# Bounded Timestamps in Process Networks

Ambuj K. Singh
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, California 93106

June 30, 1992

## 1 Introduction

Timestamps are a useful mechanism for understanding and analyzing the causality of distributed computation. They have been used to solve a wide class of problems including mutual exclusion [8], recovery [7], debugging [3, 4], and causal broadcasts [1]. A significant drawback of the existing implementation of timestamps is that they become potentially unbounded in size. As a result, algorithms based on timestamps must allow for unbounded messages or rely upon some techniques for global resetting of timestamps [1]. In this paper we investigate the reusability of timestamps and present a simple implementation of bounded timestamps.

Recall the mechanism used for updating and communicating timestamps [8, 9].

- Initially the current timestamp of every process is initialized to 0.
- Upon the occurrence of an event, a process advances its current timestamp. Thus, if $cts_i$ denotes the current timestamp of process $i$, then $cts_i := next(cts_i)$.
- While sending a message, a process attaches the new timestamp to the message.
- Upon receiving a message, a process sets its current timestamp to the maximum of its current timestamp and the timestamp of the message. In other words, process $i$ executes $cts_i := sup(cts_i, ts)$ where $ts$ is the timestamp of the received message, and $sup$ is the *supremum* function over the domain of the timestamps.

The functions $next$ and $sup$ are implemented so that the causality of events corresponds to their timestamps. Thus, in the scalar timestamps mechanism of Lamport [8], function $next$ returns the next natural number and function $sup$ is defined to be the maximum of naturals. In the vector timestamps mechanism [9], on the other hand, function $next$ increments the process's component of the timestamp and function $sup$ is defined to be element-wise maximum of vectors. Note that in case of vector timestamps processes choose their timestamps from independent domains (i.e., process $i$ is the sole generator of values in the $ith$ component of the vector). This simplifies the reuse of timestamps and consequently, we concentrate on vector timestamps in the remainder of the paper.

## 2 The Basic Idea

**Notation:** Given a timestamp vector $x$, let $x[i]$ denote the *ith* component of $x$, i.e., the component corresponding to process $i$. Let $x, y, z$ range over timestamps and let $m, n$ be scalars ranging over values in vector components. □

Suppose that a process $i$ wishes to reuse a timestamp component $m$. Then, in order for the timestamping scheme to work correctly, all old messages with value $m$ in the *ith* component should have been delivered. In other words, for any process $j$ and any outgoing channel $c$ of process $j$, the *ith* component of $cts_j$ and the *ith* components of all the timestamps in channel $c$ should differ from $m$. For simplicity, let us label the processes in the network as either *white* or *black*. A process is said to be *black* iff its current timestamp exceeds $m$ (the timestamp component that we wish to reuse), and none of its outgoing channels contain a message with a timestamp component $m$. Then, based on the timestamping algorithm, it is possible to show that a black process remains black. Process $i$ can reuse value $m$ once all the processes in the network have become black. A number of mechanisms can be used for inferring whether a process has become black and for inferring whether all the processes in the network have become black [2]. We outline one such combination of mechanisms next.

We construct a spanning tree with process $i$ as the root. A process $j$ in the tree maintains a set of variables $last_k$ as an estimate of the oldest value occurring along a channel to a neighboring process $k$ and a variable $old_j$ as an estimate of the oldest timestamp occurring in $j$ or a descendant node of $j$ in process $i$'s spanning tree. (The subtree rooted at process $j$ becomes black with respect to a value $m$ iff $old_j > m$.) All the children of process $j$ inform it of their respective *old* values and process $j$ in turn computes the minimum of all these values and $last_k$ as its own *old* value and forwards this value to its parent in the spanning tree. Timestamp component $m$ can be reused by process $i$ only if its *old* value becomes greater than $m$. One way to compute $last_k$ is for process $k$ to inform process $j$ of the most recent timestamp that it received from process $j$. All of the above values can be sent along with messages and timestamps of the underlying computation and do not require extra communication. The extra information that process $k$ needs to send to a process $j$ consists of the last timestamp that it received from process $j$ and an *old* field for each spanning tree in which process $j$ is a parent of process $k$.

What we have developed so far is a necessary condition for a timestamp component to be reused. But can a process $i$ really reuse a value $m$ if it does not occur in the processes and the channels of the network? As an example, assume that process $i$ is using values $0 \cdots 3$ in a round-robin fashion reusing a value $m$ once it has disappeared from the network. Suppose that some other process $j$ receives two messages with values 0 and 2. Process $j$ cannot order the message with value 0 before the message with 2 because the following sequence of events may have occurred.

- Process $i$ uses values $0, 1$, and 2.
- Value 0 disappears from the network and process $i$ reuses value 0.
- Process $j$ receives messages with values 0 and 2.

Similarly, process $j$ does not order the message with value 2 before the message with 0 because the message with value 0 may not have been a reuse.

Thus, in order for a process $i$ to reuse a value, every other process $j$ should be able to correctly infer the ordering of all existing values that originated from process $i$. To clarify, we define a timestamping system similar to the *sequential timestamp system* defined by Israeli and Li [6]. As in there, we define a directed graph in which each node contains a value (a timestamp component corresponding to a process). A directed edge $u \rightarrow v$ denotes that $u$ is an older value than $v$. We introduce pebbles to refer to nodes whose values are currently being used. In order for a process to reuse values, a pebble is transferred from the oldest pebbled node (i.e., a node that has an outgoing edge to all other pebbled nodes) to a new unpebbled node that is more recent than (i.e., that has an outgoing edge to) all remaining pebbled nodes. (This is in contrast to the *sequential timestamp system* where a pebble can be moved from any pebbled node and not necessarily the oldest pebbled node.) A constraint on the system is that all the pebbled nodes are totally ordered at any point in time.

**Theorem:** A timestamping system that allows for up to $k$ pebbles at a time has at least $2k-1$ nodes.

**Proof:** Assume a timestamping system with $n$ nodes that allows for up to $k$ pebbles at any time. Consider a node $u$ that gets a pebble. At that instant, it has an incoming edge from all remaining pebbled nodes. Consequently, the in-degree of node $u$ is at least $k-1$. This implies that the graph has at least $n(k-1)$ edges. Since the graph does not contain any two-edge loops, the maximum number of edges in the graph equals $n(n-1)/2$. Consequently, $n(k-1) \leq n(n-1)/2$. The desired result follows from this inequality. □

Based on the above theorem, we implement the following mechanism for reusing timestamps. In order to allow for up to *size* active values at any time, a process chooses a domain of values $0 \cdots 2 * size - 1$. At any point in time, process $i$ uses a segment of values $m \cdots m \oplus (size - 1)$. (Here $\oplus$ denotes addition modulo $2 * size - 1$.) When value $m$ disappears from the network, the process advances the segment of active values to $m \oplus 1 \cdots m \oplus size$ by reusing value $m \oplus size$. A process $j$ in the network orders value $m$ before value $n$ iff the directed distance from $m$ to $n$ is less than that of $n$ to $m$, i.e., iff $n \ominus m < m \ominus n$ ($\ominus$ denotes subtraction modulo $2 * size - 1$).

## 3 The Complete Algorithm

Assume that each process uses up to *size* distinct values in its component at any time. As suggested earlier, the processes choose their values in a domain $0 \cdots 2 * size - 1$. Let $T_k$ denote the spanning tree corresponding to process $k$. A process $i$ maintains the following variables.

- *cts* that stores the current timestamp of process $i$. It is initialized to a vector of all zeros.
- *window* that is a 2-tuple storing the smallest and the largest value for process $i$ that are currently in use. It is initialized to $(0, size - 1)$.
- $recent_j$ that stores the most recent vector timestamp that process $i$ received from process $j$. It is initialized to a vector in which ever value is $2 * size - 1$.

- $old_k$ that stores the oldest value corresponding to $T_k$ that occurs in $i$ or a descendant node of process $i$ in $T_k$. It is initialized to $2 * size - 1$.

- $desold_k$ that stores the oldest value corresponding to $T_k$ that occurs in a descendant node of process $i$ in $T_k$. It is initialized to $2*size-1$ if process $i$ has any descendants in $T_k$; otherwise it is initialized to infinity.

The updating and the communication of timestamps is achieved by the following set of actions.

- Upon occurrence of an event, process $i$ advances its current timestamp $cts$ by incrementing the field $cts[i]$ modulo $2*size-1$. If the set of current values is exhausted, i.e., $cts[i] = window[1]$, then the process is blocked until $window[1]$ advances.

- When process $i$ sends a message to process $j$, it attaches its current timestamp $cts$, the most recent timestamp received from process $j$ $recent_j$, and a set of values $old_k$ for each $k$ such that process $j$ is the parent of process $i$ in $T_k$.

- Upon receiving a message from process $j$ with a timestamp $ts$, value $recent_{ji}$ that is the most recent timestamp that process $j$ received from process $i$, and a set of values $old_{jk}$ where $i$ is the parent of $j$ in $T_k$ and $old_{jk}$ is the oldest value existing the subtree of $T_k$ with $j$ as the root, process $i$ does the following:

  1. $cts := max(cts, ts)$, i.e., updates the current timestamp by computing an elementwise maximum (modulo $2 * size - 1$) of the old value of $cts$ and the received timestamp,
  2. $recent_j := ts$, i.e., sets $recent_j$ to the new timestamp received from process $j$,
  3. for each $k$ such that $i$ is the parent of $j$ in $T_k$, $desold_k := old_{jk}$, i.e., sets $desold_k$ to be the smallest existing value corresponding to $T_k$ that occurs in a descendant node of process $i$ in $T_k$,
  4. for each $k$, $temp_k := (min\ l :: recent_l[k])$, i.e., computes a temporary value $temp_k$ to be the oldest value corresponding to $T_k$ that exists in the outgoing channels of process $i$ (minimum values are computed modulo $(2 * size - 1)$),
  5. for each $k$, $old_k := min(desold_k, temp_k)$, i.e., computes $old_k$ to be the smallest existing value corresponding to $T_k$ that occurs in $i$ or a descendant node of process $i$ in $T_k$,
  6. If $old_i > window[0]$ then $window := (old_i, old_i \oplus (size - 1))$, i.e., updates the set of values that can be used if some old values have disappeared from the network.

# 4 Discussion

A number of authors have considered the question of bounded timestamps in shared variable systems [6]. The problem is more difficult to solve in that setting on account of the constraints of wait-freedom [5]. The same constraint cannot be applied to message-passing networks as any communication necessarily involves waiting.

The number of active values that a process should allow for at any time depends upon the frequency of communication between neighbors, the height of the spanning trees, and the message transmission delay. Assume that any two neighboring processes communicate with each other within time $\tau$ and let $h$ be the maximum height of the spanning trees. Consider a value $m$ used by a process $i$. Within time $h\tau$ this value is communicated to all the processes and within another $h\tau$ time, this value has been subsumed by a later value $m \oplus 1$ in all the nodes and the channels. Within another $h\tau$ time, process $i$ detects that value $m$ has disappeared from the network. Summing these up, it follows that within $3h\tau$, process $i$ discovers that value $m$ has disappeared and consequently is able to advance its window. Let $\nu$ be the frequency with which a process performs a send, a receive, or an internal event. Then, the number of values that the process will need within a time interval of $3h\tau$ equals $3h\tau\nu$. Consequently, we should choose *size* to be at least $3h\tau\nu$ in order to make sure that a process does not block due to unavailability of values. The domain of values used by a process therefore should be at least $6h\tau\nu - 1$. This means that the number of bits required for each component of a timestamp is roughly $\log 6h\tau\nu$.

## References


[1] Birman, K., A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Broadcast", ACM Transactions on Computer Systems, 9:3, pp. 272–314, August 1991.

[2] Chandy, K. M., and J. Misra, *Parallel Program Design: A Foundation*, Reading, Massachusetts: Addison-Wesley, 1988.

[3] Cooper, R., and K. Marzullo, "Consistent Detection of Global Predicates", Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, pp. 163–173, May 1991.

[4] Fowler, J., and W. Zwaenepoel, "Causal Distributed Breakpoints", Proceedings of the 10th International Conference on Distributed Systems, Paris, France, pp. 131 – 141, May 1990.

[5] Herlihy, M., "Wait-Free Implementations of Concurrent Objects," Proceedings of the Sixth ACM Symposium on the Principles of Distributed Computing, 1988, pp. 276 – 290.

[6] Israeli, A., and M. Li, "Bounded Time-stamps", Proceedings of Twenty-eighth Annual Symposium on Foundations of Computer Science, pp. 371 – 382, 1987.

[7] Johnson, D. B., and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing", Journal of Algorithms, 11:3, pp. 462–491, Sep. 1990.

[8] Lamport, L., "Time, Clock, and the Ordering of Events in a Distributed System", Communications of the *ACM*, 21:7, July 1978, pp. 558 – 565.

[9] Mattern, F., "Virtual Time and Global States in Distributed Systems", in *Parallel and Distributed Algorithms*, M. Cosnard et al. (eds.), Elsevier/North Holland, pp. 215–226, 1989.