop pair, in the absence of contention. Later, they presented a dynamic DCAS-based algorithm [4] that requires two DCAS operations per push and pop pair, in the absence of contention.

DCAS is not supported on most current processor architectures. Implementations of DCAS using CAS or LL/SC were proposed in the literature (e.g., [3, 10,17]). However, they all entail substantial overhead for every DCAS operation,

prompting many researchers to advocate the implementation of DCAS efficiently in hardware [5]. In fact, Agesen *et. al.* [1,4] state as the main motivation for developing their algorithms, justifying the need for hardware support for DCAS on future architectures. This paper shows that DCAS is not needed for the efficient implementation of lock-free shared deques.

It is worth noting that the algorithms of Agesen *et. al.* [1,4] allow execution overlap between one operation from the right with one operation from the left. On the other hand the new algorithm serializes operations on the deque as operations are serialized on the Anchor variable. However, even without concurrency, the new algorithm promises much higher throughput than the former algorithms, which at most allow the concurrency of two operations. Basic queuing theory determines that one fast server is guaranteed to result in higher throughput than two slow servers, if the response time of each of the slow servers is more than twice that of the fast server. Operations of the former algorithms, even without contention, are substantially slower than those of the new algorithm, taking into account simulation of DCAS using CAS or LL/SC.

Finally, it is worth noting that sometimes single-producer work queues are referred to in the literature as deques [2]. However, these objects do not actually support the full semantics of the shared deque object (i.e., without restricting the number or identity of operators). Furthermore, such objects do not even support the semantics of simpler shared object types such as LIFO stacks and FIFO queues that are subsumed by the semantics of the deque object type.

3.2 Summary

This paper was motivated by the significant reliability and performance advantages of lock-free algorithms over conventional lock-based implementations, and the practicality and performance deficiencies of prior lock-free algorithms for shared deques, which depend on the strong DCAS primitive. We presented the first lock-free algorithm for shared deques based on the weaker primitives CAS or LL/SC. The algorithm can use single-word CAS or LL/SC, if the nodes available for use in the deque are preallocated statically. In order to allow the size of the deque to grow and shrink arbitrarily, we employ single-address double-width CAS or LL/SC. The new algorithm offers significant advantages over prior lock-free algorithms for shared deques with respect to the strength of required primitives and performance. With this algorithm, the deque object type joins other object types with efficient CAS-based lock-free implementations, such as LIFO stacks [9], FIFO queues [14,15,19], list-based sets and hash tables [6,12, 16], work queues [2,7], and priority queues [18].

References

1. Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele, Jr. DCAS-based concurrent deques. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146, July 2000.

2. Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, June 1998.
3. Hagit Attiya and Eyal Dagan. Improved implementations of binary universal operations. *Journal of the ACM*, 48(5):1013–1037, September 2001.
4. David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele, Jr. Even better dcas-based concurrent deques. In *Proceedings of the 14th International Symposium on Distributed Computing, LNCS 1914*, pages 59–73, October 2000.
5. Michael B. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
6. Timothy L. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing, LNCS 2180*, pages 300–314, October 2001.
7. Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 280–289, July 2002.
8. Maurice P. Herlihy. A methodology for implementing highly concurrent objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
9. IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.
10. Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.
11. Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968.
12. Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, August 2002.
13. Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002.
14. Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.
15. Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, May 1994.
16. Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, July 2003.
17. Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
18. Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, April 2003.
19. John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, October 1994.