CHAPTER

SIXTEEN

BOUNDED TIMESTAMPS

In Chapter 10, we saw two simulations, one of a multi-reader register from single-reader registers, and another of a multi-writer register from single-writer registers, which relied on the use of unbounded timestamps. The simulations used the unbounded timestamps to provide information about the ordering of write operations. In the first case, sequence numbers were generated by a single processor (the unique writer) to order its writes, and in the second case, vectors of sequence numbers were generated by several processors (the writers) to order non-overlapping write operations.

A closer look at these algorithms reveals that in both cases we can speak of *timestamps*. In the first case, timestamps are generated by a single processor, in a sequential manner, i.e., one after the other with no overlap, while in the second case timestamps are generated *concurrently*, by several processors.

This chapter shows how to provide (most of) the semantics of unbounded timestamps using only *bounded* timestamps. We first consider timestamps generated by a single processor and then consider concurrent timestamps, which can be generated by several processors. In each case, we show how the bounded timestamps, single-generator or concurrent, can be applied to bound the memory requirements of the simulations of multi-reader registers from single-reader registers (Algorithm 10.2) and multi-writer registers from single-writer registers (Algorithm 10.3), respectively.

16.1 Single-Generator Timestamp System

In order to motivate the requirements of a single-generator timestamp system, recall the way timestamps were used in the simulation of multi-reader registers from single-reader registers (Algorithm 10.2). In this simulation, the writer had

a way to generate a new timestamp (simply, add one to the previous timestamp); and each reader had a way to compare some set of timestamps and identify the one that was generated most recently (simply, the maximal one).

We conclude that a single-generator timestamp algorithm must support the following two procedures:

NewTS: takes no parameters and returns a timestamp.

CompTS: takes two timestamps as parameters and returns one of them.

Informally speaking, CompTS should return the parameter that was produced more recently by NewTS. More precisely, assuming only a single processor, the *generator*, invokes NewTS:

Timestamp Property: Consider any CompTS execution *ct* on arguments T_0 and T_1 that returns T_i. Let *nt* be the NewTS execution that returns T_i and ends most recently before *ct* begins. Then T_{1-i} was not returned by a NewTS that precedes *ct* according to the real-time order. (Since NewTS operations are invoked by a single processor, they are totally ordered.)

Using CompTS it is possible to provide a procedure MaxTS, which takes any number of parameters and returns the 'latest' among them. In more detail, consider a MaxTS execution *mt* that returns a parameter T: Let *nt* be the NewTS execution that returns T and ends most recently before *mt* begins. Then the parameters for *mt* do not include a value returned by a NewTS that begins after *nt* ends and ends before *mt* begins.

Below we concentrate on CompTS and assume MaxTS is derived from it in the obvious manner.

The algorithm we present is based on the following observation: For most algorithms, at any point in the execution, only a bounded number of timestamps are *active* in the system, i.e., contained in the local and shared objects; these are the only timestamps that can be compared by processors (to determine the relative order in which the values associated with them were written, for example). Therefore, although the number of timestamps generated so far in the execution is potentially unbounded, only a bounded number of them are relevant after a particular prefix of the execution. In our example, the simulation of multi-reader registers with single-reader registers, the relevant timestamps are those included in the values written by the writer for the readers and the values written by the readers for each other.

Assume that the generator has some set $\mathcal{A}$ which is a superset of the set of the active timestamps. Then the generator can produce a new timestamp by taking some value T not in $\mathcal{A}$, and indicating in the shared memory that T is to be considered larger than all values in $\mathcal{A}$. If we guarantee that in any configuration, $|\mathcal{A}| < M$, for some positive integer M, then a set $\mathcal{T}$ of M integers suffices to represent all timestamps.

The key for this algorithm is the ability to trace the active timestamps existing in the system, or at least a bounded superset that includes them. Although in most cases, an outside observer of the system can identify the exact set of active timestamps, it is not easy to trace them inside the system. A close study of algorithms using timestamps indicates that this problem stems from a key limitation of shared memory algorithms: When a processor writes a value to a register, it does not know which processors, if any, read the value.

We next describe a simulation of a single-reader single-writer register which allows values to be *traced*, thus avoiding this problem. Afterwards, we elaborate on the idea described above, and present the algorithm for bounded timestamps with a single generator, followed by its application to the simulation of multi-reader registers from single-reader registers.

16.1.1 Traceable Writing and Reading of a Single Register

Consider a register R, written by processor p_w and read by processor p_r. For clarity of presentation, we concentrate on a single register, and sometimes omit its name. We would like to provide a way for some observing processor in the system to determine a reasonably sized superset of the values that p_r has actually obtained from R. To achieve this goal, we will provide a *traceable* version of the register R, implemented by several shared registers and special procedures called traceable write (twrite) and traceable read (tread) to access the traceable register.

Processors p_w and p_r notify each other of their intention to access R by using the handshaking procedures of Algorithm 10.4. When p_r wishes to read the current value of R, it signals this by trying to handshake, i.e., calling tryHS_r. After p_w writes to R, it signals this by trying to handshake, i.e., calling tryHS_w.

Using the $\mathsf{checkHS}_r$ procedure, the reader p_r can notice if p_w is currently writing to R. If p_w is not writing now (i.e., $\mathsf{checkHS}_r$ returns false), then p_r can take the current value of R; furthermore, p_w will notice that p_r has read the current value of R, since p_r tried to handshake. If p_w is currently writing (i.e., $\mathsf{checkHS}_r$ returns true), then it is possible that p_w will complete without being able to notice which value p_r has read. But in this case, p_w explicitly 'sets aside' a value for p_r during its next write.

In addition to the two shared bits needed for the handshaking procedures, the algorithm uses the following shared data structures:

***Value*:** The value of the traceable register R; written by p_w and read by p_r.

***Copy*:** The value set aside by p_w for p_r; written by p_w and read by p_r.

***Viable*:** Holds a pair of values written to the traceable register, which are suspected to have been read by p_r; written by p_w and read by the observing processor.

***Prev*:** The previous value written to the traceable register; written by p_w and read by the observing processor.

Algorithm 16.1 Traceable read and write procedures for a single traceable register: code for treader p_r and twriter p_w.

Initially *Value* = *Copy* = *Prev* = v_0 and *Viable* = $\langle v_0, v_0 \rangle$,
where v_0 is the initial value of the traceable register

```
procedure tread_r():
1:  tryHS_r()
2:  v := Value
3:  if checkHS_r() then v := Copy
4:  return v

procedure twrite_w(v):
1:  Value := v
2:  if checkHS_w() then
3:      Copy := v
4:      Viable := ⟨Prev, v⟩
5:      tryHS_w()
6:  Prev := v                                   // outside the if
```

The pseudocode for tread and twrite appears as Algorithm 16.1. In the algorithm, we use the convention that the writer of a single-reader single-writer register can also read it. Superficially, this implies that there are two readers of the register; however, the writer can just 'remember' the previous value it has written to the register, using a local variable. This convention cuts down on the clutter in the pseudocode.

Viable contains the value twritten by *tw*, the most recent twrite whose checkHS returned true. It also contains the value twritten by *tw'*, the twrite immediately preceding *tw*; the checkHS of *tw'* might or might not have returned true. As we will show, since the checkHS of *tw* returned true, some tread happened after the beginning of *tw* and before the end of *tw'*.

Prev keeps track of the previous value twritten; this value will end up being written to *Viable* in the next twrite if its checkHS returns true.

The traceable read and write procedures guarantee a useful property. Values that are potentially in use are stored in shared variables, either *Prev* or *Viable*. Note that only p_w writes to *Prev* and *Viable*, and that p_r does not read from it. Therefore, some other observing processor can learn which values are active by reading *Prev* and *Viable*.

Consider any execution α that is correct for the read/write register communication system, in which p_w makes calls to twrite, p_r makes calls to tread, and no other procedures write to the variables used by these two procedures.

Before stating the traceable use property in more detail, we need to define

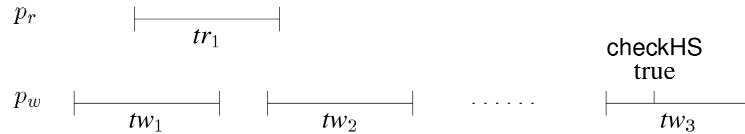

Figure 16.1: Basic illustration for Lemma 16.1: tr_1 reads from tw_1.

a tread reading from a twrite. Let *tr* be an execution of tread in α. If *tr* returns the value it reads from *Value*, then *tr* is defined to *read from* twrite execution *tw*, where *tw* contains the latest write to *Value* that is linearized before *tr*'s read of *Value*. If *tr* returns the value it reads from *Copy*, then *tr* is defined to *read from* twrite execution *tw*, where *tw* contains the latest write to *Copy* that is linearized before *tr*'s read of *Value*. If there is no write to *Value* (or *Copy*) that is linearized before *tr*'s read from *Value* (or *Copy*), meaning that *tr* returns the initial value of the traceable register, then *tw* is defined to be an imaginary 'initializing' twrite that writes the initial value and ends just before α begins.

In the proof, we talk about the relative order of occurrence of reads, writes, and handshaking procedures. It is perhaps not immediately clear that they can be totally ordered, since they do not happen atomically. Recall from Chapter 10 that for each handshaking procedure, we can identify a read or write operation inside it that corresponds to when it 'really happens'. For a tryHS procedure, this operation is the write of its own handshaking bit. For a checkHS procedure, this operation is the read of the other handshaking bit. Since α is correct for the read/write register communication system, all the reads and writes can be linearized. We use the linearization order of these operations to linearize the handshaking procedures.

Viable and *Prev* contain three (possibly different) values twritten by p_w; the key property we wish to prove is that these values include the most recent value tread by p_r. This property is precisely stated and proved later, in Lemma 16.2.

But first, we prove that if the most recent value tread by p_r is put in *Viable* either by the twrite it reads from or by the next twrite, then it stays there until p_r treads a later value.

Lemma 16.1 *Let tr_1 be a* tread *by p_r in α that returns v and let tw_1 be the* twrite *by p_w that tr_1 reads from. Let tr_2 be the first* tread *by p_r after tr_1 that does not read from tw_1. Let tw_2 be the next* twrite *by p_w following tw_1. Suppose v is put in Viable by either tw_1 or tw_2 prior to the beginning of tr_2. Then v remains in Viable until the beginning of tr_2.*

Proof. The value v stays in *Viable* until a later twrite, call it tw_3, flushes it out. Since tw_3 writes to *Viable*, its checkHS_w returns true (see Figure 16.1). There are two cases:

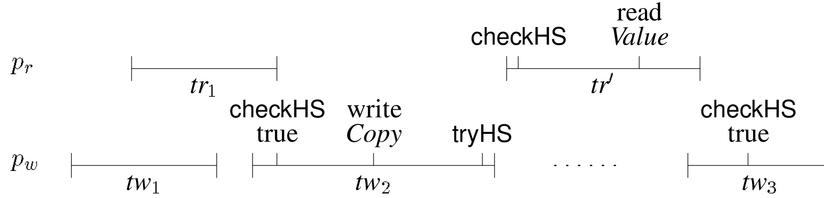

Figure 16.2: Illustration for Lemma 16.1, Case 1: tw_2 writes to *Viable*.

Case 1: The checkHS$_w$ in tw_2 returns true; therefore, tw_2 includes a tryHS$_w$.

First, we show that the checkHS$_w$ in tw_2 follows the tryHS$_r$ in tr_1, implying that the tryHS$_w$ in tw_2 follows the tryHS$_r$ in tr_1. Suppose in contradiction that the checkHS$_w$ in tw_2 precedes the tryHS$_r$ in tr_1. If tr_1 returns from *Value*, then since tw_2's write to *Value* is between tw_1's write to *Value* and tr_1's read of *Value*, it follows that tr_1 does not read from tw_1, a contradiction. If tr_1 returns from *Copy*, implying that its checkHS$_r$ returns true, then tw_2's write to *Copy*, and hence its tryHS$_r$, follows tr_1's read of *Copy*, since tr_1 does not read from tw_2. But then there is no tryHS$_w$ by p_w between the tryHS$_r$ and checkHS$_r$ of tr_1, and thus the checkHS$_r$ of tr_1 returns false, a contradiction.

Since tw_3's checkHS$_w$ returns true, there exists another tread, call it tr', by p_r whose tryHS$_r$ is between the tryHS$_w$ of tw_2 and the checkHS$_w$ of tw_3. (See Figure 16.2.) Since the read of *Value* and, if it occurs, the read of *Copy* in tr' follow the writes to *Value* and *Copy* in tw_2, tr' reads from tw_2 or a later twrite. Thus by the time v is removed from *Viable*, there has been a tread that does not read from tw_1.

Case 2: The checkHS$_w$ in tw_2 returns false; thus tw_1 must write v to *Viable*, and it must be that the checkHS$_w$ in tw_1 returns true.

Suppose tw_1 is the imaginary initializing twrite. If tr_1 returns from *Value*, then tw_2's write to *Value* must come after tr_1's read of *Value*, and so tw_2's checkHS$_w$ follows tr_1's tryHS$_r$. Since tw_2 is the first write by p_w, its checkHS$_w$ returns true, a contradiction. If tr_1 returns from *Copy*, then there must be a tryHS$_w$ by p_w between tr_1's tryHS$_r$ and checkHS$_r$. But then p_w writes to *Copy* before tr_1 reads from *Copy*, contradicting tr_1's reading from the initial value. Thus tw_1 cannot be the imaginary initializing twrite.

Since checkHS$_w$ in tw_2 is false, the tryHS$_w$ in tw_1 must follow the tryHS$_r$ of tr_1. Note that tw_3 is the first twrite after tw_1 whose checkHS$_w$ returns true. We want to show that another tread, tr', following tr_1 has already begun by the time of tw_3's checkHS$_w$, and that tr' does not read from tw_1.

Since tw_2's checkHS$_w$ returns false, Handshaking Property 2 implies that the read of tr_1's tryHS$_r$ precedes (the write of) tw_1's tryHS$_w$.

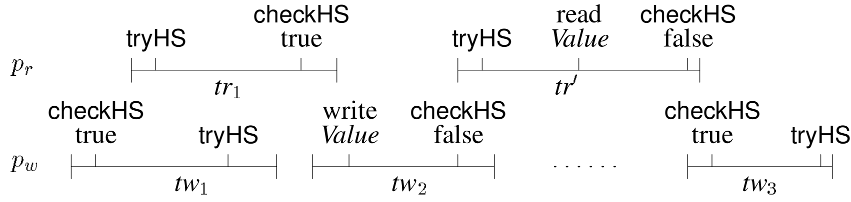

Figure 16.3: Illustration for Lemma 16.1, Case 2: tw_2 does not write to *Viable*.

We now show that the write of tr_1's tryHS_r cannot cause any subsequent $\mathsf{checkHS}_w$ by p_w to return true. Suppose in contradiction the write of tr_1's tryHS_r causes a subsequent $\mathsf{checkHS}_w$ by p_w to return true. Then it must occur between tw_2's $\mathsf{checkHS}_w$, which returns false, and tw_3's $\mathsf{checkHS}_w$, which returns true. Since this write follows tw_2's $\mathsf{checkHS}_w$, tr_1 cannot return from *Value*, since otherwise it would read from tw_2 or later, instead of tw_1. Since tr_1 must return from *Copy*, its $\mathsf{checkHS}_r$ must return true. Handshaking Property 1 implies that there exists a tryHS_w by p_w between the tryHS_w of tw_1 and the $\mathsf{checkHS}_r$ of tr_1. But this is a cotnradiction, since tw_3 contains the first tryHS_w after tw_1.

Since tw_3's $\mathsf{checkHS}_w$ returns true, Handshaking Property 1 implies that (the write of) some tryHS_r, call it tr', by p_r occurs between (the write of) the tryHS_w in tw_1 and the $\mathsf{checkHS}_w$ in tw_3. We have just argued that tr' cannot be tr_1. Furthermore, let tr' be the tread whose tryHS_r precedes the $\mathsf{checkHS}_w$ of tw_3 and follows the tryHS_w of every twrite that precedes tw_3. Note that the tryHS_r of tr' must follow the $\mathsf{checkHS}_w$ of tw_2, or else tw_2's $\mathsf{checkHS}_w$ would return true, since the tryHS_r of tr' is assumed to follow the tryHS_w of tw_1. (See Figure 16.3.)

By assumption that no tryHS_w is between the tryHS_r of tr' and the $\mathsf{checkHS}_w$ of tw_3, the $\mathsf{checkHS}_r$ of tr' returns false and thus tr' returns *Value*. In other words, tr' reads from tw_2 or later, and thus, by the time v has been removed from *Viable* by tw_3, there has been a tread that does not read from tw_1. □

To prove that Algorithm 16.1 correctly traces the values being tread, it remains to show that if a value v is tread then it is inserted in *Viable*. The proof of the next theorem shows that either the twrite writing v puts it in *Viable* or the next twrite does so and in the meantime, v is in *Prev*.

Theorem 16.2 *Let tr_1 be a* tread *by p_r in α that returns a value v and let tw_1 be the* twrite *by p_w that tr_1 reads from. Let tr_2 be the first* tread *by p_r after tr_1 that does not read from tw_1. Then v is in either Prev or Viable in every configuration of α from the end of tw_1 to the beginning of tr_2.*

Proof. *Case 1:* Either tw_1 is the imaginary initializing twrite or the $\mathsf{checkHS}_w$ in

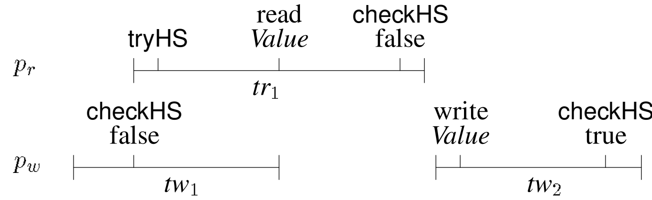

Figure 16.4: Illustration for Theorem 16.2: $\mathsf{checkHS}_w$ in tw_1 returns false.

tw_1 returns true. Therefore, tw_1 writes v to *Viable* and the theorem follows from Lemma 16.1.

Case 2: tw_1 is not the imaginary initializing twrite and the $\mathsf{checkHS}_w$ in tw_1 returns false. Thus tw_1 does not write to *Copy*; since tr_1 reads from tw_1, it returns from *Value*, indicating that the $\mathsf{checkHS}_r$ in tr_1 returns false.

As long as there is no subsequent twrite by p_w after tw_1, v remains in *Prev*. Let tw_2 be the next twrite by p_w. Since tr_1 reads from tw_1, and not tw_2, the write to *Value* in tw_2 follows the read of *Value* in tr_1. Thus the $\mathsf{checkHS}_w$ in tw_2 returns true, since if a twrite prior to tw_1 has its tryHS_w after tr_1's tryHS_r, tr_1's $\mathsf{checkHS}_r$ would not return false. (See Figure 16.4.) Thus tw_2 puts v in *Viable* before removing v from *Prev*, and the theorem follows from Lemma 16.1. □

So far, we have shown that values read by tread can be traced by any processor who reads *Prev* and *Viable*. Since we intend to use tread and twrite instead of ordinary read and write operations, we also need to prove they are linearizable.

Clearly, if *tr* returns v from *Value*, then v was written either by the most recent twrite operation that completed before *tr* starts or by a twrite operation that overlaps *tr*. For the case when *tr* returns v from *Copy*, we use the following simple lemma:

Lemma 16.3 *Let tr be a* tread *by* p_r *in* α *that returns a value* v *from Copy and let tw be the* twrite *by* p_w *that tr reads from. Then tw completes after tr starts.*

Proof. Suppose in contradiction that *tw* completes before *tr* starts. Then there must be a tryHS_w between *tr*'s tryHS_r and $\mathsf{checkHS}_r$. This tryHS_w is part of some twrite tw' that follows *tw*. But before doing the tryHS_w, tw' writes to *Copy* (see Figure 16.5), contradicting the fact that *tr* reads from *tw*, and not from tw'. □

Theorem 16.4 *The procedures* tread *and* twrite *in Algorithm 16.1 wait-free simulate a single-reader single-writer register.*

Proof. By Lemma 16.3, a tread operation reading from *Copy* returns a value written by an overlapping twrite operation. This shows that a tread operation reads

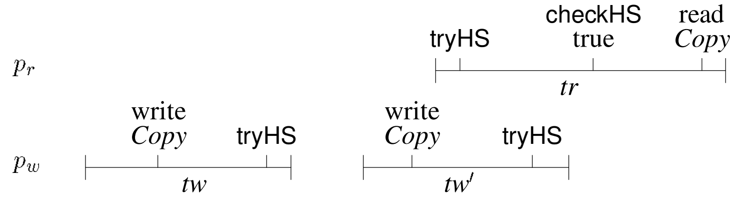

Figure 16.5: Illustration for Lemma 16.3.

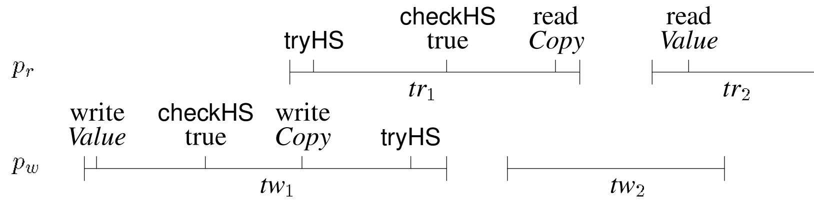

Figure 16.6: Illustration for Theorem 16.4: tr_1 returns from *Copy*.

from the most recent twrite operation that completed before it starts or from a twrite operation that overlaps it. To complete the proof of linearizability, we need to show there are no new-old inversions.

So, consider two tread operations, tr_1, which reads from twrite tw_1, and tr_2, which reads from twrite tw_2. If tr_1 completes before tr_2 begins, then either tw_1 and tw_2 are the same, or tw_1 precedes tw_2. If tr_1 and tr_2 return from the same variable (either *Value* or *Copy*), then the claim follows from the linearizability of the low-level variables.

If tr_1 returns from *Copy* then, by Lemma 16.3, tw_1 completes after tr_1 starts. Since tw_1 first writes to *Value*, then tr_2 returns from *Value* the value written by tw_1 or a later *twrite* operation from *Value*. (See Figure 16.6.)

If tr_2 returns from *Copy* then, by Lemma 16.3, tw_2 completes after tr_2 starts. The linearizability of the low-level variables implies that tr_1 cannot read from a twrite operation that starts after tr_1 completes. Therefore, tw_1 is either equal to tw_2 or to an earlier twrite. □

Clearly, tread and twrite are wait-free and require only a constant number of ordinary read and write operations. The procedures use a total of four single-writer single-reader registers, each containing either one or two ordinary values, (*Value*, *Prev*, *Copy*, and *Viable*) and two single-writer single-reader binary registers (the handshaking bits).

16.1.2 The Single-Generator Timestamp Algorithm

Let us now turn to the algorithm for the single-generator timestamp system. As discussed earlier, the key idea of the algorithm is fairly simple: The generator keeps track of its timestamps, using the traceable use reads and writes; when the generator needs to produce a new timestamp, it picks a new timestamp that is not currently active, and writes in a special *Order* register that the new timestamp is to be considered larger than all timestamps existing in the system.

However, a closer look at the way single-generator timestamps are used reveals that the above idea is not sufficient. Consider, for example, the simulation of a multi-reader register from single-reader registers (Algorithm 10.2). An important part of this algorithm is having a reader write to all readers the value it is going to return. It is possible that the reader will later read another value from the writer, and therefore, the variable *Viable* for the original (traceable) register will not contain this value. To avoid this situation, values are forwarded between readers using twrite and tread, instead of ordinary reads and writes. Before producing a new timestamp, the generator reads the *Prev* and *Viable* variables for all the traceable registers, which, as we prove below, contain the set of all timestamps that may be used in the system.

As usual, the processors are numbered p_0 through p_{n-1}; one of them is the generator and all of them can compare timestamps to find the maximum. Timestamps are integers from some finite set $\mathcal{T}$ and values are records that can contain other types, including timestamps.

We assume that an application obeys the following rules when using NewTS, CompTS, tread, and twrite:

Rule 1. Only one processor, the generator, calls NewTS.

Rule 2. For each ordered pair of distinct processors (p_i, p_j), p_i transmits timestamps to p_j only via a known finite set of traceable registers.

Rule 3. Whenever a processor writes a value containing a timestamp, it uses twrite and the value contains at most one timestamp.

Rule 4. Whenever a processor reads a value containing a timestamp, it uses tread.

Rule 1 implies that the NewTS invocations occur sequentially; this rule is true for some applications, such as the multi-reader simulation in Chapter 10. Rules 2 through 4 imply that all timestamps are transmitted from one processor to another via a traceable register. This requirement is necessary (but not sufficient) for the system to keep track of timestamps that are potentially in use.

In the timestamp system, then, we have a finite number of traceable registers through which processors communicate timestamps using twrite and tread. To disambiguate the traceable use procedures, the procedure calls should include the name of the traceable register as a parameter and the registers (*Value*, *Copy*, *Viable*,

Algorithm 16.2 A bounded timestamp system, for a single generator p_k.

```
procedure NewTS():                                              // for the generator p_k
1: read all Viable and Prev variables
2: let S be the set of all timestamps appearing in the Viable and Prev variables
3: let T be a timestamp not in S                   // remove timestamps not in S and
4: for i := 0 to n − 1 do Order[k, i] := (Order[k, i]|S).T           // append T
5: return T

procedure CompTS_i(k, T_1, T_2), 0 ≤ i ≤ n − 1:                // for each processor p_i
1: read Order[k, i]
2: return the parameter T_l that appears later in Order[k, i]
```

Prev, and the handshaking bits) used by the procedures should be parameterized with the name of the traceable register. However, to avoid clutter, these indications are generally not included.

The pseudocode appears as Algorithm 16.2. For compatibility with the multiple-generator system, presented later in this chapter, it is written for a given generator, p_k.

For a specific generator, p_k, the new procedures use some additional (non-traceable) shared registers, *Order*$[k, i]$ for all i. Conceptually we need only one *Order* variable, in which the generator records the order in which it has generated the timestamps that are potentially in use, as determined by the *Prev* and *Viable* variables. Since we only have at our disposal single-reader registers, we must use a separate copy of *Order* for each reader. Initially, each *Order*$[k, i]$ contains the timestamp included in the initial value of the traceable registers.[1]

We assume that the application uses the timestamps in a restricted manner; in addition to Rules 1 through 4 stated above, another rule is required to ensure that timestamps which are potentially in use are traced. First, we formally define the timestamps that are potentially in use. A timestamp T is *active for processor* p_i *via processor* p_j after some finite prefix of an execution if either:

- $p_j = p_i$ and T is the value returned by the most recent NewTS invocation by p_i (which implies that p_i is the generator), or
- $p_j \neq p_i$ and the most recent twrite by p_j to one of the registers tread by p_i (including the imaginary initializing twrite) or the most recent tread by p_i from one of the registers twritten by p_j contains T.

Thus we require:

Rule 5. If a processor calls CompTS or twrite, then every timestamp contained in a parameter must be active for that processor.

[1] For simplicity, we assume that the application has exactly one timestamp that appears in the initial value of every traceable register.

Together with Rules 2 and 3, this rule allows us to bound the number of timestamps potentially in use at any given time. If this rule were violated, then a processor could remember in its local state all timestamps ever generated so far, and reintroduce each one into the system later on.

We now define, inductively, the *forwarding distance* of a timestamp T that is active for a processor p_i via p_j. If p_j equals p_i, then T has forwarding distance 0. If p_j is not equal to p_i, then T has forwarding distance one greater than the forwarding distance of T for p_j when p_j performed the twrite from which p_i obtained T.

Notice that it is possible for the same timestamp to be active for p_i via several different processors, and with several different forwarding distances. For instance, this situation could occur if p_i obtains T directly from the generator, with forwarding distance 1, and if p_i also obtains T indirectly via p_j, with forwarding distance 2. In this case, to avoid ambiguity, we will consider T to have the smallest forwarding distance.

Next we define the NewTS invocation nt that *generates* timestamp T that is active for p_i via p_j, by induction on the forwarding distance:

1. If the forwarding distance of T for p_i is 0 (implying p_i is the generator), then nt is the most recent NewTS invocation.

2. Suppose T has forwarding distance $f > 0$ for p_i. Let *tw* be the twrite execution by p_j from which p_i obtains T. Then nt is the NewTS execution that generated T for p_j when p_j performed *tw*.

Fix an admissible execution α of the single-generator timestamp system and assume that the application follows Rules 1 through 5; we now prove that the Timestamp Property is satisfied. The proof relies on two lemmas. Lemma 16.6 shows that every active timestamp is in *Order*, and thus CompTS is well-defined. Lemma 16.7 shows that timestamps are listed in *Order* in the correct order. To prove these lemmas, we need a preliminary lemma stating that active timestamps with positive forwarding distance are in a *Prev* or *Viable* variable.

Lemma 16.5 *If T is active for p_i via p_j and has positive forwarding distance after some finite prefix of α, then T is in either a Prev or Viable variable in every configuration of the prefix after the completion of the first* twrite *by the generator following the* NewTS *that generated T for p_i.*

Proof. We will show this lemma by induction on f, the forwarding distance of T for p_i.

Basis: Suppose $f = 1$. Then the lemma follows from Rule 2 and Theorem 16.2.

Induction: Suppose $f > 1$. Let *tw* be the twrite of p_j which writes T for p_i using traceable register R. The forwarding distance of T for p_j during *tw* is $f - 1$.

Thus the inductive hypothesis states that T is in a *Prev* or *Viable* variable starting with the stated twrite of the generator and lasting as long as T is active for p_j, which, by Rule 5, is at least through the end of tw.

Since T is active for p_i via p_j at the end of the prefix under consideration, by Theorem 16.2, T is in either $Prev^R$ or $Viable^R$ from the end of *tw* to the end of the prefix. □

Lemma 16.6 *If T is active for p_i via p_j after some finite prefix of α, then T is in Order$[k, i]$ in every configuration of the prefix after the completion of the NewTS that generated T for p_i.*

Proof. We will show this lemma by induction on f, the forwarding distance of T for p_i.

Basis: If $f = 0$, i.e., $p_i = p_j$ is the generator, then by the code, T is put in *Order*$[i]$ during the NewTS that generated T. T stays in *Order*$[i]$ at least until the next twrite by p_i, at which time it ceases to be active for p_i via p_i.

Induction: Suppose $f > 0$. By Lemma 16.5, T is in either a *Prev* or *Viable* variable as long as it is active. Thus any execution of NewTS will observe T as being potentially in use and will neither remove it from *Order*$[i]$ nor choose it as the new timestamp. □

Lemma 16.7 *If T_1 and T_2 are active for p_i after some finite prefix of α and the NewTS that generated T_1 for p_i precedes the NewTS that generated T_2 for p_i, then T_1 precedes T_2 in Order$[i]$.*

Proof. Let nt_1 and nt_2 be the NewTS executions that generated T_1 and T_2, respectively, for p_i. Since nt_1 finishes before nt_2 begins and T_1 is still active after nt_2, a twrite by the generator containing T_1 follows nt_1 and precedes nt_2. By Lemma 16.5, T_1 remains in a *Prev* or *Viable* variable after the first twrite by the generator following nt_1.

During nt_2, the generator will observe T_1 in a *Prev* or *Viable* variable, and thus, will not remove it from *Order*$[i]$ or choose it. (This implies that $T_2 \neq T_1$.) Since the generator always adds new entries to the end of *Order*$[i]$, T_1 precedes T_2 in *Order*$[i]$. □

Theorem 16.8 *If the application follows Rules 1 through 5, then the Timestamp Property is satisfied in every admissible execution.*

Proof. Assume processor p_i executes CompTS$(T_1, T_2), T_1 \neq T_2$, which returns T_j. By Lemma 16.7, timestamps appear in *Order*$[k, i]$ in the correct order. Thus, the NewTS invocation that generated T_j for p_i is later than the NewTS invocation that generated the other parameter for p_i. By Lemma 16.5, the NewTS that generated an active timestamp for p_i is the latest NewTS that generated that timestamp. □

Suppose the single-generator timestamp system is to support an application with r traceable registers. Then the number of (ordinary) single-reader single-writer registers required is $O(r + n)$: $O(1)$ for each traceable register plus n *Order*$[k, i]$ variables.

The number of operations required for each invocation of the NewTS procedure is dominated by the number of operations needed to read the *Viable* and *Prev* variables, which is $O(r)$. (Note, by the way, that the *Viable* and *Order* variables are accessed in NewTS and CompTS by ordinary read and write operations since they are not traceable registers.) The procedure CompTS requires $O(1)$ read and write operations.

An interesting question concerns the size of timestamps and registers. To make sure that NewTS can always pick an unused timestamp, we need to have $|\mathcal{T}| > M$, where M is the number of active timestamps. There can be as many $3r$ different timestamps appearing in all the *Viable* and *Prev* variables. Thus we must have $|\mathcal{T}| > 3r$. Each timestamp value can be described using $O(\log r)$ bits. The largest registers are the *Order*$[k, i]$ registers, each of which requires $O(r \log r)$ bits.

16.1.3 Reducing the Complexity

The number of reads and writes performed in a NewTS operation is dominated by reading the *Viable* and *Prev* registers; this task is sometimes called *garbage collection* since it collects the unused timestamps. A simple improvement is to generate more than one timestamp after each garbage collection, and to collect garbage every once in a while; this requires an increase in the size of the timestamps' pool. For example, if we have at most a active timestamps in each configuration, and we want to collect garbage only every b NewTS operations, then the timestamps could be all integers in the range $[0..a + b + 1]$.

The generator collects garbage (by reading *Viable* and *Prev*) after $b-1$ NewTS operations, and writes b new timestamps to the *Order* registers. The generator uses these timestamps in its b following NewTS operations, until they run out, and then repeats. In this way, after $b - 1$ NewTS operations each with $O(n)$ writes, there is one NewTS operation with $O(r)$ additional reads, where r is the number of traceable registers. By choosing an appropriate value of b, the cost of garbage collection is amortized (Exercise 16.3).

Moreover, most of the cost of NewTS is reading *Viable* and *Prev* for all traceable registers. Since these reads are not done atomically, the procedure remains correct even if they are read in several invocations of NewTS. Therefore, we can spread the reading of *Viable* and *Prev* for all traceable registers several invocations of NewTS, and generate b timestamps after such a collect is completed, for an appropriate value of b. This way, no individual NewTS invocation requires many read operations.

Algorithm 16.3 A bounded simulation of multi-reader register R from single-reader registers.

Initially *Report*$[i,j]$ = *Val*$[i] = (v_0, T_0)$, $1 \leq i,j \leq n$,
where v_0 is the desired initial value of R and T_0 is any timestamp

when read$_r(R)$ occurs: // reader p_r reads from register R
1: $(v[0], T[0]) :=$ tread$_r$(*Val*$[r]$)
2: for $i := 1$ to n do $(v[i], T[i]) :=$ tread$_r$(*Report*$[i, r]$)
3: let j be such that $T[j] =$ MaxTS$(T[0], T[1], \ldots, T[n])$
4: for $i := 1$ to n do twrite$_r$(*Report*$[r, i], (v[j], T[j])$)
5: return$_r(R, v[j])$

when write(R, v) occurs: // the writer writes v to register R
1: $T :=$ NewTS()
2: for $i := 1$ to n do twrite$_0$(*Val*$[i], (v, T)$)
3: ack(R)

16.2 Application: A Bounded Simulation of Multi-Reader Registers

Let us now see how a single-generator timestamp system can be employed to bound the shared memory requirements of a particular algorithm—the simulation of multi-reader registers from single-reader registers (Algorithm 10.2).

The pseudocode appears as Algorithm 16.3. Essentially, it is Algorithm 10.2, with the procedures developed in the previous section used to replace the simple integer timestamps used by the unbounded algorithm. Recall that p_0 is the writer and p_1 through p_n are the readers. Each *Val*$[i]$ is a traceable register twritten by the writer p_0 and tread by p_i, $1 \leq i \leq n$. Each *Report*$[i, j]$ variable is a traceable register twritten by p_i and tread by p_j, $1 \leq i, j \leq n$. Thus there are $O(n^2)$ traceable registers. Note that the unique timestamp T_0 appearing in the initial value of all the traceable registers will be the sole timestamp appearing initially in *Order*$[0, i]$, for each i.

Proving that this algorithm simulates a multi-reader register is done along the lines of the proof of the unbounded simulation (Algorithm 10.2). It is based on the following lemma:

Lemma 16.9 *Suppose timestamp T_1 is twritten to Report$[i]$ and subsequently, timestamp T_2 is twritten to Report$[i]$. Then T_2 is generated by a* NewTS *that is no earlier than the* NewTS *that generates T_1.*

Proof. Suppose, by way of contradiction, this is not true. Consider the earliest twrite tw where this is violated. Let tw' be the previous twrite.

Both twrites are part of read operations by the same processor, and the read containing tw' ends before the read containing tw begins. Prior to tw, the second read operation treads *Report*$[i]$, which was twritten by tw'. Thus the second read observes the result of the first read. By the Timestamp Property, carried over to MaxTS, tw twrites a timestamp that is generated at least as recently as the timestamp twritten by tw'. □

Note that in the unbounded case, this lemma followed trivially since timestamps are increasing integers.

The complete correctness proof of this algorithm is left as an exercise to the reader (Exercise 16.4). In this proof, the Timestamp Property, carried over to MaxTS, replaces the semantics of $\max$ on integers, while the linearizability of the traceable registers is used instead of the linearizability of the ordinary registers.

Let us analyze the complexities of the algorithm. A write operation requires the generation of a new timestamp, and the traceable write of n values. Using the figures calculated earlier, this sums up to $O(n^2)$ low-level ordinary read and write operations on single-writer single-reader registers. A read operation involves traceable reading of $n + 1$ variables, calculation of the maximum timestamp, and forwarding to n other processors. Using the figures calculated earlier, this sums up to $O(n)$ low-level read and write operations.

The dominating factor in the space complexity is the number of bits required to represent the bounded timestamps. This number depends on the number of timestamps that can be active in the system, which is $O(n^2)$ (Exercise 16.5). Thus, $O(\log n)$ bits are needed to store a single timestamp. The largest registers are the *Order*$[0, i]$ variables, each containing $O(n^2 \log n)$ bits.

16.3 Concurrent Timestamp System

We now turn to show a timestamp system in which some number, say m, of the n processors can generate timestamps, possibly in overlapping invocations of the procedure NewCTS (the 'C' stands for 'concurrent'). Every processor may potentially need to find the more recent of two timestamps with the procedure CompCTS. Such timestamp systems can be used to replace the unbounded integer timestamps used by the simulation of multi-writer registers (Algorithm 10.3) and in other applications discussed in the notes at the end of the chapter. The basic idea is very similar to the vector timestamps used in the multi-writer simulation; each timestamp is a vector with m entries, where m is the number of generators. However, each entry, instead of attaining unbounded integer values, contains a bounded timestamp from the single-generator timestamp system (described in the previous section).

Specifically, a concurrent timestamp system must support the following two procedures:

NewCTS$_i$**:** invoked by processor p_i, $0 \leq i \leq n-1$, if p_i is a generator; takes no parameters and returns a timestamp.

CompCTS$_j$**:** invoked by any processor p_j, $0 \leq j \leq n-1$; takes two timestamps as parameters and returns one of its parameters.

Informally speaking, CompCTS should return the parameter that was produced more recently by a NewCTS.

When there is a single generator and NewTS operations are executed sequentially, the notion of 'most recent' is well-defined. This is not the case for a concurrent timestamp system where NewCTS operations by different processors may overlap. Instead, we require the existence of a total order on all NewCTS operations that extends their real-time order, and we require a CompCTS operation to return the parameter that was produced most recently, with respect to this order. This property is formally stated as follows:

Concurrent Timestamp Property: There exists a total order $\Rightarrow$ on all NewCTS operation executions that extends the real-time order on non-overlapping operations, such that the following holds.

Consider any CompCTS execution *ct* on arguments VT_0 and VT_1 that returns VT_i. Let *nt* be the NewCTS execution that returns VT_i and ends most recently before *ct* begins. Then VT_{1-i} was not returned by a NewCTS operation that precedes *ct* in $\Rightarrow$.

Clearly, CompCTS can be used to implement a MaxCTS procedure that finds the maximal timestamp among its parameters, in which comparisons are done using CompCTS.

The implementation of a concurrent timestamp system that we will now describe uses *vectors* of (single-generator timestamp system) timestamps as the concurrent timestamps. Another important difference from the single-generator case is that we must provide traceable registers that are *multi-reader*. These two differences cause much of the complications.

The basic idea of the algorithm, as mentioned above, is to use vectors of single-generator timestamps, which will be manipulated as in the multi-writer register simulation of Chapter 10. Thus we use the single-generator procedures NewTS and CompTS in order to generate and compare the (scalar) entries in the vectors. Each entry corresponds to a particular generator. We will have separate copies of the procedures for each possible generator.

In the single-generator case, the application using NewTS and CompTS must obey certain rules to ensure that timestamps are traceable and thus the Timestamp Property is satisfied. According to these rules, timestamps must be transferred between processors using traceable writes and reads (twrite and tread). In the multi-generator case, the NewCTS and CompCTS procedures can be viewed as applications using the single-generator procedures NewTS and CompTS. Thus

we need analogous rules here to ensure that the vector timestamps, which contain single-generator timestamps, are transferred traceably. However, we cannot use twrite and tread unchanged from the single-generator case, because the traceable register abstraction now needs to support multiple readers—it is very helpful to be able to share a traceable register containing a vector timestamp with all the readers. Thus, we modify twrite and tread to handle multiple readers. We call the resulting procedures mtwrite and mtread.

Finally, the application that is using the concurrent timestamp system needs to obey certain rules to ensure that the Concurrent Timestamp Property is satisfied. The processors are constrained to call NewCTS and CompCTS in certain ways, and vector timestamps must be transferred traceably. Conveniently enough, the vector timestamps can be transferred by the CTS user in the same way as by the CTS implementation, using mtread and mtwrite.

We first explain the traceable read and write for a multi-reader traceable register, and then we present the concurrent timestamp procedures.

16.3.1 A Multi-Reader Traceable Register

Consider a traceable register R that is to be traceably written by one processor p_j (denoted $\mathsf{mtwrite}_j$) and traceably read by several processors (denoted mtread_i for each treader p_i). The procedures mtread and mtwrite are very similar to their single-reader counterparts. To handle the multiple readers, during the $\mathsf{mtwrite}_j$, p_j checks for a handshake with each potential reader p_i. If a handshake is detected for p_i, then p_j updates a separate version of the variables *Copy* and *Viable* just for p_i.

The variables *Value* and *Prev* are multi-reader variables, written by p_j and read by all the treaders of R. Such multi-reader variables can be constructed from single-reader registers using the multi-reader register simulation from Chapter 10. The single-generator bounded timestamp system used to provide these multi-reader registers is inside the register, which is a black box to the multi-reader traceable read and write procedures whose implementation we are now describing.

Detailed pseudocode appears as Algorithm 16.4. The identities of the mtwriter p_j and the mtreader p_i for the handshaking procedures are indicated by the superscript (j, i).

The following two theorems state that the multi-reader version of the traceable read and write procedures satisfy the same properties as the original (Theorems 16.2 and 16.4). Moreover, the procedures are linearizable even if multiple readers may call mtread operations on the same traceable register. The proofs are left to Exercise 16.6 and Exercise 16.7.

Theorem 16.10 *Algorithm 16.4 guarantees the following in every admissible execution α: Let tr_1 be a* mtread *by p_r in α that returns a value v and let tw_1 be the* mtwrite *by p_w in α that tr_1 reads from. Let tr_2 be the first* mtread *by p_r in α*

Algorithm 16.4 Implementation of a multi-reader traceable register to support the bounded multi-generator system: code for the mtwriter p_j and every mtreader p_i.

Initially *Value* = *Prev* = v_0, *Copy*$[i] = v_0$ and *Viable*$[i] = \langle v_0, v_0 \rangle$ for all i such that p_i is an mtreader, where v_0 is the initial value of the traceable register

```
procedure mtread_i():                         // for every p_i that is an mtreader
1:  tryHS_i^{j,i}()
2:  v := Value
3:  if checkHS_i^{j,i}() then v := Copy[i]
4:  return v

procedure mtwrite_j(v):                       // for the mtwriter p_j
1:  Value := v
2:  for all i such that p_i is an mtreader do
3:      if checkHS_j^{j,i}() then
4:          Copy[i] := v
5:          Viable[i] := ⟨Prev, v⟩
6:          tryHS_j^{j,i}()
7:  Prev := v                                 // outside the for and the if
```

after tr_1 *that does not read from* tw_1. *Then* v *is in either* *Prev*$[r]$ *or* *Viable*$[r]$ *in every configuration of* α *from the end of* tw_1 *to the beginning of* tr_2.

Theorem 16.11 *Algorithm 16.4 wait-free simulates a multi-writer single-reader (linearizable) register.*

Let k be the number of mtreaders of the multi-reader traceable register. Clearly the number of ordinary read and write operations in mtread is $O(1)$ and in mtwrite is $O(k)$. The number of shared variables required is $O(k)$.

16.3.2 The Multiple-Generator Timestamp Algorithm

As mentioned earlier, the algorithm is very much like the construction embedded in the multi-writer register simulation (Algorithm 10.3), which in turn is very similar to vector timestamps studied in Chapter 6. Each timestamp is a vector with m entries: The k-th entry of the vector contains a bounded timestamp from the single-generator timestamp system of p_k, the k-th generator.

The multiple-generator timestamp system uses a single-generator timestamp system for each generator, p_k. Recall that the single-generator algorithm treated values as typed objects that contain timestamps and possibly other components. Thus, each generator only handles the k-th coordinate of the vector timestamp,

Algorithm 16.5 A bounded multi-generator timestamp system.

Initially $VTS[k] = \langle v_0, \ldots, v_{n-1}\rangle$, $0 \le k \le m-1$,
where each v_i is the timestamp in the initial value of *Order*$[i, i]$

procedure NewCTS$_k$: // for each p_k that is a generator
1: for $j := 0$ to $m-1$ do
2: $lvts[j] :=$ mtread$_i(VTS[j]).[j]$ // extract the jth coordinate
3: $lvts[k] :=$ NewTS$_k()$
4: mtwrite$_k(VTS[k], lvts)$
5: return *lvts*

procedure CompCTS$_i(VT_1, VT_2)$: // for every p_i
1: if $VT_1 = VT_2$ then return VT_1
2: let k be the smallest index in which the parameters do not agree
2: let j be such that $VT_j[k] =$ CompTS$_i(k, VT_1[k], VT_2[k])$
// parameter k indicates generator k
3: return VT_j

and treats other coordinates (containing timestamps of other generators) as uninterpreted values.

In addition, for each generator p_k, $0 \le k \le m-1$, there is a shared variable:

VTS$[k]$**:** a single-writer multi-reader traceable register containing a vector of scalar timestamps, written by p_k and read by all processors.

The pseudocode appears as Algorithm 16.5. Note that the procedure NewCTS is the bounded analog of unbounded procedure NewCTS in Algorithm 10.3.

Assume that the application using the concurrent timestamp system obeys the following rules.

Rule 1$'$. Only one processor, p_k, calls NewCTS$_k$.

Rule 2$'$. For each ordered pair of distinct processors (p_i, p_j), p_i transmits vector timestamps to p_j only via a finite set of known registers. Scalar timestamps are only transmitted as part of vector timestamps.

Rule 3$'$. Whenever a processor writes a value containing a vector timestamp, it uses mtwrite, and only timestamps generated by p_k appear in entry k of the vector, $0 \le k \le m-1$.

Rule 4$'$. Whenever a processor reads a value containing a vector timestamp, it uses mtread.

These rules are similar to Rules 1 through 4 specified in Section 16.1.1: The transmission of single-generator timestamps is strictly controlled. Rule 3$'$

requires that each coordinate in a forwarded vector timestamp be generated by the corresponding unique generator.

We also require a rule that is the analog of Rule 5, but first we must define what it means for a vector timestamp to be active. A vector timestamp T is *active* for processor p_i via processor p_j after some finite prefix of an execution if:

– $p_j = p_i$ and T is the value returned by the most recent NewCTS_i invocation; or

– $p_j \neq p_i$ and either the most recent mtwrite by p_j to one of the registers read by p_i or the most recent mtread by p_i from one of the registers written by p_j contains T.

A key feature we use below is (see Exercise 16.8).

Lemma 16.12 *If a vector timestamp T is active after some prefix of the execution, then each component $T[k]$ is active in the single-generator timestamp system of p_k after this prefix.*

We impose the following additional requirement on the application:

Rule 5′. All vector timestamps appearing in parameters to CompCTS and mtwrite are active.

It follows that:

Lemma 16.13 *If the application obeys Rules 1′ through 5′, then the projection of an execution for a particular generator p_k corresponds to an execution of the single-generator timestamp system for p_k that obeys Rules 1 through 5.*

Using the notion of active vector timestamp, we can extend the definitions of forwarding distance and generating NewCTS operation to vector timestamps.

To prove the correctness of the multi-generator timestamp system, we first prove that if a vector timestamp is active, then all its component (scalar) timestamps are in the appropriate *Order* variables; this is the equivalent of Lemma 16.6.

Lemma 16.14 *If vector timestamp T is active for p_i via p_j after some finite prefix of α, then $T[k]$ is in Order$[k, i]$, for all k and i, in every configuration of the prefix after the completion of the NewCTS that generated T for p_i.*

Proof. By Lemma 16.13, the projection of the execution for generator p_k is an admissible execution of the single-generator timestamp system. Since T is active, each component $T[k]$ is active, by Lemma 16.12. By Lemma 16.6, $T[k]$ is in *Order*$[k, i]$. □

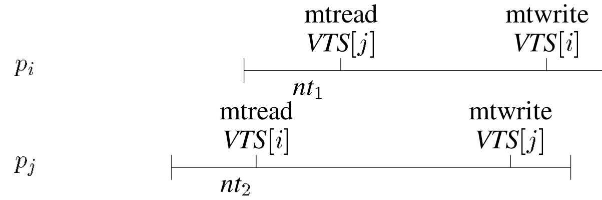

Figure 16.7: Overlapping NewCTS operations.

To prove that Algorithm 16.5 satisfies the Concurrent Timestamp Property, we must first define a total order $\Rightarrow$ on all NewCTS operation executions.

Consider two NewCTS operation executions, nt_1 by p_i and nt_2 by p_j.

If the mtwrite to $VTS[i]$ in nt_1 is linearized before the mtread from $VTS[i]$ in nt_2, then we define $nt_1 \Rightarrow nt_2$; if the mtwrite to $VTS[j]$ in nt_2 is linearized before the mtread from $VTS[j]$ in nt_1, then we define $nt_2 \Rightarrow nt_1$. (Exercise 16.9 asks you to prove that these two cases do not occur simultaneously.) If nt_1 and nt_2 are non-overlapping, then one of the above conditions holds, and the order defined between nt_1 and nt_2 extends the real-time order between them.

Suppose now that neither of the above conditions holds (see Figure 16.7); then nt_1 and nt_2 overlap, and therefore, $i \neq j$. Let T_1 be the timestamp generated by nt_1, and T_2 be the timestamp generated by nt_2.

We first argue that $T_1 \neq T_2$. Without loss of generality, assume that p_j mtreads $VTS[i]$ before p_i mtreads $VTS[j]$.

$T_2[i]$ is active with positive forwarding distance from p_j's mtread of $VTS[i]$ in nt_2 onward. By Lemma 16.5 and Lemma 16.13, $T_2[i]$ is in a *Prev* or *Viable* variable once it has positive forwarding distance. When nt_1 performs NewTS$_i$, it observes $T_2[i]$ as potentially in use and thus $T_1[i] \neq T_2[i]$.

Since $T_1 \neq T_2$, there exists a smallest coordinate k in which T_1 and T_2 differ. Since $VTS[k]$ is linearizable, the mtreads from $VTS[k]$ in nt_1 and in nt_2 are ordered. If p_i's mtread from $VTS[k]$ in nt_1 is linearized before p_j's mtread from $VTS[k]$ in nt_2 (as in Figure 16.8) then we define $nt_1 \Rightarrow nt_2$; otherwise, we define $nt_2 \Rightarrow nt_1$.

We now prove the main theorem of this chapter:

Theorem 16.15 *If the application obeys Rules 1′ through 5′, then Algorithm 16.5 satisfies the Concurrent Timestamp Property in every admissible execution.*

Proof. Exercise 16.10 asks you to prove that $\Rightarrow$ is a total order on all NewCTS operations, which preserves the real-time order of non-overlapping operations.

Assume processor p_l executes CompCTS(T_1, T_2), $T_1 \neq T_2$, which returns T_1. Let nt_1 be the NewCTS by processor p_i that generated T_1 for p_l, and let nt_2

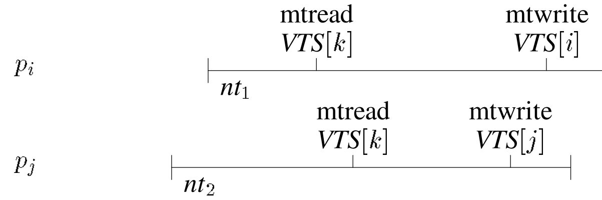

Figure 16.8: Defining the order of overlapping operations.

be the NewCTS by processor p_j that generated T_2 for p_l. We have to prove that $nt_2 \Rightarrow nt_1$.

Assume, by way of contradiction, that $nt_1 \Rightarrow nt_2$. Consider the smallest coordinate k in which T_1 and T_2 differ. The definition of $\Rightarrow$ implies that p_i's read from $VTS[k]$ in nt_1 is before p_j's read from $VTS[k]$ in nt_2 (again, consider Figure 16.8).

Lemma 16.12 implies that $T_1[k]$ is active from the time p_i mtreads $VTS[k]$ during nt_1 onwards, since T_1 is active from this point onward. By the linearizability of $VTS[k]$, when p_j reads $VTS[k]$ in nt_2, it obtains a value that is at least as recent as $T_1[k]$. By assumption, $T_1[k] \neq T_2[k]$, and thus it must be that the NewTS_k that generated $T_1[k]$ completed before the NewTS_k that generated $T_2[k]$. The relationship between the NewTS_k executions, together with Lemma 16.13 and Lemma 16.7, implies that $T_1[k]$ is before $T_2[k]$ in $Order[k, l]$, when p_l executes $\mathsf{CompCTS}(T_1, T_2)$. The pseudocode for CompCTS implies p_l should return T_1, which is a contradiction. □

Exercise 16.12 asks you to verify that the number of read and write operations on ordinary single-writer multi-reader registers performed by *NewCTS* is $O(r+m)$, where r is the number of traceable registers, and m is the number of generators, and the number of read and write operations performed by *CompCTS* is $O(1)$. Exercise 16.13 asks you to calculate the number of single-writer multi-reader registers and their size.

16.4 Application: A Bounded Simulation of Multi-Writer Registers

In the previous section, we have seen how the single-generator timestamp system can replace the unbounded integer timestamps used in the simulation of multi-reader registers from single-reader registers. In this section, we shall see how the concurrent timestamp system can replace the vectors of unbounded integers

Algorithm 16.6 Bounded simulation of a multi-writer register R from single-writer registers: code for readers and writers.

Initially $Val[i] = (v_0, T_0)$, where v_0 equals the desired initial value of R and T_0 is an arbitrary vector timestamp, $0 \leq i \leq m-1$

when read$_r(R)$ occurs: // reader p_r reads from register R, $0 \leq r \leq n-1$
1: for $i := 0$ to $m-1$ do $(v[i], vts[i]) :=$ mtread$_r(Val[i])$
2: let j be such that $vts[j] =$ MaxCTS$_r(vts[1], \ldots, vts[m])$
3: return$_r(R, v[j])$

when write$_w(R, v)$ occurs: // writer p_w writes v to register R, $0 \leq w \leq m-1$
1: $vts :=$ NewCTS$_w()$
2: mtwrite$_w(Val[w], (v, vts))$
3: ack$_w(R)$

used as timestamps in the simulation of multi-writer registers from single-writer registers.

The pseudocode appears in Algorithm 16.6. It is essentially Algorithm 10.3, with procedures NewCTS and MaxCTS, developed in the previous section, replacing the simple manipulations of the integer vector timestamps done in the unbounded algorithm. The m writers are $p_0, \ldots, p_{m-1}$ and all of the processors, $p_0, \ldots, p_{n-1}$, are the readers. Each $Val[i]$ variable is a multi-reader single-writer traceable register, written by p_i and read by all the processors. Note that the timestamp T_0 contained in the initial value of $Val[i]$ will be the sole entry in each $Order[i, j]$

Proving that this algorithm simulates a multi-writer register follows the lines of the proof of the unbounded simulation (Algorithm 10.3).

The candidate linearization order π is defined in two steps. First, we put into π all the write operations according to the $\Rightarrow$ order on the timestamps associated with the values they write. Second, we consider each read operation in turn, in the order in which its response occurs in α. A read operation that returns a value associated with timestamp *VT* is placed immediately before the write operation that follows (in π) the write operation that generated *VT*.

Since $\Rightarrow$ extends the real-time order of NewCTS operations, we have the analog of Lemma 10.7. The Concurrent Timestamp Property implies the following analog of Lemma 10.8:

Lemma 16.16 *If VT_1 is written to $Val[i]$ and later VT_2 is written to $Val[i]$, then any execution of CompCTS(VT_1, VT_2) returns VT_2.*

The complete proof of correctness of Algorithm 16.6 is left to Exercise 16.14; it uses the above lemmas, replaces the semantics of $\max$ on vector timestamps

with the correctness of MaxCTS, and uses the linearizability of mtread and mtwrite. Exercise 16.14 also requests you to analyze the complexity of the algorithm.

Exercises

16.1 Prove that the forwarding distance of an active timestamp is finite.

16.2 A simple implementation of MaxTS from CompTS reads *Order* in each invocation; write a direct implementation of MaxTS that reads *Order* only once, and prove it satisfies the Timestamp Property, carried over to MaxTS.

16.3 Find the value of b such that if garbage is collected every b NewTS operations, then the amortized number of read and write operations per NewTS is $O(1)$.

16.4 Prove that Algorithm 16.3 is a simulation of a single-writer multi-reader register from single-writer single-reader registers.

16.5 Show that the number of active timestamps in Algorithm 16.3 is $O(n^2)$.

16.6 Prove Theorem 16.10.

16.7 Show that the mtread and mtwrite procedures wait-free simulate a single-writer multi-reader register (Theorem 16.11).

Hint: The main property you need to prove is linearizability; this can be done by following the proof of Theorem 16.4 for mtreads by different readers.

16.8 Prove Lemma 16.12.

16.9 Consider two NewCTS operation executions, nt_1 by p_i and nt_2 by p_j. Prove that it cannot happen that both the write of nt_1 to $VTS[i]$ is before the read from $VTS[i]$ in nt_2, and the write of nt_2 to $VTS[j]$ is before the read from $VTS[j]$ in nt_1.

16.10 Prove that $\Rightarrow$ is a total order on all NewCTS operations that preserves the real-time order of non-overlapping operations.

16.11 A simple implementation of MaxCTS from CompCTS reads an *Order* variable in each invocation; write a direct implementation of MaxCTS that reads *Order* only once, and prove it satisfies the Concurrent Timestamp Property, carried over to MaxCTS.

16.12 Prove that the number of read and write operations on ordinary single-writer multi-reader registers performed by *NewCTS* is $O(r + m)$, where r is the number of traceable registers and m is the number of generators, and the number of read and write operations performed by *CompCTS* is $O(1)$.

16.13 For the concurrent timestamp system presented in Section 16.3, calculate the number of single-writer multi-reader registers and their size.

16.14 Prove that Algorithm 16.6 simulates a multi-writer register. Analyze the number of ordinary read and write operations performed on single-writer single-reader registers and the space usage.

16.15 Use a bounded concurrent timestamp system to bound the sequence numbers in the universal simulations, given in Chapter 15, of shared objects from consensus objects (e.g., Algorithm 15.4).

Chapter Notes

Timestamps were originally abstracted by Israeli and Li [137]. They presented an algorithm for the case timestamps are generated *sequentially*, i.e., NewTS invocations do not overlap, but are not necessarily performed by a single generator. Dolev and Shavit refined their definition and presented an algorithm for a *concurrent* timestamp system [97].

Our constructions are based on the concurrent timestamp algorithm of Dwork and Waarts [102]. The synchronization structure in our tread and twrite (and in mtread and mtwrite) procedures follows Haldar [126], who, in turn, follows Vidyasankar's simulation of single-writer multi-reader registers with weaker types of registers [251]. The treatment of forwarding (of timestamps between readers) is inspired by the work of Attiya, Bar-Noy and Dolev [22], where timestamps are used in the simulation of shared memory with message passing.

The algorithms of Dwork and Waarts and of Haldar reduce the number of memory accesses (reads and writes) and the space complexity (number of registers) by careful combination of the tracing mechanisms for several registers.

Our presentation extracts the single-generator timestamp algorithm embedded in the concurrent timestamp algorithm of Dwork and Waarts. As we have seen, single-generator timestamps have interesting applications; we also believe this improves the understanding of this complex algorithm.

Dwork, Herlihy and Waarts [100] have shown how to use traceable registers to bound the shared memory requirements of algorithms based on asynchronous phases. Chapter 17 contains another application of bounded timestamps.