Space-Efficient Atomic Snapshots in Synchronous Systems*

Gil Neiger
Ranu Singh

GIT-CC-93/46

July 28, 1993

Abstract

We consider the problem of implementing an atomic snapshot memory in synchronous distributed systems. An atomic snapshot memory is an array of memory locations, one per processor. Each processor may update its own location or scan all locations atomically. We are interested in implementations that are space-efficient in the sense that they are honest. This means that the implementation may use no more shared memory than that of the array being implemented and that the memory truly reflect the contents of that array. If \( n \) is the number of processors involved, then the worst-case scanning time must be at least \( n \). We show that the sum of the worst-case update and scanning times must be greater than \( \lceil 3n/2 \rceil \). We exhibit two honest implementations. One has scans and updates with worst-case times of \( n + 1 \) for both operations; for scans, this is near the lower bound. The other requires longer scans (with worst-case time \( \lceil 3n/2 \rceil + 1 \)) but shorter updates (with worst-case time \( \lceil n/2 \rceil + 1 \)). Thus, both implementations have the sum of the worst-case times at \( 2n + O(1) \), which is within \( n/2 \) of the lower bound. Closing the gap between these algorithms and the combined lower bound remains an open problem.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

*This research was supported in part by the National Science Foundation under grant CCR-9106627.
1 Introduction

A common model of a distributed system is one in which processors communicate through a shared memory. The memory consists of a set of locations that can be read and written independently. Because processors operate independently and distributed algorithms often require a consistent view of the computation, researchers have developed the atomic snapshot abstraction. This abstraction supports two operations: a processor may update one location in memory or it may atomically scan the entire memory. These operations are required to be linearizable [7]; this means that the overall execution should be consistent with one in which the operations executed serially in an order consistent with the order of operations whose execution does not overlap in real time. There has been considerable research in the development of atomic snapshot memory. The problem was first explicitly considered by Afek et al. [1] and by Anderson [2]. Improved algorithms were developed later [3, 4, 8], with the fastest requiring $O(n \log n)$ low-level operations per update or scan. (Dwork et al. [5] studied a slightly weaker problem, while Gafni and Borowsky [6] considered a stronger version.)

This paper takes a new approach to this problem. First, we restrict our domain to synchronous systems. We assume that processors have access to a global clock and can execute in lockstep. This considerably simplifies the problem but does not trivialize it. We seek solutions that are highly efficient with respect to time and space. In previous implementations, scan and update operations required from $O(n \log n)$ to $O(2^n)$ low-level read/write operations, where $n$ is the number of processors in the system; this seems very expensive, especially for the simple update operations. We consider only solutions that require $O(n)$ low-level operations and seek to reduce the multiplicative constant as much as possible. Previous implementations also required a considerable amount shared memory; for example, the solution of Afek et al. requires $n$ shared variables each of size $O(n)$. We consider only solutions with the minimum space requirements. That is, if the atomic snapshot memory is implementing an array of $n$ locations of $b$ bits each, then we allow only implementations in which the shared memory consists of $n$ locations of $b$ bits each. Furthermore, we consider only implementations that are honest. These have the property that each cell of the implementing array must contain a value that actually was the parameter of some high-level update operation on that cell.

Consider any honest implementation of an atomic snapshot memory in a synchronous system. Let $U$ be the worst-case execution time of an update operation and $S$ be the worst-case execution time of a scan operation. We first show that $S \geq n$ (this is relatively obvious). We then show that $S + U > \lceil 3n/2 \rceil$; this shows that there must be a trade-off between optimizing the performance of scans and that of updates. We give two honest implementations that illustrate this trade-off. One has $U = S = n + 1$ and the other has and another with $U = \lceil n/2 \rceil + 1$ and $S = \lceil 3n/2 \rceil + 1$. Thus, both implementations require $S + U = 2n + O(1)$. Resolving the gap between the upper and lower bounds on the combined measures is an open problem.
2 Definitions, Assumptions, and Notation

This section describes our model of computation and defines the atomic snapshot problem.

A distributed system consists of a set of processors $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ that communicate through shared memory. We assume that the shared memory is a set of locations $\mathcal{M} = \{x_1, x_2, \ldots, x_m\}$; it is assumed that each memory location can hold a value from domain $D$, where $|D| = d$. Processors can execute two operations on the memory. A write operation, of the form "write $v$ into $x_i$," stores the number $v \in D$ in location $x_i$. A read operation, of the form "read $x_i$ into $\ell$," reads the value of location $x_i$ and stores it in a local variable $\ell$. For the sake of simplicity, we assume that each memory access takes exactly one time unit.

The memory is assumed to be linearizable [7]. This means that, in each execution, there is some total ordering of the memory accesses such that each read operation returns the value written by the most recent write to the same location and such that two operations that occur at different times are ordered according to their execution time. We seek to implement a higher level abstraction that is also linearizable, so it is reasonable to assume the linearizability of the underlying memory; this is a standard assumption that is made in all preceding work on atomic snapshots.

We consider systems that are synchronous. That is, time is divided into a series of discrete time units. Processors are assumed to share a global clock and thus always agree on the current time. Because of this, our programming notation make use of wait statements, whose meaning should be clear from context.

The synchronous abstraction described here is very strong and is not realized by most practical systems. However, Neiger and Toueg [9] have shown that it can be simulated in systems with approximately synchronized clocks, which can be practically implemented. (The results of Neiger and Toueg applied to systems with message passing; it is not hard to extend them to systems with shared memory.)

This paper considers implementations of atomic snapshot memory. Such a memory provides the abstraction of an array of memory cells. Two kinds of operations are permitted. An update operation is indexed by a value $i$ and takes a parameter $v$; $update_i(v)$ changes to the value of the $i$th cell to $v$. A scan operation $scan$ takes no parameters and returns an array of values, one for each cell in the array.

Updates and scans are implemented using the low-level read and write operations described above. Because of this, an execution of an update or a scan an execution interval that extends from the time at which it is invoked until that time at which it exits or returns. All implementations are required to be linearizable. Thus, for any execution, there must be a total order of the operations executed such that the following properties hold:

1. for each cell, each scan must return the value that was the parameter of the most recent update to the cell that preceded the scan in the total order; and

2. if the execution intervals of two operations do not overlap, then they must appear in the total order in the same order in which they actually executed.
We consider *honest* implementations of atomic snapshots. This means that the only shared memory that is permitted for the implementation is an array of \( n \) cells. We further require that the simulated cells can hold as many values as the implementing cells; that is, there is no additional storage permitted. Finally, we require that, at any point in any execution, the contents of the \( i \)th implementing cell must be the parameter of some \( \text{update}_i \) that was invoked by that point.

We require our implementations to be *fault-tolerant* in the following sense. Any processor must always be able to complete an operation (update or scan) within a finite amount of time, regardless of the failures of other processors. When any number of processors are allowed to fail by stopping, such implementations are called *wait-free*. All our implementations are easily shown to be wait-free. Note, however, that wait-freedom is not a difficult property to provide in synchronous systems.

We assume that the processors that update and the processors that scan form disjoint sets. In addition, each updating processor can update only one cell in the array. That is, cell \( i \) can be updated only processor \( p_i \). These assumptions are purely to simplify the exposition. The proven results will change only by small additive constants if we allow any processor to scan and to update any cell in the array.

To evaluate the performance of implementations of atomic snapshots, we are concerned with two measures. Let \( U \) be the length of the longest update in any execution of an implementation; define \( S \) similarly for scans. The following section presents some lower bound results regarding these measures.

## 3 Lower Bound Results

This section presents two lower-bound results regarding the worst-case execution times of the two snapshot operations. The first is relatively obvious:

**Theorem 1**: For any honest real-space implementation of atomic snapshots, \( S \geq n \).

**Proof**: Consider some implementation with any value for \( U \). Suppose that a processor performs one scan that ends at time \( t \) and does not begin another scan until after \( t + U \). It is possible that all processors invoked and completed update operations in the intervening time. For this reason, the scanning processor has no information about the contents of the simulated array. Because the implementation is honest, this means that all cells in the array must be read. This takes \( n \) time units. \( \square \)

The second result gives a bound on the combined update and scan complexities:

**Theorem 2**: For any honest implementation of atomic snapshots, \( U + S > \lfloor 3n/2 \rfloor \).

**Proof**: If \( S > \lfloor 3n/2 \rfloor \), we are done. Otherwise, let \( d = S - n \); note that \( d \leq \lfloor n/2 \rfloor \), so \( n - d \geq \lceil n/2 \rceil \). We will show that \( U > \lfloor (n-d)/2 \rfloor \). This will imply that \( S + U > (n+d) + \lfloor (n-d)/2 \rfloor = \lfloor 3n/2 \rfloor + \lfloor d/2 \rfloor \). By Theorem 1, \( S \geq n \), so \( d \geq 0 \) and \( S + U > \lfloor 3n/2 \rfloor \).
Suppose for a contradiction that \( U \leq [(n - d)/2] \). Consider some scan operation executing in isolation. This operation takes at most \( S = n + d \) time units. This means that there are at least \( n - d \) cells that are read only once. Consider the earliest and latest reads of such cells; suppose that the two cells are \( i \) and \( j \), respectively. The reads must occur at least \( n - d \) time units apart. That is, they must occur at times \( t_i \) and \( t_j \), where \( t_j \geq t_i + (n - d) - 1 \). Now, suppose that there is an \( update_i \) that begins at time \( t_i \). It must end by time \( t_i + (U - 1) \leq t_i + [(n - d)/2] - 1 \). Suppose that there is an \( update_j \) that begins at time \( t_i + [(n - d)/2] \). It must end by time \( t_i + [(n - d)/2] + (U - 1) \leq t_i + [(n - d)/2] + [(n - d)/2] - 1 \leq t_i + (n - d) - 1 \leq t_j \). Thus, the two updates are non-overlapping and their execution intervals are entirely contained within the part of the scan beginning with the read of cell \( i \) and ending with the read of cell \( j \). For the following, we assume that each update is of a value different from that of the preceding update to the same location.

Because the implementation is honest, the \( update_i \) must write its value into that cell during its execution interval. The earliest that this can happen is at time \( t_i \). The scan reads that cell only at that time. Because the algorithm cannot control the order in which the memory linearizes reads and writes that occur at the same time, it is possible that the read may be linearized before the write. Because this is the scan’s only read of cell \( i \), the scan will have to return an older (different) value for that cell. Similarly, the \( update_j \) writes to that cell no later than time \( t_j \), which is the only time at which the scan reads that cell. In this case, the write could be linearized first; since this is the scan’s only read of that cell, it must return the value written by the update.

From the above, it follows that the scan must be linearized before the \( update_i \) and after the \( update_j \). This means that the \( update_i \) must be linearized before the \( update_j \). But in the actual execution, these updates are non-overlapping and occur in the opposite order. This contradicts the existence of the implementation.\(^1\)

\( \square \)

The following section present two honest real-space implementations of atomic snapshot with \( S + U = 2n + c \), where \( c \) is a constant that is either 2 or 3.

4 Implementations

This section considers two honest implementations of atomic snapshots. Implementation 1 has \( U = S = n+1 \), while Implementation 2 has \( U = [n/2] + 1 \) and \( S = [3n/2] + 1 \). Thus, both have \( U + S = 2n + O(1) \). For the sake of simplicity, each implementation assumes that the updating and scanning processors form disjoint sets.

Both of these implementations are based on dividing time into slots. Each time unit is either a write-slot or a read-slot. Only write operations can occur in a write-slot and only read operations can occur in a read-slot. The sloting of time is done a priori and, because processors are synchronized, the slot type of a given time unit is common

\(^1\)For the sake of readability, certain technical details are omitted from the above proof. Specifically, one would have to argue that a scan that reads a cell only once would have to return the value read from that cell and that the existence of the updates concurrent with the scan would not change the scan’s pattern of reading. To show that there must be some execution in which these facts are true is tedious but relatively straightforward.
function scan
   slot := now mod (n + 1)
   if slot = 0 then
      wait one time unit
      slot := 1
   end if
   for j := slot to n do
      read array[j] into local[j]
   end for
   if slot > 1 then
      wait one time unit
      for j := 1 to slot - 1 do
         read array[j] into local[j]
      end for
   end if
   return(local)
end {scan}

Figure 1: Implementation 1 of Atomic Snapshot

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>⋮</th>
<th>n - 1</th>
<th>n</th>
<th>n + 1</th>
<th>n + 2</th>
<th>n + 3</th>
<th>⋮</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>R</td>
<td></td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Time Slotting for Implementation 1

knowledge. Note that this slotting of time is not an additional system assumption; it is simply a technique used by the algorithms and depends only on the system’s global clock.

4.1 Implementation 1: Fast Scans

This section presents an implementation of atomic snapshots with short scan operations; a scan never takes more than one time unit more than the amount of time needed to read the entire array. On the other hand, updates (which perform only one write operation) can take as long as the scans. The implementations of the update and scan operations are given in Figure 1. Note that the implementation uses an array of memory cells called array.

For this implementation, time is slotted in the following way: of every \( n + 1 \) time units, one is a write-slot and the remaining \( n \) are read-slots. Specifically, every time unit whose number is a multiple of \( n + 1 \) is a write-slot. This is illustrated in Figure 2.

An update operation waits until the next write-slot, at which time it writes its parameter to the appropriate cell and exits. A scan operation waits until the next
read-slot; this may require waiting one time unit. It then reads the array, one cell at a
time, beginning with the cell indexed by the current time modulo \( n + 1 \) (since this is a
read-slot, this value is between 1 and \( n \)). If all cells have been read after reading cell \( n \),
the scan returns the array of values read. Otherwise, the scan waits one time unit (this
is a write-slot) and then reads the remainder of the array, beginning with cell 1. After
this, it returns the array of values read.

We can now prove the complexity and correctness of this implementation.

**Lemma 3:** For Implementation 1, \( U = n + 1 \).

*Proof:* An update always waits until the next write-slot (if it does not start at one).
Since there is a write-slot in every \( n + 1 \) consecutive time units, this means that an updat-
ing processor can always write by the \( n + 1 \)st time unit once it begins. Since the update
concludes after the write, \( U \leq n + 1 \). If an update begins at time \( t = 1 \mod (n + 1) \),
then the update does not conclude until time \( t + n \). Thus, \( U = n + 1 \). \( \square \)

**Lemma 4:** For Implementation 1, \( S = n + 1 \).

*Proof:* Suppose that a scan begins at time \( t \). If \( t = 0 \mod (n + 1) \), then the scan
waits one time unit and then reads the entire array in time units \( t + 1 \) to \( t + n \); the
entire scan thus takes \( n + 1 \) time units. If \( t = 1 \mod (n + 1) \), then the scan reads
the entire array in time units \( t \) to \( t + (n - 1) \); the entire scan takes \( n \) time units. If
\( t = r \mod (n + 1) \) (\( r \notin \{0, 1\} \)), then the scan reads part of the array from time \( t \) to
\( t + (n - r) \), waits at time \( t + (n - r) + 1 \) (this is a write-slot), and reads the remainder
of the array from time \( t + (n - r) + 2 \) to \( t + (n - r) + r = t + n \); the entire scan takes
\( n + 1 \) time units. Thus, \( S = n + 1 \). \( \square \)

**Theorem 5:** Implementation 1 correctly implements atomic snapshots.

*Proof:* Consider some execution of Implementation 1. The proof requires the exis-
tence of a linearization of all scan and update operations such that every scan returns
for each cell the value of the most recent update to that cell. This linearization must
respect the order of non-overlapping operations. To do so, we give a linearization time
for each operation and specify an ordering of operations with the same linearization
time. An update operation to cell \( i \) whose write takes place at time \( t_w \) has linearization
time \( t_w + t \). A scan operation whose last read takes place at time \( t_l \) has linearization
time \( t_l \). For a given linearization time, order all updates before all scans; the relative or-
der of scans at a given time is unimportant.\(^2\) This linearization must be consistent with
the values returned by the scans and preserve the order of non-overlapping operations.

We now show that each scan returns, for cell \( i \), the value of the most recent update
to cell \( i \) that precedes the scan with respect to the linearization order. Consider a scan
whose last read occurs at time \( t_l = m \mod (n + 1) \). We show that value returned by
the scan for cell \( i \) is written by an update that is serialized before the scan. Consider
now two cases:

\(^2\)It is not hard to see that there can be only one update linearized at a given time. This is because
there is only one updating processor for each cell.
• $i \leq m$. The value returned was written no later than the latest write-slot before
time $t_\ell$, which was at time $t_\ell - m$. The update of such a value has a linearization
time of at most $(t_\ell - m) + i = t_\ell - (m - i) \leq t_\ell$. This implies that the update is
linearized before the scan.

• $i > m$. This implies $m < n$, so there is a write-slot in the midst of the scan (at
time $t_i - m$). Cell $i$ is read before this write-slot, so the value is thus written no
later than the preceding write-slot, at time $t_\ell - m - (n + 1)$. The corresponding
update is thus linearized no later than time $(t_\ell - m - (n + 1)) + i \leq t_\ell - (n + 1) +
(i - m) \leq t_\ell - (n + 1) + (n - 1) = t_\ell - 2$, which is before the scan.

It is easy to see that the value returned is that of the most recent update that is
linearized before the scan.

Finally, we need to show that the linearization order is consistent with that of
non-overlapping operations. This would follow immediately if the linearization time of
each operation were during its execution interval; unfortunately, this is not the case
for updates. Nevertheless, we can prove that the desired ordering holds; consider four
cases for nonoverlapping operations:

• Two updates. If two updates are non-overlapping, they write in different write-
slots. Suppose that these are at time $t_1$ and $t_2$, respectively, where $t_1 < t_2$.
Clearly, $t_2 \geq t_1 + (n + 1)$. The linearization time of the first update is no later
than time $t_1 + n$, while the linearization time of the second is no earlier than
$t_2 + 1 \geq (t_1 + (n + 1)) + 1 = t_1 + n + 2 > t_1 + n$. Thus, the updates are linearized
in the correct order.

• An update preceding a scan. Suppose that the update writes at time $t_w$; its
linearization time is no later than $t_w + n$. For the scan to follow the update, it
can start no earlier than time $t_w + 1$. Thus, its linearization time is no earlier
than $t_w + n$. This implies that the update is ordered before the scan because,
when updates and scans have the same linearization time, updates are ordered
first.

• A scan preceding an update. Suppose that the linearization time of the scan $t_\ell$.
If an update begins after $t_\ell$, then its linearization time is at least $t_\ell + 2$, and it is
thus ordered after than scan.

• Two scans. If two scans do not overlap, then the last read of the first is before the
last read of the second. Since the times of these reads are the scans' linearization
times, the two scans are linearized in the correct order.

We conclude that the implementation is correct.
procedure update\(_i\)(x)

  wait until now mod (n + 2) \in \{0, [n/2] + 1\}

  write x into array[i]

end \{ update\(_i\) \}

Figure 3: The Procedure update\(_i\) for Implementation 2 of Atomic Snapshot

### 4.2 Implementation 2: Faster Updates

This section presents an implementation of atomic snapshots for which update operations are half as long as Implementation 1; however, the scan operations are longer by a similar amount. Thus, this implementation is to be preferred if more updates than scans are expected. The implementations of the update and scan operations are given in Figures 3 and 4.

For this implementation, time is slotted in the following way: of every \(n + 2\) time units, two are write-slots and the remaining \(n\) are read-slots. Two write-slots are times congruent to 0 or to \([n/2] + 1\) modulo \(n + 2\). This is illustrated in Figure 5.

As in Implementation 1, an update operation waits until the next write-slot, at which time it writes its value to the appropriate cell and exits. A scan operation might not be able to start immediately, even if it begins in a read-slot. It must wait until the next read-slot that immediately follows a write-slot (thus, it is possible that it may read immediately if it begins in such a read-slot). At this point, there are enough consecutive read-slots to read about half the array. The half chosen depends on the time of the first such read-slot. If the time is congruent to 1 modulo \(n + 2\), then the scanning processor reads the first \([n/2]\) elements of the array, waits one time unit (this is a write-slot), and reads the remaining \([n/2]\) elements of the array. Otherwise, the second part of the array is read first, there is a pause (during a write-slot), and the first half of the array is read. In both cases, the write-slot that separates the two periods of reading is called the intervening write-slot.

We can now prove the complexity and correctness of this implementation.

**Lemma 6:** For Implementation 2, \(U = [n/2] + 1\).

**Proof:** An update always waits until the next write-slot (if it does not start at one). Consider an update that begins at time \(t\), and let \(m = t \mod (n + 2)\). If \(m \in \{0,[n/2] + 1\}\), then the update immediately writes and exits, taking 1 time unit. If \(0 < m \leq [n/2]\), it waits through 1 + [n/2] - \(m\) read-slots before writing. If \([n/2] + 1 < m \leq n + 1\), then it waits through \([n/2] - (m - ([n/2] + 2)) = [n/2] + [n/2] + 2 - m = (n - m) + 2\) read-slots. In the first case, it waits for at most \(1 + [n/2] - 1 = [n/2]\) time units. In the second, it waits for at most...
function scan

wait until now mod (n + 2) \in \{1, \left\lfloor \frac{n}{2} \right\rfloor + 2\} do
if now mod (n + 2) = 1 then
    for j := 1 to \left\lfloor \frac{n}{2} \right\rfloor do
        read array[j] into local[j]
    wait one time unit
    for j := \left\lfloor \frac{n}{2} \right\rfloor + 1 to n do
        read array[j] into local[j]
else
    for j := \left\lfloor \frac{n}{2} \right\rfloor + 1 to n do
        read array[j] into local[j]
    wait one time unit
    for j := 1 to \left\lfloor \frac{n}{2} \right\rfloor do
        read array[j] into local[j]
return(local)

end \{scan\}

Figure 4: The Procedure scan for Implementation 2 of Atomic Snapshot

<table>
<thead>
<tr>
<th>1</th>
<th>\cdots</th>
<th>\left\lfloor \frac{n}{2} \right\rfloor</th>
<th>\left\lfloor \frac{n}{2} \right\rfloor + 1</th>
<th>\left\lfloor \frac{n}{2} \right\rfloor + 2</th>
<th>\cdots</th>
<th>n + 1</th>
<th>n + 2</th>
<th>n + 3</th>
<th>\cdots</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>\cdots</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>\cdots</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

Figure 5: Time Slotting for Implementation 2
\( (n - (\lceil n/2 \rceil + 2)) + 2 = \lceil n/2 \rceil \). Thus, \( U \leq \lceil n/2 \rceil + 1 \) (one time unit is needed for the write). If \( m = 1 \), the update actually takes this long, so \( U = \lceil n/2 \rceil + 1 \). \( \square \)

**Lemma 7:** For Implementation 2, \( S = \lceil 3n/2 \rceil + 1 \).

**Proof:** Suppose that a scan begins at time \( t \). Let \( r = t \mod (n+2) \) and consider two cases:

- \( r \in \{0, 1\} \) or \( r > \lceil n/2 \rceil + 2 \). If \( r = 0 \), the processor waits through one write-slot. If \( r > \lceil n/2 \rceil + 2 \), the processor waits through \( n+2-r \) read-slots and one write-slot. The processor then reads for \( \lceil n/2 \rceil \) read-slots (reading the first half of the array), waits through one write-slot, and reads for \( \lceil n/2 \rceil \) read-slots. In the worst case \( r = \lceil n/2 \rceil + 3 \), the entire scan takes \( n+2-(\lceil n/2 \rceil + 3) + 1 + \lceil n/2 \rceil + 1 + \lceil n/2 \rceil = \lceil 3n/2 \rceil + 1 \) time units.

- \( 1 < r \leq \lceil n/2 \rceil + 2 \). If \( r < \lceil n/2 \rceil + 2 \), then processor first waits through \( \lceil n/2 \rceil - (r - 1) \) read-slots and one write-slot. It then reads for \( \lceil n/2 \rceil \) read-slots (reading the second half of the array), waits through one write-slot, and reads for \( \lceil n/2 \rceil \) read-slots. In the worst case \( r = 2 \), the entire scan takes \( (\lceil n/2 \rceil - 1) + 1 + \lceil n/2 \rceil + 1 + \lceil n/2 \rceil = \lceil 3n/2 \rceil + 1 \) time units.

From the above, we conclude that \( S \leq \lceil 3n/2 \rceil + 1 \). If the scan begins at time \( t = 2 \mod (n+2) \), the scan actually takes that long, so \( S = \lceil 3n/2 \rceil \). \( \square \)

**Theorem 8:** Implementation 2 correctly implements atomic snapshots.

**Proof:** The proof is similar to that of Lemma 5 in that we give a linearization time for each operation.

The linearization time of every operation is a write-slot. For updates, it is the write-slot in which the value is actually written. For scans, it is the intervening write-slot. To further order the operations, consider each write-slot to be divided into three sub-slots: “first,” “middle,” and “last.” Scan operations are linearized in the middle sub-slot. To linearize the updates, we must consider the time \( t_w \) of the write-slot. If \( t_w = 0 \mod (n+2) \), then updates to cells 1 to \( \lceil n/2 \rceil \) are linearized in the first sub-slot, while updates to cells \( \lceil n/2 \rceil + 1 \) to \( n \) are linearized in the last sub-slot. If \( t_w = \lceil n/2 \rceil + 1 \mod (n+2) \), then updates to cells 1 to \( \lceil n/2 \rceil \) are linearized in the last sub-slot, while updates to cells \( \lceil n/2 \rceil + 1 \) to \( n \) are linearized in the first sub-slot. The relative order of operations within the same sub-slot is unimportant. We must now show that this linearization is consistent with the values returned by the scans and that it preserves the order of non-overlapping operations.

We first show that each scan returns, for cell \( i \), the value of the most recent update to cell \( i \) that precedes the scan with respect to the linearization order. Consider a scan whose intervening write-slot is time \( t_w \). We show that value returned by the scan for cell \( i \) is written by an update that is serialized before the scan. Consider now two cases:
• \( t_s = 0 \mod (n + 2) \). The intervening write-slot occurs after the scan has read the second half of the array and before it reads the first half. Consider now the value returned for cell \( i \). There are two cases:

- \( i \leq \lfloor n/2 \rfloor \). Cell \( i \) is in the first half of the array and is thus read after the intervening write-slot. The value returned is thus written at or before \( t_s \). An update writing before \( t_s \) is certainly linearized before the scan; an update writing at \( t_s \) is linearized in the first sub-slot of \( t_s \) (because \( i \leq \lfloor n/2 \rfloor \)) and is before the scan, which is in the middle.

- \( i > \lfloor n/2 \rfloor \). Cell \( i \) is in the second half of the array and is thus read before the intervening write-slot. The value returned is thus written before \( t_s \) and its update is thus linearized before the scan. We also observe an update writing a later value must be linearized after the scan. Such an update writes no earlier than \( t_s \) and, because \( i > \lfloor n/2 \rfloor \), it will be linearized in the last sub-slot of \( t_s \) and is after the scan, which is in the middle.

• \( t_s = \lceil n/2 \rceil + 1 \mod (n + 2) \). This case is similar to the first.

It is easy to see that the value returned is also that of the most recent update that is linearized before the scan.

Finally, we need to show that the linearization order is consistent with that of non-overlapping operations. This follows immediately from the fact that the linearization time of every operation occurs during its execution interval; the linearization times of nonoverlapping operations are thus ordered to match the operations’ relative execution order.

We conclude that the implementation is correct. \( \square \)

### 4.3 Comments

Both of the above implementations were written with the simplifying assumption that the updating and scanning processors were disjoint. If this is not the case, then a constant additive factor in running time may be saved by having a scanning processor not read its own cell.

These implementations were proven correct for the single-writer case; that is, it was assumed that only processor \( p_i \) may invoke \( \text{update}_i \). Because of this, it was never necessary to order the updates linearized at the same time unit (or sub-slot). The implementations remain correct if this assumption is removed. The linearization of concurrent updates must then be more carefully specified. A correct linearization still exists because the underlying memory is assumed itself to be linearizable.

We developed Implementation 2 from Implementation 1 by attempting to shorten update time. This was done by dividing the scanning period in half by inserting an intervening write-slot. This cut the update time roughly in half, as write-slots appeared twice as frequently. To be correct, it was necessary to “synchronize” scans by having
them all begin reading immediately after a write-slot. For this reason, the scan time went up by roughly the length of the new update time.

One might ask, why not reduce update times even further by adding more write-slots? Suppose that there were write-slots in every \( n/4 \) time units. This could potentially reduce the update time to approximately \( n/4 \) and scan times to \( 5n/4 \). It turns out that this strategy fails. The correctness of Implementation 2 above depends critically on the fact that there is only one intervening write-slot during the reading period of a scan. If more than one such slot is permitted (which would be necessary if write-slots were more frequent), then different scans might not consistently linearize the updates occurring in different intervening slots.

5 Conclusions

This paper has explored the implementation of atomic snapshots in synchronous distributed systems with shared read/write memory. Because of the synchrony in such systems, these implementations can be much simpler and more efficient than those for asynchronous systems. If algorithms may use large amounts of memory, then extremely fast and trivially simple implementations are possible. For example, suppose that memory consisted of \( n \) cells of unbounded size. With each update, processor \( p_i \) could simply write to cell \( i \) the values of all updates it executed in the last \( n \) time units, each paired with its invocation time (since there can be at most \( n \) such updates, the memory is unbounded only because the size to represent the invocation times is unbounded). A scan can simply read the \( n \) cells and, for each cell, return the most recent value that was written before the scan was invoked. This implementation is trivially correct; updates take only one time unit and scans take only time \( n \).

Our work has chosen instead to consider implementations with very limited memory. In particular, the shared memory allowed is only large enough to store the array being simulated and it is required that memory honestly reflect the simulated array. These restrictions forbid implementations such as the one described above.

Under such restrictions, we showed that the worst-case scan time \( S \) must be at least \( n \), where \( n \) is the number of cells in the array. This is not surprising, as a scan operation could conceivably have to read all \( n \) cells of the array. We then show that, if \( U \) is the worst-case update time, \( S + U > \lfloor 3n/2 \rfloor \). This means that, if fast scans are used (i.e., those with \( S = n \)), then some updates must take time greater than \( n/2 \). This is disappointing, as the only real “work” that an update requires is a single write operation. Nevertheless, it is not surprising; most implementations in asynchronous systems require an update operation to perform an entire scan, resulting in similar execution times for the two operations.

Section 4 gave two implementations. This first has scan operations that run almost as fast as the proven lower bound; unfortunately, its updates may require \( n + 1 \) time units. The second implementation reduces the worst-case update time to \( \lceil n/2 \rceil + 1 \), but incurs a corresponding increase in scan time. Both implementations have \( S + U = 2n + c \), where \( c \) is a small constant. Thus, there is a gap of about \( n/2 \) time units between the proven lower bound and the two implementations. Closing this gap is a subject for
future research.

Acknowledgements

This work grew out of discussions with Kiran Achyutuni, Rida A. Bazzi, Richard Clayton, and Mehdi Sayfi. We are especially grateful to Rida for his continued interest and his critical assessment of the work as it developed.

References


