Shrinking Timestamp Sizes of Event Ordering Protocols

Achour Mostéfaoui
IRISA, Campus de Beaulieu
35042 Rennes Cedex, FRANCE
achour@irisa.fr

Oliver Theel
Darmstadt University of Technology
D-64283 Darmstadt, Germany
theel@informatik.tu-darmstadt.de

Abstract

Almost all published work on causal ordering mechanisms assumes theoretically unbounded counters for timestamps, thus ignoring the real-world problem that arises if one is actually interested in an operable implementation, since unbounded counters simply cannot be realized. An argument for its justification often encountered states, that the counter size can be chosen such that counters practically do not overflow or wrap-around. For example, using matrix timestamps in a distributed computation involving not more than 50 processes and 32 bits per integer results in a timestamp size of almost 10 K-byte. In this paper, we present a solution, called Factorized Timestamp Approach (FTA) that substantially reduces the amount of piggybacked control information. It is based on introducing the notion of phases in which much smaller timestamps are used. Simulation results given in the paper show the suitability of this approach.

1 Introduction

Causal ordering of events is a very important communication abstraction of a distributed computing system. Various implementations for such an abstraction have been presented in the literature, among them Lamport clocks [4], vector clocks [5], and matrix clocks [9]. These implementations are mainly based on the notion of logical time. Clocks are represented by a scalar, a vector, or a matrix of counters (integer variables whose value strictly increase).

Informally, one can say that the more complex the informational structure of a timestamp, the more reasoning can be done about the order of events. Furthermore, the less restrictive the event ordering, the less synchronization delay enforced by the event ordering subsystem onto the distributed application. Unfortunately, more complex structured timestamps tend to have higher storage requirements. For example, a timestamp derived from a Lamport clock consists of a single integer value regardless of the number of processes whose events need to be ordered. Using vector clocks, the storage complexity increases to $O(n)$ integer values for a single timestamp with $n$ being the number of processes whose events need to be ordered. In case of matrix clocks the complexity is of order $O(n^2)$.

A straightforward solution to reduce timestamp sizes even when using vector or matrix clocks, is to limit the storage requirements per integer. Unfortunately, if the origin and the number of events to be ordered is unknown, then limiting the number of bits increases the danger of integer overflow or integer wrap-around. Both cases are likely to cause an abnormal termination of the application.

The approach for reducing timestamp sizes presented in this paper is as follows. We reduce the number of bits allocated per counter but logically “intercept” counter overflow by the event ordering subsystem. Whenever a counter overflows, a new event ordering phase (or simply phase) is started and all counters are reset to zero. Each phase is associated with a unique phase number. The timestamps given to events are extended to include the current phase number. Thus, although the counters have been reset, timestamps continue to be unique, enabling the event ordering subsystem to continuously and correctly order events. Clearly, when phase numbers are derived by some sort of phase counter, the problem of phase counter overflow or phase counter wrap-around arises, analogously to event counter overflow and wrap around, respectively. Nevertheless, this does not really cause a problem in the current context, since only a single phase counter value must be included in the timestamps, regardless of the number of processes or whether Lamport clocks, vector or matrix clocks are used. The disadvantage of this approach is that the transition from one event ordering phase to a subsequent one must be done in a controlled manner, causing additional control information to be exchanged during the phase transition among the involved processes. The concept of phase-based computation, as exploited by our approach, is well-known in the context of asynchronous distributed systems (distributed iteration [8], message stability [10], synchronizers [1], distributed discrete event simulation [6], global snapshots [2]).

Recently, Torres-Rojas and Ahamad presented an approach based on plausible clocks [11]. It is aimed at trading quality of reasoning among two events against timestamp size. When using a minimal size independent of the number of processes $n$, reasoning of the same quality as with Lamport’s clocks is supported. As the timestamp size increases, more and more reasoning becomes possible. Finally, when adopting a timestamp size of $O(n)$, plausible clocks allow reasoning identical to vector clocks. The approach presented in this paper is orthogonal to plausible clocks. Contrary to plausible clocks, we present an approach that reduces the timestamp size...
Protocol over the communication level such that the events of dis-
tributed computations are causally ordered at the process level. In the new
approach for reducing timestamp sizes is introduced in Section 3.

2.2 Model of Computation for Causal Order

Let $t$ be a data structure which is called timestamp of event $e_i$.
The timestamp is piggybacked to the transmitted message. If a
message is received then it depends on a delivery condition $\mathcal{C}$ when
the message will be delivered to the destination process $P_k$. $\mathcal{C}$ is
a boolean expression that, as soon as it becomes true, causes the
message to be delivered, otherwise, as long as $\mathcal{C}$ is false, a received
message is buffered but hidden from the destination process.

Figure 1. Causally ordered deliveries

Figure 1 illustrates an example of a computation that respects
causal order. Although $m_3$ is received before $m_1$, its delivery to
$P_3$ is delayed in order to respect causal order until $m_1$ has been
delivered to $P_3$.

2.3 Example: Implementing Causal Order through Matrix Clocks

For our purpose, matrix clocks count the number of messages sent
by each process to each other process. A process is assumed not
to send messages to itself.

Each process $P_i$ is endowed with a local matrix $sent_i[1..n, 1..n]$ of
integers (initialized to 0) where $n$ is the number of processes.

$sent_i[k, l] = \alpha \Leftrightarrow P_i$'s knowledge, $P_i$ has sent $\alpha$ messages to
$P_l$.

The matrix local to each application process represents the pro-
cess’s local view concerning the observation of messages ex-
changed among the processes during the distributed computation.

$sent_i[k, l]$ keeps track of the number of messages sent by $P_l$ and
already delivered to $P_i$.

A timestamp $t$ attached to any message consists of a copy of the
sender’s matrix $sent_i$. This information (the “past” of the message)
states the number of messages that causally precede the current
message on each channel. The comparison of this timestamp with
the number of already delivered messages to the destination
process $P_j$ (given by column $j$ of matrix $sent_j$) forms the delivery
condition. When the message is finally delivered, the destination
process updates its state view (local matrix $sent_j$) by merging it
with the timestamp carried by the message. Figures 2 and 3 show
how the timestamps and the additional data structures are exploited
for enforcing a causal ordering of events.

| 1: when $P_i$ sends $m$ to $P_j$ |
| 2: send $[m, sent_i]$ to $P_j$; |
| 3: $sent_j[i, j] := sent_j[i, j] + 1$; |
| 4: end |

Figure 2. Algorithm for sending of a message
The communication subsystem, which is in charge of causally or-
hold any message of the new phase when there are still message s
messages that any other process has sent within the ending ph ase.

necessary for providing every process with the precise numb er of
phase. The “flooding” of all channels with control messages i s
ter each process has broadcast its control messages for a par ticular
Thus, the creation and the sending of control messages cease s af-
subsequent control messages related to the same phase switc h.

When a phase switch becomes necessary, the process which cau ses
the phase switch (due to its attempt to send the

3.1 Basic Algorithm

In this section, we describe the basic algorithm of our approach, to
which we refer to as Factorized Timestamp Approach (FTA) since
the phase counter can actually be regarded as a higher order factor-
ization part of the original timestamps. For ease of description, we
call approaches not based on timestamp factorization Basic Time-
stamp Approaches (BTA).

As already stated, FTA is based on introducing a new phase of
computing whenever an integer entity of the clocks used for
timestamps (i.e., a Lamport clock or an arbitrary element of a vec-
tor or matrix clock), is about to overflow. At the end of the phase
switch (i.e., prior to the beginning of the new phase), all clocks are
reset to zero, whereas the phase counter is increased by one. It is
important to note that the value of the phase counter is part of any
timestamp attached to messages, thus, additional to the value of
the BTA timestamp. When the phase switch is completed then the
message that caused the phase switch – although actually initiated
at the end of the previous phase – is sent.

3.2 Switching phases

As already mentioned, successive phase switches define phases.
Within a single phase, each process can only send $2^b - 1$ messages
on a specific outgoing channel, with $b$ being the number of bits
allocated for the clock counter. If an additional message must be
sent, then this triggers a phase switch introducing a new phase.

When a phase switch becomes necessary, the process which causes
the phase switch (due to its attempt to send the $2^b$-th message via
a certain channel) broadcasts a special control message to all oth-
er processes. All other processes, upon delivery of this control mes-
also start broadcasting a control message to all other pro-
cesses, thereby entailing the delivery of necessary status informa-
tion for globally starting a new phase. Processes that already have
broadcast a control message do not re-broadcast when they receive
subsequent control messages related to the same phase switch.
Thus, the creation and the sending of control messages ceases af-
after each process has broadcast its control messages for a particular
phase. The “flooding” of all channels with control messages is
necessary for providing every process with the precise number of
messages that any other process has sent within the ending phase.
The communication subsystem, which is in charge of causally or-
dering message deliveries, needs this information in order to with-
hold any message of the new phase when there are still messages
of the old phase being transmitted but not yet delivered.

The problem of consistently performing a phase switch is related to
the problem of deriving a consistent global state. A solution to
the global state problem is given in [2]. It has later been adapted
for consistent checkpointing. Helary [3] presents an algorithm that
is closely related to our approach and actually serves as base for its
implementation. Contrary to Chandy and Lamport, who focused
their solution on a single “phase switch,” Helary presented an al-
gorithm which allows multiple phase switches (see Figures 4 and
5).

First, the initiating process sends a special control message to all
other processes (lines 2–3 of Figure 4). This message bears a
unique phase number so that it can be distinguished from former
or future phase switching-related messages. Once the initiating
process has sent the control messages, it increases a local phase
counter by one (line 4).

When a process receives a control message (line 1), it delays the
processing of the message if other causally related control mes-
sages are received (lines 2–3 of Figure 5). If no such messages
are received, the process starts the phase switch (line 4). If this
is the case then no actions are taken. Otherwise, the process re-
distributes the control message to all other processes (lines 5–6)
and subsequently advances the local phase counter (line 7).

FTA uses a combination of the generic algorithm given in Sec-
tion 2 and the phase switch algorithm of Helary given in Figures
4 and 5. Generally, all messages, i.e., “normal” event messages
and exceptional control messages, are tagged with an extended
timestamp consisting of the “basic” timestamp and additionally
the scalar value of the local phase counter.

From now on, we use “extended timestamp” for indicating the
extended version used by FTA. Basic timestamp (part) refers only
to a timestamp without a phase counter part. Basic timestamps
are used by BTA, i.e., approaches not exploiting phase switches.
3.3 Performance Optimization

Generally, a phase switch is initiated by a process which wants to send a message via a communication channel that would, without a phase switch, overflow. For minimizing the number of control messages sent through the communication infrastructure during a phase switch, we take advantage of the following observation: if a process has already received the maximum number of messages via a particular channel (the maximal number of each channel is known to all processes) then the destination process can implicitly assume that the sending process will in the near future – more precisely: while trying to send the next application message – initiate a phase switch. Vice versa, the sending process can assume that the destination process knows about the impossibility of receiving a further message from the particular sending process within the current phase. Consequently, in case the sending process actually initiates a phase switch, this process does not need to send a control message to this particular destination process since the information included in such a message is already implicitly available at the destination process. Figure 6 illustrates such a situation.

In the example, we assume that integers included in the basic timestamp are restricted to two bits. $P_1$ has already sent the maximal number of three messages to $P_3$. $P_2$ explicitly knows that no further message from $P_1$ will arrive prior to a phase switch message. Consequently, $P_2$, when issuing a phase switch through control messages, only needs to send a single control message to process $P_3$ but not to process $P_2$. Thus, one control message can be saved. The control message from $P_1$ to $P_3$, though, is necessary, since process $P_3$ cannot participate in the phase switch unless it knows that no further message of the old phase from $P_3$ is on its way to $P_2$. For instance, without such a control message, the communication subsystem of the site hosting $P_3$ cannot decide whether or not to deliver message $m_3$ of phase 3 to process $P_3$ since a causally related message $m_1$ belonging to the old phase 2 might exist.

3.4 The FTA Protocol

In this section, we describe the FTA protocol in detail for the causal ordering protocol of Section 2.3 based on matrix clocks. The protocol is given in Figure 7. As send events are non-blocking, a process can send messages within a phase although some messages of the previous phases have not yet been delivered. However, a process can deliver messages of phase $t'$ only when it has delivered all messages of phases $t' < t$. The key of the solution consists in managing two phase numbers $\text{phase}_1$ and $\text{del} \cdot \text{phase}_1$. The first one is used to timestamp messages (current sending phase), the second keeps the phase number of the messages that are currently being delivered (current delivery phase). Processes also manage two local arrays, a matrix $\text{sent}_i[1..n, 1..n]$ and a vector $\text{del}_i[1..n]$ of bounded counters ($0..\text{bound}$) (e.g., $\text{bound} = 2^b - 1$ if $b$ is the size of basic counters in bits). All data is initialized to 0. The matrix gives a view of message exchanges within the current sending phase, and the vector gives the number of messages which the process has delivered within the current delivery phase.

A new sending phase is initiated when a process cannot record a sending in $\text{sent}_i$ (a given $\text{sent}_i[i, x]$ overflowed). A new delivery phase is initiated when all the entries of $\text{del}_i$ reach the fixed bound. Each message $m$ carries its past represented by two fields: its sending phase number (integer $\text{m} \cdot \text{phase}$) and its past within this phase (matrix $\text{m} \cdot \text{send}$). The destination process $P_i$ can deliver a received message $m$ when it has already delivered all messages of previous phases ($\text{m} \cdot \text{phase} = \text{del} \cdot \text{phase}_i$) and all messages sent within the same phase but causally before this one ($\forall x \in 1..n, x \neq i; m \cdot \text{send}[x, y] \leq \text{del}_i[x]$). Three actions are performed prior to the delivery of a message:

1. The delivery of an application message increments the counter of the corresponding channel whereas the delivery of a control message “fills it up.”

2. If the sending phase of the message is equal to the current sending phase, the past of the message is integrated into $P_i$’s knowledge ($\text{sent}_i$). If some process has closed this delivery phase, $P_i$ has to do the same otherwise it updates $\text{sent}_i$.

3. If all messages of the current delivery phase have been delivered, a new delivery phase is initiated.

4 Proof

This section gives a formal proof of FTA’s correctness. Two properties must be proved: safety and liveness. All messages will be eventually delivered (liveness) according to the causal ordering constraint defined in Section 2 (safety).

Lemma 4.1 The factorized counters $(\text{phase}_i, \text{sent}_i[x, y])$ and $(\text{del} \cdot \text{phase}_i, \text{del}_i[x])$ exhibit monotonic behavior.

Proof: The successive values of each two-field counter does not decrease (the basic counter is the lower order part and the phase counter is the high order part of a factorized timestamp). Each time, $\text{sent}_i[x, y]$ or $\text{del}_i[x]$ are reset, the phase counter is incremented (the proof follows directly from lines 4,9,12,13, 17, 18, 22 and 23 of the protocol).

Lemma 4.2 Timestamps of causally ordered messages are monotonically increasing. Let $m_1$ and $m_2$ be two messages sent by $P_i$ and $P_j$ timestamped $(m_1 \cdot \text{phase}, m_1 \cdot \text{send})$ and $(m_2 \cdot \text{phase}, m_2 \cdot \text{send})$, respectively, such that $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_1$ sent to $P_k$. Then, it follows

$\forall x, y \in 1..n: (m_1 \cdot \text{phase}, m_1 \cdot \text{send}[x, y]) \leq (m_2 \cdot \text{phase}, m_2 \cdot \text{send}[x, y])$ and
When $P_i$ sends $m_1$ to $P_j$, begin
02: if $sent_i[i, j] = \text{bound}$ then new_sending_phase $\Rightarrow$ fi;
03: send ($m_1$, phase, sent) to $P_j$;
04: \{$sent_i[i, j] := sent_i[i, j] + 1$\} end

When $P_j$ receives any message ($m, m_\text{phase}, m_\text{sent}$) from $P_i$, begin
06: /* $m$ can be either an application message or a control message */
07: delay $m$ until ($m_\text{phase} = \text{del}_i$) \& \& ($\forall k \in 1..n, k \neq i : m_\text{sent}[k, i] \leq m_\text{del}[k]$)
09: if $m = \text{close}_i$ then del$_i[j] = \text{bound}$ else del$_i[j] := \text{del}_i[j] + 1$; fi;
10: if ($m_\text{phase} = \text{phase}_i$) then $\text{send}_i$ must be updated /*
11: if ($\text{del}_i[j] = \text{bound}$) then new_sending_phase
12: \{$forall x, y \in 1..n: sent_i[x, y] := max(sent_i[x, y], m_\text{sent}[x, y])$\};
13: $sent_i[i, j] := sent_i[i, j] + 1$ fi;
14: fi;
15: fi;
16: if ($\forall x \in 1..n: \text{del}_i[x] = \text{bound}$) then /* initiation of a new delivery phase */
17: $\text{del}_i[phase] := \text{del}_i[phase] + 1$;
18: forall $x \in 1..n: \text{del}_i[x] := 0$; 
19: fi;
20: if $m \neq \text{close}_i$ then deliver $m$ to $P_i$ fi end
21: Procedure new_sending_phase is begin
22: forall $x \in 1..n, x \neq i:$ if $sent_i[i, x] < \text{bound}$ then send ($\text{close}_i$, $\text{phase}_i$, $\text{sent}_i$) to $P_x$;
23: $\text{phase}_i := \text{phase}_i + 1$;
24: forall $x, y \in 1..n: sent_i[x, y] := 0$; end

Theorem 4.3 Delivery events respect causal order.

Proof: Causal order is never violated if – whenever a process $P_i$ is the destination process of two messages $m_1$ and $m_2$ with $m_1 \rightarrow m_2$ such that $m_2$ has arrived and is deliverable – then $m_1$ has also arrived at $P_i$, and has been already delivered.

\begin{align*}
(m_1, \text{phase}, m_1, \text{sent}[i, k]) &< (m_2, \text{phase}, m_2, \text{sent}[i, k]) \\
\end{align*}

\begin{prooftree}
[\exists m' : send(m_1) \rightarrow send(m') \land send(m') \rightarrow send(m_1)]
\end{prooftree}

\begin{itemize}
\item $r > 0 \ (\exists m': \text{send}(m_1) \rightarrow \text{send}(m') \land \text{send}(m') \rightarrow \text{send}(m_1))$ by the induction hypothesis, Lemma 4.2 holds for $m_1$ (sent by $P_i$ to $P_j$) and $m'$ and, by the basic case, $\forall x, y \in 1..n, (m_1, \text{phase}, m_1, \text{sent}[x, y]) \leq (m_2, \text{phase}, m_2, \text{sent}[x, y])$.
\end{itemize}

The relation that exists between the timestamps of $m_1$ and $m'$ (given by $i$) is extended to the pair $m_1$ and $m_2$ (by $ii$), proving Lemma 4.2. \qed

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{matrix_clocks.png}
\caption{The FTA Protocol for Matrix Clocks}
\end{figure}
by lines 16, 17 and 18 which means that \( b = \text{bound} \). As 
\((a, \text{bound}) \leq (m_1, \text{phase}, m_1, \text{sent}[j, i]) \) and \( m_1, \text{sent}[j, i] < \text{bound} \) (line 2 of the protocol), we have necessarily \( a < m_1, \text{phase} \). Lines 16, 17 and 18 incur \((a', b') = (a + 1, 0) \) \( \leq (m_1, \text{phase}, m_1, \text{sent}[j, i]) \) contradicting the hypothesis \((m_1, \text{phase}, m_1, \text{sent}[j, i]) < (a', b') \).

2. If \( m \) is sent by \( P_j \), three cases must considered:

\begin{itemize}
  \item \( a < m, \text{phase} \): whatever message \( m \) might be, \((a', b') = (a + 1, 0) \)\)
  \item \( a = m, \text{phase} \) and \( b < m, \text{sent}[j, i] \): If \( m \) is an application message, it cannot cause a phase shift as \( b < m, \text{sent}[j, i] < \text{bound} \). If \( m \) is a close, message then there may not exist a message \( m_1 \) sent within the same phase after \( m \) contradicting \( a = m, \text{phase} \). Similarly to the previous case, any other message can cause at most the shift from \((a, b)\) to \((a + 1, 0)\) \( \leq (m, \text{phase}, m, \text{sent}[j, i]) \).
  \item \( a = m, \text{phase} \) and \( b = m, \text{sent}[j, i] \): \( m \) and \( m_1 \) represent the same message by Lemma 4.2 and consequently this means that \( m_1 \) has been already delivered.
\end{itemize}

\[ \square \]

**Theorem 4.4** All sent messages will eventually be delivered.

**Proof:** As channels are reliable all messages will arrive at their destination by finite time. Let \( m_s \) (sent by process \( P_j \)) be one of the smallest (according to the acyclic relation \( \rightarrow \)) of the messages that have been sent and not yet delivered to \( P_j \). Such a message verifies \( \forall m \to m, A m \) not yet delivered \( = \emptyset \) (there can exist more than one such message as \( \rightarrow \) is a partial order). The liveness property will be proved by induction on the phase number.

- **Basic case** \((m, \text{phase} = \text{del}, \text{phase}) = 0\): If \( m_s \) sent by some \( P_j \), cannot be delivered then \( \exists k \neq i \in 1..n : m_s, \text{sent}[k, i] > \text{del}[k] \). Before sending \( m_s \), \( P_j \) was aware about \( P_i \) sent \( a = m_s, \text{sent}[k, i] \) application messages to \( P_i \). All these messages precede \( m_s \) causally and, thus, have been already delivered (by definition of \( m_i \)). The delivery of “a” messages, sent by \( P_k \) within the same phase leads to the execution line 8 of the protocol “a” times and thus \( \text{del}[k] = a = m_s, \text{sent}[k, i] \) contradicting the hypothesis.

- **Induction case** \((m, \text{phase} > 0)\): Before sending any message within phase \( m, \text{phase} \), \( P_j \) has closed the previous phase \( p = m, \text{phase} - 1 \) (it has executed procedure \text{new},\text{sending},\text{phase}). This means that on each outgoing channel, either \( P_j \) sent during the phase \( p \) a number of messages equal to \text{bound} or it has sent a control message. In both cases they have been delivered, by the induction hypothesis, as they were sent in phase \( p = m, \text{phase} - 1 \). Their delivery to each \( P_i \), led to \( \text{del}[i] = \text{bound} \) (line 8). Then (line 9), either the receiving process has already switched to a new phase \(( \text{phase} > p \) or it has not done so and thus it must do so as the test of line 10 holds. This flooding is done by each process until every process did the shift to phase \( p + 1 \). Eventually all these messages will be delivered and thus lines 15, 16 and 17 will be executed leading to the initiation of delivery phase number \( m, \text{phase} \).

If \( m_s \), sent by some \( P_j \), cannot be delivered then:

1. \( m, \text{phase} = \text{del}, \text{phase} \) and \( \exists k \neq i \in 1..n : m_s, \text{sent}[k, i] > \text{del}[k] \). For the same reasons then the basic case, this cannot occur.
2. \( m, \text{phase} > \text{del}, \text{phase} \). As explained above, the initiation of the phase \( m, \text{phase} \) by at least one process (the sender of \( m_s \) did) incurs a flow of control messages (that causally precede \( m_s \)) that trigger each process to switch to a delivery phase equal to \( m, \text{phase} \). By hypothesis all messages that precede \( m_s \) have been delivered and thus \( m, \text{phase} = \text{del}, \text{phase} \). \( \square \)

### 5 Analysis

In this section, we analyze timestamp sizes needed for enforcing causal order with FTA and BTA and identify the overall amount of control overhead that is transmitted through the communication channels for both approaches under worst case assumptions.

#### 5.1 Timestamp Size

**Definition 3 (Timestamp Size)** Let \( t \) be a timestamp composed of \( m \) positive integers, \( t[k], 0 \leq k \leq m - 1 \), indicates the \( k \)-th integer of timestamp \( t \). Furthermore, let \( N_s = 2^{b_s} - 1 \) (\( b_s \) integer) be the maximal number that can be presented by \( t[k] \). Then, the size of integer \( t[k] \) is \( b_s \). It is indicated by \( |t[k]| \). The timestamp size \( ||t|| \) of \( t \) is defined as follows \( ||t|| := \sum b_{s}^{-1} |t[k]| \).

The above definition regards a timestamp as a series of integers. Clearly, timestamps need not necessarily to be represented as such. Furthermore, potential redundancy inherent in a specific timestamp can be reduced be through the use of, e.g., compression techniques prior to piggybacking the timestamp and then transmitting the message. In the context of this analysis, we do not consider those techniques since they are not problem specific. They rather present orthogonal techniques that can additionally be exploited.

**Basic Timestamp Size**

In a distributed computation with \( n \) processes and equal integer sizes for all integers, the size of timestamps is \( ||t^{\text{scal}}|| = ||t^{\text{vec}}|| = ||t^{\text{mat}}|| = n \times |t^{\text{scal}}[1]| \) (\( t^{\text{scal}} \) being an arbitrary entry of the timestamp) for scalar, vector, and matrix timestamps, respectively.

The results are very intuitive: scalar timestamp size only depends on the integer size, regardless of the number of processes. Vector timestamps naturally exhibit a linear dependence between their size and the number of processes as well as a linear dependence between timestamp size and integer size. Matrix timestamps, finally, show a linear relation between timestamp size and integer size but a quadratic growth in timestamp size if the number of processes increases. The more interesting observation is the following one. Almost all published work on causal ordering mechanisms assumes theoretically unbounded integers for timestamps, thus ignoring the real-world problem that arises if one is actually interested in an operable implementation, since unbounded integers simply cannot be realized.
An argument for its justification states that the integer size can be chosen such that counters practically do not overflow or wrap-around (using 32 or even 64 bits). Although we agree to this argument, it must be recognized that this approach leads to a substantial amount of piggybacked information. For example, using matrix timestamps in a distributed computation involving not more than 50 processes, a counter size of 32 results in a timestamp size of 10 K-byte.

**Extended Timestamp Size**

As a convention, we assume that the first integer \( t[0] \) of an extended timestamp \( t \) indicates the phase number. The timestamp size for an extended timestamp \( t_n^{(x,y,z,m)} \) is \( || t_n^{(x,y,z,m)} || = || t_n^{(x,y)} || = || t_n^{(x)} || + || t^{(y,z)} || + || t^{(z,m)} || \), and \( || t_n^{(x,y)} || = || t_n^{(x)} || + n \cdot || t^{(y,z)} || + n^2 \cdot || t^{(z,m)} || \) (\( i \neq 0 \) being an arbitrary entry of the timestamp) for scalar, vector, and matrix timestamps, respectively.

As the phase number is independent from the number of processes, FTA, in general, uses smaller timestamps compared to BTA when using vector or matrix clocks.

### 5.2 Control Overhead

**Definition 4 (Control Overhead)** Let \( DC \) be as distributed computation involving \( n \) processes that performs on top of a causal ordering communication subsystem \( S \). The sum of the sizes of all timestamps that are transmitted through the communication channels when executing \( DC \) is called control overhead (of \( DC \)). It is indicated by \( \mathcal{CO}_{DC} \). The highest possible control overhead of \( DC \) is called worst case control overhead (of \( DC \)), indicated by \( \mathcal{CO}^w_{DC} \).

The control overhead of a particular distributed computation generally behaves non-deterministic in an asynchronous distributed system. Thus, when executing the same computation, the control overhead is likely to vary from execution to execution. This is in contrast to the worst case control overhead. Here, certain worst case assumption can be made. When using sufficiently large basic timestamps, non-determinism with respect to control overhead as defined in this paper does not occur, since every message that is issued by the distributed computation results in a single message to be transmitted through the communication channels. Thus, in this case, worst case control overhead and control overhead are always identical. When using FTA then additional control messages occur. In such a case, the distribution of send events among the processes is crucial to the number of control messages that must be transmitted. For example, as discussed prior, when all processes use all their channels in an homogeneous manner, no control messages are needed at all since the counters of the channels reach their limit at the same time. This leads to the conclusion that FTA generally creates less control overhead than worst case control overhead.

The identification of worst case control overhead of an arbitrary distributed computation is not trivial. In this paper, we make the following assumption for simplifying this task: since for the scope of the analysis we are interested in neither the nature nor the results of the computation but merely in the number of messages generated by it and sent through the communication channels, a worst case scenario can easily be created when sending all messages that arise through a single channel. Thus, this channel will be used as heavily as possible. This has the following two consequences with regard to the timestamps: 1) integer sizes must be at least as large as required to contain the maximal number of messages when using BTA, and 2) when FTA is used, then the size of the phase counter must be at least as large as required for holding the maximal phase number.

#### Worst Case Control Overhead of FTA

Let \( w \) be the number of application messages created by a distributed computation \( DC \) involving \( n \) processes running on top of a causal-ordering communication subsystem based on BTA. The worst case control overhead is then given by \( \mathcal{CO}^w_{DC} = w \cdot || t_n^{(v)} || \) (\( i \) being either \( s \), \( v \), or \( m \)).

#### Worst Case Control Overhead of BTA

If the subsystem is based on FTA then the worst case control overhead is \( \mathcal{CO}^w_{F TA} = \left[ w/2^{h^{(v)}} \right] \cdot n(n-1) \cdot || t_n^{(v)} || + w \cdot || t_n^{(s)} || \). The first part of the latter formula presents the worst case control overhead due to \( \left[ w/2^{h^{(v)}} \right] \) phase switches. During a phase switch, maximal \( n(n-1) \) control messages of length \( || t_n^{(v)} || \) must additionally be sent. The second part gives the amount of control information piggybacked due to the \( w \) application messages. In Figure 8, we present examples of worst case control overhead.

<table>
<thead>
<tr>
<th>Worst case control overhead (10 processes sending in total 10 million messages)</th>
<th>FTA</th>
<th>BTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_n^{(x,y,z,m)} )</td>
<td>scalar</td>
<td>150 · 10^3</td>
</tr>
<tr>
<td></td>
<td>vector</td>
<td>397 · 10^7</td>
</tr>
<tr>
<td></td>
<td>matrix</td>
<td>2782 · 10^7</td>
</tr>
</tbody>
</table>

**Figure 8. Worst case control overhead of FTA vs. BTA**

The distributed computation is assumed to send 10 million messages in total involving 10 processes. For BTA the minimal integer size of \( 24 = \left[ \log_2 10^7 \right] \) for holding a value of 10 million is used. For FTA, we consider integer sizes of 4, 8 or 16 for the counters of the basic timestamp part and integer sizes of 20, 16 or 8, respectively, for the phase counter size (also called phase size for short).

When vector or matrix timestamps together with a particular timestamp size are adopted then the worst case control overhead can be minimized for FTA by identifying a suitable partition of the extended timestamp size into phase size and basic timestamp size. This optimization problem can be formally stated by using the previous formulae. Unfortunately, due to transcendental functions, no closed solution to the optimization problem can be given. Nevertheless, because the problem is of discrete nature, only \( c \) possibilities exist. These possibilities can easily be enumerated leading to an efficient identification of a minimum.
6 Simulation

In this section, we present simulation results for the control overhead that arise during a distributed computation using FTA or BTA. We developed a simulator with which we investigated the behavior of FTA and BTA in terms of control overhead. The simulator parameters permit to set the number of processes for a particular distributed computation and to vary the arrival rate of send message events through the directed communication channels that connect the processes. The distribution of the time between two subsequent send message events is assumed to be of exponential nature. The message transfer (time) delay (for both, application and control messages) can be adjusted by specifying the parameters of a normal distribution. Additional parameters are the integer size of the basic timestamps (or timestamp part) and the size of the extended timestamp. Finally, the maximal simulation time can be given. The number of messages sent via each channel can be indirectly specified since it depends on the arrival rate of send message events of a channel and the overall simulation time.

We performed a large number of different simulation using various parameter settings. We assumed distributed computations involving 10, 20, 30, and 40 processes with an event ordering subsystem exploiting matrix clocks for BTA and FTA. In all eight scenarios, we set the simulation time and the sent event arrival rate in such a way that approximately 10 million application messages were sent. The results, depending on the partitioning of an extended timestamp into phase size and integer size (for FTA) are given in Figure 9. The particular setting of the mean transfer delay and the number of processes turned out to hardly influence the ratio FTA control overhead versus BTA control overhead. The overall outcome of the simulation is that choosing FTA over BTA always leads to substantial savings in control overhead regardless of the partitioning of the extended timestamp into phase counter and basic timestamp part.

![Figure 9. Simulation results of FTA vs. BTA](image)

7 Conclusion

Enforcing causal order among events of a distributed computation is a challenging task. Many solutions to this problem known from literature are based on the concept of timestamps. Thus, depending on the number of participating processes and the timestamp structure, substantial bandwidth requirements arise since the timestamps must be transmitted via the communication subsystem. This paper presented a novel technique to reduce the storage and bandwidth requirements of timestamps, we call Factorized Timestamp Approach (FTA). It is based on including a factorized age and bandwidth requirements of timestamps, we call Factorized Timestamp Approach (FTA). We developed a simulator with which we investigated the behavior of FTA and BTA in terms of control overhead.

Due to lack of space, some results could not be presented in this paper. They are given in [7].

References