Internal Report TCS-95-7
October, 1995

Implementing Causal Ordering with Bounded Time-stamps

Saileshwar Krishnamurthy[1] and Madhavan Mukund[2]

School of Mathematics
SPIC Science Foundation
92 G.N. Chetty Road, T. Nagar
Madras 600 017, INDIA

Abstract

This paper investigates a solution to the problem of causal ordering in message-passing distributed systems. Causal ordering is the restriction that messages are delivered in a *fifo* fashion with respect to the global causal order between events in the system. This is stronger than the condition that each local channel is *fifo*.

In our algorithm, causal ordering is implemented by having each process maintain, as the computation proceeds, its latest information about every other process in the system. To achieve this, messages are tagged with time-stamps. The novel feature of the protocol described here is that it allows for the reuse of time-stamps. Under certain conditions, this permits an implementation of causal ordering using time-stamps that are uniformly bounded.

Keywords: Distributed algorithms, causal ordering, bounded time-stamps

[1]Work done while visiting the School of Mathematics, SPIC Science Foundation as part of program requirements of the Birla Institute of Technology and Science, Pilani 333 031, India. This author's current address is: The Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398. E-mail: krish@cs.purdue.edu

[2]E-mail: madhavan@ssf.ernet.in

Introduction

With the advent of fast, personal workstations, the shape of computing is changing. Instead of clustering computer systems around powerful central mainframes, connecting individual computers in a distributed network provides a more flexible and effective method of utilising the resources at hand.

Programming distributed systems to achieve a joint goal requires a mechanism for coordinating the activities of the individual components. This coordination can be achieved in many ways: for instance, individual processes could write information of global interest into a shared memory. Alternatively, processes could synchronise periodically and exchange information about each other.

However, when the individual components in the network are spatially separated, the only practical method for exchanging information is by sending and receiving messages over channels which connect pairs of processes. Such message-passing communication is usually described as being *asynchronous* since, in principle, there may be an arbitrary delay between the sending of a message and its receipt at the other end.

Communication using message-passing introduces a high degree of non-determinism in the behaviour of the overall system. If the communication channel is unreliable, messages may be lost in transit. Even if all messages sent do reach their recipients, they may arrive in an order which is different from the one in which they were sent.

Normally, two assumptions are made about the underlying system to eliminate some of this non-determinism and help make programming these systems more tractable. The first assumption is that the channels are reliable—that is, every message sent is eventually delivered and no spurious messages are generated by the channels. The second assumption is that individual channels function in a first-in first-out (*fifo*) fashion. In other words, between each pair of processes, messages are received in the same order in which they were sent.

These assumptions do not rule out a global disruption in the order of delivery of messages. This could prove problematic in some settings. Consider a distributed database with replicated copies of data. Updates to a data item will have to be propagated to each process holding a copy of the data. All these processes should receive the updates in the same order to ensure global consistency. For instance, suppose we have a system with three processes, p, q, r where q and r both hold copies of a data item x which p wants to update. The update is passed on by p to r via a message M_1. Then, p informs q of the change via a message M_2. Based on the update contained in M_2, q makes a further change to x and passes this information on to r in a message M_3. Since M_1 and M_3 reach r on different channels, it could happen that M_3 is delivered to r before M_1, even though each individual channel functions in a fifo manner. So, r will update x in the wrong order.

This kind of global non-determinism can be eliminated using an abstraction known as *causal ordering*, first proposed in the Isis system [1]. Causal ordering is the property that the delivery of messages across the system respects the *global* partial order between events in the system—we shall make this definition more precise in the next section. In the Isis system, causal ordering is implemented by passing along the *entire* message history with each transmission. This is clearly impractical in the long run.

A much simpler protocol was proposed in [6]. The idea is to distinguish between the receipt of a message at a process and its delivery. Each message is tagged with a set of vector time-stamps [3]. When a message is received, the recipient first examines these time-stamps and determines whether delivering the message immediately would violate causal ordering. If there is no problem with causal ordering, the message is delivered. Otherwise, delivery is postponed and the message is kept aside to be considered again for delivery when the next message is received at that process.

This algorithm was further simplified in [5]. The protocol in [5] involves passing an $N \times N$ matrix of time-stamps with each message, where N is the number of processes in the system. The time-stamp at position $\langle i, j \rangle$ in the matrix corresponds to the number of messages which have been sent so far from process i to process j. Though this scheme is straightforward and effective, it has a major drawback—the entries in the matrix grow without bound as the computation progresses.

In this paper, we describe a modification of the protocol of [5] in which time-stamps can be reused. Our scheme relies on keeping track of the latest information passed between processes, using a simplified version of the algorithm proposed in [4]. This information is used to collect acknowledgements by determining which messages have reached their recipients. Using these acknowledgements, a process can decide when it is safe to reuse a time-stamp.

Though the scheme we propose is slightly more complicated than that of [5], the overhead associated with passing on time-stamp information with each message is essentially the same. Our protocol also requires only $O(N^2)$ extra values to be transmitted with each message. Further, in certain cases we can ensure that only a *bounded* set of time-stamps are used at any given time. This permits us to implement causal ordering with a *uniform* upper bound on the size of the time-stamping information associated with each message.

As mentioned earlier, causal ordering is a strengthening of the assumption that individual channels behave in a fifo manner. However, our algorithm, like those of [5, 6], actually implements causal ordering in *any* message-passing system with reliable channels. In other words, we do not need to assume that channels are fifo.

The paper is organised as follows. In the next section, we introduce our model of computation and formally define the problem. In Section 2, we describe a simple protocol by which processes can maintain up-to-date information about each other. The next section describes an implementation of causal ordering based on the protocol of Section 2. In Section 4, we refine our protocol for causal ordering by adding structure to the time-stamps. This structure is exploited in the next section to reuse time-stamps and obtain, in certain cases, a protocol for causal ordering which uses only a bounded set of time-stamps. We conclude with a discussion on directions for future work.

1 Preliminaries

The Model

We consider a message-passing model of distributed systems for our algorithms. A message-passing system consists of a finite set of processes and a finite set of channels. Channels are assumed to have infinite buffers and to be error-free. However no restriction is made on the order in which messages are delivered. The delay experienced by a message in a channel is arbitrary but finite.

Events on each process can be classified as *send*, *deliver* and *internal*. A *send* event corresponds to the insertion of a message into a channel, and a *deliver* event to the removal of a message from a channel. An *internal* event refers to events that occur entirely *within* a process and affect no other process in the system. In this paper however, we do not consider *internal* events. In fact all the protocols that we propose are to be superposed on an underlying computation and *internal* events are considered part of the underlying computation.

Some of the protocols in this paper make a distinction between the receipt of a message and its delivery. In case a message cannot be delivered according to the protocol, we assume that the message is preserved by the receiving process. In these protocols, every time a new message is received and delivered, an attempt is made to deliver any old messages that have been received and are as yet undelivered.

Events and ordering

We consider a system with a set of processes $\mathcal{P}$. Let N denote the number of processes in $\mathcal{P}$. We use the symbols p, q, r, s and t to denote specific processes in the system. We denote a message M sent from a process p to a process q by $M : p \Rightarrow q$. With each such message, we associate two distinct events, $send(M)$ and $deliver(M)$. If $M : p \Rightarrow q$, we call the event $send(M)$ a p-event and the event $deliver(M)$ a q-event.

Definition 1.1 *We define the relation "predecessor of", denoted $\sqsubset$, between events in our system as follows: Let e_1 and e_2 be two events. Then $e_1 \sqsubset e_2$ if and only if one of the following two conditions is true:*

1. *There is a process p such that e_1 and e_2 are both p-events and e_1 immediately precedes e_2 in p. In other words, there is no other p-event e_3 which occurs after e_1 and before e_2.*

2. *Corresponding to some message $M : p \Rightarrow q$, e_1 is the p-event $send(M)$ and e_2 the q-event $deliver(M)$.*

Notice that $\sqsubseteq^*$, the reflexive and transitive closure of $\sqsubset$, defines a *total order* on all the events that occur on a particular process.

We also define Lamport's "happened before" relation [2] as the transitive closure $\sqsubset^+$ of the relation $\sqsubset$. Since we shall use this relation often, we use a separate symbol for it: we denote the "happened before" relation by $\rightarrow$. A situation in which $e_1 \rightarrow e_2$ does not hold is denoted by $e_1 \not\rightarrow e_2$. Notice that $e_1 \not\rightarrow e_2$ does not imply that $e_2 \rightarrow e_1$. In fact, it is easy to see that $\rightarrow$ defines a *strict partial order* on the events of the system. For convenience, we abbreviate $send(M) \rightarrow send(M')$ by $M \rightarrow M'$.

Let E be a finite set of events which contains at least one p-event. Since all p-events are totally ordered by $\sqsubseteq^*$, the maximum p-event in E, denoted by $max_p(E)$, is well defined. If there is *no* p-event in E then, by convention, $max_p(E)$ is defined to be the special event *null*. So for any event e in the system, we assume that $null \sqsubseteq^* e$.

In a message-passing system, *causal ordering* is the property that the order of delivery of messages in the system respects the order of despatch of messages. This corresponds to a global *fifo* requirement on message delivery with respect to the partial order $\rightarrow$. More precisely, we have the following definition.

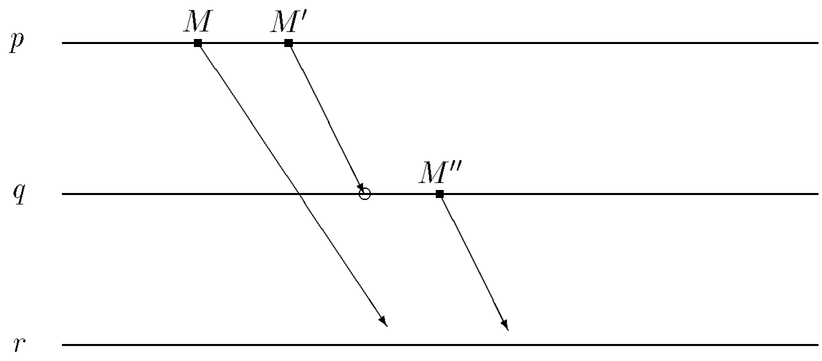


Figure 1: Causal ordering is respected if M is delivered before M''

Definition 1.2 *A computation of a message-passing system is* causally ordered *if for any two messages M_1 and M_2 where $M_1 \rightarrow M_2$—that is, $send(M_1) \rightarrow send(M_2)$—it is the case that $deliver(M_2) \not\rightarrow deliver(M_1)$.*

The example in Figure 1 demonstrates the causal ordering property. Causal ordering is violated if M'' is delivered before M.

One of our goals is to design a protocol that can be superposed on an underlying computation to ensure that it is causally ordered. Our protocols involve adding extra control information to every message that is sent in the system. This is used to record information about the messages sent by other processes. Thus, every time a message is delivered, the process which receives the message gets fresh information about the rest of the system.

Gossip

In a message-passing system, the only way a process can obtain information about other processes is through the messages it receives. In other words, at any point in the computation, the only events in the system "known" to a process are those that have happened earlier than the current event, according to the partial order $\sqsubseteq^*$. This is formalised in the following definition.

Definition 1.3 *Let p be a process and e_p be a p-event. The information that p has about the rest of the system after the occurrence of e_p is the set $e_p{\downarrow} = \{e' \mid e' \sqsubseteq^* e_p\}$.*

In particular, we are interested in the *latest* information that p has about the rest of the system after e_p. This corresponds to recording, for every other process q, the maximum q-event that p can "see" in $e_p{\downarrow}$. In certain cases, we will also be interested in keeping track of other processes' latest information—for instance, we want to record the most recent information p has regarding q's latest information about r. These notions are formally captured by primary and secondary information.

Definition 1.4 *The* primary information *a process p has about a process q after a p-event e_p—denoted $latest_{p \leftarrow q}(e_p)$—is defined to be $max_q(e_p{\downarrow})$. If there are no q-events in $e_p{\downarrow}$, then $latest_{p \leftarrow q}(e_p)$ is set to null.*

The secondary information *a process p has about a process r through a process q after the occurrence of a p-event e_p—denoted $latest_{p \leftarrow q \leftarrow r}(e_p)$—is defined to be the event $latest_{q \leftarrow r}(latest_{p \leftarrow q}(e_p))$.*

The primary and secondary information of p after e_p is collectively called its gossip.

Let p, q and r be processes, e_p a p-event and e_q a q-event. Since the latest information p has about r at e_p, $latest_{p \leftarrow r}(e_p)$ and the latest information q has about r at e_q, $latest_{q \leftarrow r}(e_q)$, both refer to r-events, this information can always be compared—recall that all r-events are totally ordered by $\sqsubseteq^*$. We say that p at e_p has later information about r than q at e_q if $latest_{q \leftarrow r}(e_q) \sqsubseteq^* latest_{p \leftarrow r}(e_p)$.

Notice that $latest_{p \leftarrow p \leftarrow q}(e_p) = latest_{p \leftarrow q}(e_p)$, for every $p, q \in \mathcal{P}$ in the system. That is, the primary information is implicitly available in the secondary information.

The first step in describing our protocol for enforcing causal ordering is to design a scheme whereby processes update their gossip in a consistent manner.

2 Updating gossip: A naive solution

We associate with each process $p \in \mathcal{P}$ a local clock T_p which is incremented with the occurrence of every event in p. For any event e occurring in the process p, $T_p(e)$ denotes the value assigned to T_p when e occurs. We associate the clock value 0 with the special event *null*. So, $T_p(null) = 0$ for every process $p \in \mathcal{P}$. Notice that if e_1 and e_2 are two p-events such that $e_1 \to e_2$, $T_p(e_1) < T_p(e_2)$. Our protocol will ensure that every message M sent by a process p is tagged (i.e., time-stamped) with the value that is assigned to T_p when $send(M)$ occurs. We use $T_p(M)$ to abbreviate $T_p(send(M))$ for a message M.

To keep track of the gossip in the system, each process p maintains GOSSIP_p, an associative $N \times N$ array of non-negative integers indexed by $\mathcal{P} \times \mathcal{P}$. Each entry in GOSSIP_p represents the time-stamp of an event. $\mathsf{GOSSIP}_p[q, r]$ is interpreted as the latest information that q knows about r about which p is aware. In other words, after a p-event e_p, the entry $\mathsf{GOSSIP}_p[q, r]$ should correspond to the time-stamp assigned by T_r to the event $latest_{p \leftarrow q \leftarrow r}(e_p)$. Initially, $\mathsf{GOSSIP}_p[q, r] = 0$ for every $q, r \in \mathcal{P}$.

To maintain up-to-date gossip information in GOSSIP_p, each process p obeys a simple protocol. With each message it sends, it encloses the time-stamps corresponding to its current gossip. When a process receives a message, it examines the gossip time-stamps attached to the message and updates its local time-stamps whenever the sending process has more recent information.

Protocol 1 *A naive implementation of Gossip*

1. *Sending a message $M : p \Rightarrow q$*
   - *Set* $\mathsf{GOSSIP}_p[p, p] := T_p(M)$.
   - *Let GP_M be an associative $N \times N$ array of non-negative integers indexed by $\mathcal{P} \times \mathcal{P}$. Copy* GOSSIP_p *into GP_M.*
   - *The actual message sent by p consists of the triple $\langle M, T_p(M), GP_M \rangle$.*
2. *Receiving a message $M : p \Rightarrow q$*

- *Compute the function best* $: \mathcal{P} \rightarrow \{p, q\}$ *as follows:*

$$\forall r \in \mathcal{P},\ best(r) = \begin{cases} p & \text{if } GP_M[p,r] > \mathsf{GOSSIP}_q[q,r] \\ q & \text{otherwise} \end{cases}$$

- *Update* GOSSIP_q *as follows:*
  For each $r \in \mathcal{P}$ *such that* $best(r) = p$*:*

$$\begin{aligned} \mathsf{GOSSIP}_q[q,r] &:= GP_M[p,r]. \\ \forall t \in \mathcal{P},\ \mathsf{GOSSIP}_q[r,t] &:= GP_M[r,t]. \end{aligned}$$

The goal of Protocol 1 is to ensure that GOSSIP_p represents the secondary information of p, for every process $p \in \mathcal{P}$.

Theorem 2.1 *Let* e_p *be a* p*-event for* $p \in \mathcal{P}$*. Then for every* q*,* $r \in \mathcal{P}$ *the following holds immediately after* e_p *occurs:* $\mathsf{GOSSIP}_p[q,r] = T_r(latest_{p \leftarrow q \leftarrow r}(e_p))$*.*

Proof The proof is by induction on k, the size of $e_p\!\downarrow$.

1. *Base case* $(k = 1)$

   Clearly e_p is the only event in $e_p\!\downarrow$, and so e_p cannot be a *deliver* event—otherwise, the corresponding *send* event would also be in $e_p\!\downarrow$. Since e_p is a *send* event, Protocol 1 ensures that $\mathsf{GOSSIP}_p[p,p] = T_p(M)$ and $\mathsf{GOSSIP}_p[q,r] = 0$, for all pairs $(q,r) \neq (p,p)$. The theorem is trivially true in this case.

2. *Induction step* $(k > 1)$

   By the induction hypothesis, immediately after any r-event e_r such that $e_r \neq e_p$ and $e_r \in e_p\!\downarrow$, we have $\mathsf{GOSSIP}_r[s,t] = T_t(latest_{r \leftarrow s \leftarrow t}(e_r))$.

   If e_p is a *send* event, there must exist a p-event e'_p (where $e'_p \not\equiv null$) such that $e'_p \sqsubset e_p$—that is, e'_p immediately precedes e_p. (If not, there exists an event $e' \sqsubset e_p$ because $|e_p\!\downarrow| > 1$, and from Definition 1.1, since e' is not a p-event, e_p is a *deliver* event, which is false). The theorem is true for e'_p by the induction hypothesis, and Protocol 1 ensures that GOSSIP_p remains unchanged for all entries except $\mathsf{GOSSIP}_p[p,p]$ after the occurrence of e_p. So the theorem is true for e_p as well, since $latest_{p \leftarrow q \leftarrow r}(e'_p) = latest_{p \leftarrow q \leftarrow r}(e_p)$, for every pair of processes $(q,r) \neq (p,p)$.

   On the other hand, let e_p be the event $deliver(M)$ corresponding to a message $M : s \Rightarrow p$ and let e_s be the corresponding event $send(M)$. Clearly, there exists a p-event e'_p (where e'_p could be *null*) such that $e'_p \sqsubset e_p$. So $e_s \sqsubset e_p$ and $e'_p \sqsubset e_p$. Also, the theorem is true for e'_p, $e_s \in e_p\!\downarrow$ by the induction hypothesis. That is, $GP_M[q,r] = \mathsf{GOSSIP}_s[q,r] = T_r(latest_{s \leftarrow q \leftarrow r}(e_s))$, for every q, $r \in \mathcal{P}$ and $\mathsf{GOSSIP}_p[q,r] = T_r(latest_{p \leftarrow q \leftarrow r}(e'_p))$, for every q, $r \in \mathcal{P}$.

   When M is received, Protocol 1 computes $best(r) = s$, for every $r \in \mathcal{P}$ such that $GP_M[s,r] > \mathsf{GOSSIP}_p[p,r]$. By the induction hypothesis this is equivalent to computing $best(r) = s$, for every $r \in \mathcal{P}$ such that $latest_{p \leftarrow r}(e'_p) \sqsubseteq^* latest_{s \leftarrow r}(e_s)$.

   If there exists $r \in \mathcal{P}$ such that $best(r) = s$, it is easy to see that after e_p occurs, the following hold: $latest_{p \leftarrow p \leftarrow r}(e_p) = latest_{s \leftarrow s \leftarrow r}(e_s)$ and, for every $t \in \mathcal{P}$, $latest_{p \leftarrow r \leftarrow t}(e_p) = latest_{s \leftarrow r \leftarrow t}(e_s)$. Since e_s satisfies the induction hypothesis,

$latest_{s \leftarrow r \leftarrow t}(e_s) = GP_M[r,t]$ and $latest_{s \leftarrow s \leftarrow r}(e_s) = GP_M[s,r]$. However, Protocol 1 performs exactly these assignments while updating GOSSIP_p.

$\square$

Proposition 2.2 *Let p, q, $r \in \mathcal{P}$ and let GOSSIP_p and GOSSIP_q represent the arrays maintained by p and q after the events e_p (a p-event) and e_q (a q-event) respectively. Protocol 1 ensures that if p after e_p has later gossip about r than q after e_q then $\mathsf{GOSSIP}_p[p,r] \geq \mathsf{GOSSIP}_q[q,r]$.*

Proof If $latest_{q \leftarrow r}(e_q) \sqsubseteq^* latest_{p \leftarrow r}(e_p)$ then $T_r(latest_{p \leftarrow p \leftarrow r}(e_p)) \geq T_r(latest_{q \leftarrow q \leftarrow r}(e_q))$. From the previous theorem, it then follows that $\mathsf{GOSSIP}_p[p,r] \geq \mathsf{GOSSIP}_q[q,r]$. $\square$

3 Gossip and Causal Ordering

We now enhance our protocol so that it implements causal ordering. In addition to maintaining GOSSIP_p and T_p, the enhanced protocol requires every process p to maintain SENT_p, an $N \times N$ associative array of non-negative integers indexed by $\mathcal{P} \times \mathcal{P}$, and DELIV_p, an associative array of N non-negative integers indexed by $\mathcal{P}$. $\mathsf{SENT}_p[q,r]$ is interpreted as the time-stamp of the latest message from q to r that p is aware of. Notice that this is not necessarily the time-stamp of the latest information r has about q that p is aware of. $\mathsf{DELIV}_p[q]$ is the time-stamp of the latest message from q that has been delivered at p. Initially, $\mathsf{GOSSIP}_p[q,r] = \mathsf{SENT}_p[q,r] = \mathsf{DELIV}_p[q] = 0$ for every $p,q,r \in \mathcal{P}$.

The idea behind the enhanced protocol is to eliminate violations of causal ordering at each individual process. Suppose we include the array SENT_p along with the information GOSSIP_p and the time-stamp $T_p(M)$ in the control information accompanying each message M sent by process p. Let $M_1 : p \Rightarrow r$ and $M_2 : q \Rightarrow r$ be messages such that $M_1 \rightarrow M_2$. When q sends M_2, the value $\mathsf{SENT}_q[p,r]$, denoting the time-stamp of the latest message sent from p to r that q is aware of, is greater than or equal to $T_p(M_1)$. If M_2 is received before M_1, r can detect that the delivery of M_2 violates causal ordering by noting that the time-stamp recorded in $\mathsf{SENT}_q[p,r]$ in the message is more recent than the time-stamp of the latest message from p to r which has actually been delivered at r.

It turns out that this localised checking of causal ordering violations at each process is sufficient to ensure that causal ordering is preserved across the entire system. Formally, the enhanced protocol is as follows.

Protocol 2 *Implementing Gossip and Causal Ordering*

1. *Sending a message $M : p \Rightarrow q$*

    - $\mathsf{GOSSIP}_p[p,p] := T_p(M)$.
    - *As in Protocol 1, let GP_M be an $N \times N$ associative array of non-negative integers indexed by $\mathcal{P} \times \mathcal{P}$ and set $GP_M := \mathsf{GOSSIP}_p$.*
    - *Let ST_M be an $N \times N$ associative array of non-negative integers indexed by $\mathcal{P} \times \mathcal{P}$ and set $ST_M := \mathsf{SENT}_p$.*

- *The actual message sent is the quadruple* $\langle M, T_p(M), GP_M, ST_M \rangle$.
- *After sending the message, update* $\mathsf{SENT}_p[p,q] := T_p(M)$.

2. *Receiving a message* $M : p \Rightarrow q$

- *Compute the function best* $: \mathcal{P} \to \{p, q\}$ *as follows:*

$$\forall r \in \mathcal{P},\ best(r) = \begin{cases} p & \text{if } GP_M[p,r] > \mathsf{GOSSIP}_q[q,r] \\ q & \text{otherwise} \end{cases}$$

- *Deliver the message only if:*

$$\text{For each } r \in \mathcal{P} \text{ such that } best(r) = p,\ ST_M[r,q] \le \mathsf{DELIV}_q[r]$$

- *If the message is delivered, update process q's data structures as follows:*
  - *For each* $r \in \mathcal{P}$ *such that* $best(r) = p$*:*

$$\begin{aligned} \mathsf{GOSSIP}_q[q,r] &:= GP_M[p,r]. \\ \forall s \in \mathcal{P}, \mathsf{GOSSIP}_q[r,s] &:= GP_M[r,s]. \\ \forall s \in \mathcal{P}, \mathsf{SENT}_q[r,s] &:= ST_M[r,s]. \end{aligned}$$

  - $\mathsf{DELIV}_q[p] := GP_M[p,p]$.
  - $\mathsf{SENT}_q[p,q] := GP_M[p,p]$.
- *Apply the delivery condition to all undelivered messages in the buffer.*

Notice that the previous protocol, Protocol 1, updates gossip information correctly across the system *regardless* of the order of delivery of messages. Since Protocol 2 retains the same protocol for maintaining gossip information and only modifies the order in which messages are delivered, it is clear that the new protocol also ensures that every process in the system updates latest gossip information consistently. We formalise this in the following theorem, without proof.

Theorem 3.1 *Protocol 2 ensures that immediately after the p-event* e_p, $\mathsf{GOSSIP}_p[q,r] = T_r(latest_{p \leftarrow q \leftarrow r}(e_p))$, *for every* $p, q, r \in \mathcal{P}$.

It is also easy to see that Protocol 2 ensures consistency of the SENT data structure. This is formalised in the following theorem. The proof is along the same lines as the gossip consistency proof of Theorem 2.1 and is omitted.

Theorem 3.2 *Protocol 2 ensures that immediately after the p-event* e_p, $\mathsf{SENT}_p[q,r] = T_q(e_q)$, *where* e_q *is the maximum event in the set* $\{send(M) \in e_p{\downarrow} \mid M : q \Rightarrow r\}$.

We want to establish the correctness of Protocol 2—that is, we want to show that it implements causal ordering. We break up the correctness argument into two steps. We first prove that the protocol is *safe*— in other words, we show that causal ordering is never violated. We then show that it is *live*—that is, every message is delivered within a finite amount of time. We need a few preliminary results to prove the safety and liveness of Protocol 2.

3.1 Preliminary results

Lemma 3.3 *Let M_1 and M_2 be messages such that $M_1 : p \Rightarrow q$ and $M_2 : r \Rightarrow s$. If $M_1 \to M_2$, the following hold:*

1. $ST_{M_2}[p,q] \geq T_p(M_1)$.
2. $GP_{M_2}[r,p] \geq T_p(M_1)$.

Proof Let $e_1 \equiv send(M_1)$ and $e_2 \equiv send(M_2)$. Let $f = latest_{r \leftarrow p}(e_2) = max_p(e_2 \downarrow)$. Since $e_1 \in e_2\downarrow$, $e_1 \sqsubseteq^* f$. Both e_1 and f are p-events, so $T_p(e_1) \leq T_p(f)$. From Theorem 3.1, we have the following: $GP_{M_2}[r,p] = T_p(latest_{r \leftarrow r \leftarrow p}(e_2)) = T_p(latest_{r \leftarrow p}(e_2)) = T_p(f)$. So, $GP_{M_2}[r,p] = T_p(f) \geq T_p(e_1)$. Similarly, it is easy to see from Theorem 3.2 that $ST_{M_2}[p,q] \geq T_p(e_1)$. □

Lemma 3.4 *Let p be a process. Then, at any stage of the computation, for every other process q, $\mathsf{DELIV}_p[q] \leq \mathsf{GOSSIP}_p[p,q]$.*

Proof It is clear from Protocol 2 that after a p-event e_p, $\mathsf{DELIV}_p[q]$ is the time-stamp $T_q(M)$ of the most recently delivered message $M : q \Rightarrow p$. So, $send(M) \in e_p\downarrow$. By Theorem 3.1, after e_p, $\mathsf{GOSSIP}_p[p,q] = T_q(latest_{p \leftarrow p \leftarrow q}(e_p)) = T_q(latest_{p \leftarrow q}(e_p))$. Clearly, $send(M) \sqsubseteq^* latest_{p \leftarrow q}(e_p)$, so $T_q(M) \leq T_q(latest_{p \leftarrow q}(e_p))$ which means that $\mathsf{DELIV}_p[q] \leq \mathsf{GOSSIP}_p[p,q]$. □

Lemma 3.5 *Let e_p be a p-event for $p \in \mathcal{P}$ and let $E_d = \{M | deliver(M) \sqsubseteq^* e_p\}$ be the set of messages whose delivery is known to p at e_p. Then, for every message $M : q \Rightarrow r$ in E_d, where $r \neq p$, there is a message $M' : s \Rightarrow p$ in E_d such that $M \to M'$.*

Proof Let $M : q \Rightarrow r$ be a message in E_d such that $r \neq p$. Since $deliver(M) \sqsubseteq^* e_p$, there must be a sequence of events $deliver(M) \equiv e_1 \sqsubset e_2 \sqsubset \cdots \sqsubset e_n \equiv e_p$ in $e_p\downarrow$. Since e_p is a p-event and $deliver(M)$ is not, if we move back along this sequence and examine e_{n-1}, e_{n-2} ..., we must eventually find an event e_k such that e_k is an s-event for some $s \neq p$ while e_{k+1} is a p-event. By the definition of the relation $\sqsubset$, it must be the case that $e_k \equiv send(M')$ and $e_{k+1} \equiv deliver(M')$ for some message $M' : s \Rightarrow p$. Since $e_1 \equiv deliver(M)$ and $e_k \equiv send(M')$, $e_1 \neq e_k$. As a result, $deliver(M) \to send(M')$, and $M \to M'$. □

3.2 Safety

We now show that Protocol 2 is *safe*—that is, we show that the protocol ensures that causal ordering is never violated. This will follow from the following theorem.

Theorem 3.6 *Let $M : p \Rightarrow q$ be a message and let e_q be a q-event. If $deliver(M) \notin e_q\downarrow$, then $M \not\to M_q$ for every message M_q such that $deliver(M_q) \in e_q\downarrow$.*

Proof Let E_q be the set of all messages delivered at q in $e_q\downarrow$. That is, $E_q = \{M' \mid deliver(M') \sqsubseteq^* e_q \text{ and } M' : r \Rightarrow q \text{ for some } r \in \mathcal{P}\}$.

The proof is by induction on the size of E_q.

1. *Base case* ($|E_q| = 1$).

   Let $E_q = \{M_1\}$, where M_1 is of the form $M_1 : r \Rightarrow q$ for some process $r \in \mathcal{P}$. Suppose that $M \rightarrow M'$ for some $M' \in e_q{\downarrow}$. By Lemma 3.5, if $M' \neq M_1$ then $M' \rightarrow M_1$. From this, it follows that $M \rightarrow M_1$. By Lemma 3.3, $ST_{M_1}[p,q] \geq T_p(M)$ and $GP_{M_1}[r,p] \geq T_p(M)$. Clearly, $T_p(M) > 0$. Since M_1 is the first message delivered at q, it is easy to see that when M_1 is received at q, $\mathsf{DELIV}_q[s] = 0$, for every $s \in \mathcal{P}$, and $\mathsf{GOSSIP}_q[s,t] = 0$ for all pairs $(s,t) \neq (q,q)$. So, $\mathsf{GOSSIP}_q[q,p] = 0 < T_p(M) \leq GP_{M_1}[r,p]$ and Protocol 2 computes $best(p)$ to be r. However, since $ST_{M_1}[p,q] \geq T_p(M) > 0 = \mathsf{DELIV}_q[p]$, the delivery condition in Protocol 2 is not satisfied and M_1 cannot be delivered. That is, $M_1 \notin E_q$, which is a contradiction.

2. *Induction step* ($|E_q| = n > 1$).

   Let $E_q = \{M_1, M_2, \ldots, M_n\}$ and $deliver(M_1) \rightarrow deliver(M_2) \rightarrow \cdots \rightarrow deliver(M_n)$. Let e'_q be the q-event immediately preceding $deliver(M_n)$—i.e., $e'_q \sqsubset deliver(M_n)$. Since there are only $n-1$ messages delivered at q in $e'_q{\downarrow}$, by the induction hypothesis, for each message M' such that $deliver(M') \sqsubseteq^* e'_q$, we have $M \not\rightarrow M'$.

   Suppose that $M \rightarrow M'$ for some message M' such that $deliver(M') \in e_q{\downarrow} \setminus e'_q{\downarrow}$. If $M' \in E_q$, it must be the case that $M' = M_n$, since $deliver(M') \notin e'_q{\downarrow}$. Otherwise, if $M' \notin E_q$, by Lemma 3.5, there is a message $M'' : s \Rightarrow q$ in E_q such that $M' \rightarrow M''$. Since $deliver(M') \notin e'_q{\downarrow}$, it must be the case that $M'' = M_n$. So, in either case, $M \rightarrow M_n$.

   Let $M_n : r \Rightarrow q$. We shall show that $M \rightarrow M_n$ implies that M_n cannot be delivered, which is a contradiction. If $M \rightarrow M_n$, by Lemma 3.3, $GP_{M_n}[r,p] \geq T_p(M)$ and $ST_{M_n}[p,q] \geq T_p(M)$.

   Claim: At e'_q, before $deliver(M_n)$, $\mathsf{GOSSIP}_q[q,p] < T_p(M)$.

   Proof of Claim:

   At e'_q, $\mathsf{GOSSIP}_q[q,p] = T_p(latest_{q \leftarrow q \leftarrow p}(e'_q))$. Let $latest_{q \leftarrow p}(e'_q) = send(M')$ where M' is a message $M' : p \Rightarrow r$. Clearly, $deliver(M') \in e'_q{\downarrow}$, so, by the induction hypothesis, $M \not\rightarrow M'$. Since M and M' are both p-events, it follows that $M' \rightarrow M$. So, $T_p(M') < T_p(M)$ and $\mathsf{GOSSIP}_q[q,p] < T_p(M)$.

   From the claim, it follows that when M_n is received, Protocol 2 computes $best(p) = r$ since $GP_{M_n}[r,p] \geq T_p(M) > \mathsf{GOSSIP}_q[q,p]$.

   By Lemma 3.4, $\mathsf{DELIV}_q[p] \leq \mathsf{GOSSIP}_q[q,p]$, so $\mathsf{DELIV}_q[p] < T_p(M)$. Since we have $ST_{M_n}[p,q] \geq T_p(M) > \mathsf{DELIV}_q[p]$, the delivery condition in Protocol 2 is not satisfied and M_n cannot be delivered. That is, $M_n \notin E_q$ which is a contradiction.

$\square$

We can now argue that Protocol 2 guarantees that causal ordering is never violated. Suppose $M_1 : p \Rightarrow q$ and $M_2 : r \Rightarrow s$ such that $M_1 \rightarrow M_2$. Then, we must have $deliver(M_2) \not\rightarrow deliver(M_1)$. If not, we have a situation which contradicts the previous theorem: set $M = M_1$, $M_q = M_2$ and $e_q = deliver(M_2)$. Then, $deliver(M) \notin e_q{\downarrow}$ but $M \rightarrow M_q$, where $deliver(M_q) \in e_q{\downarrow}$.

3.3 Liveness

Having proved that Protocol 2 is *safe*—that is, message deliveries obey casual ordering—we still have to show that it is *live*. Liveness corresponds to proving that all messages are eventually delivered.

However, we first need to prove the following lemma which depends on the safety of Protocol 2.

Lemma 3.7 *Let e_q be a q-event and $M : p \Rightarrow q$ be a message such that $\mathsf{DELIV}_q[p] < T_p(M)$ holds immediately after e_q occurs. Then $deliver(M) \notin e_q{\downarrow}$.*

Proof From Protocol 2, it is clear that $\mathsf{DELIV}_q[p]$ corresponds to the time-stamp of the message from p to q most recently delivered at q. Let $\mathsf{DELIV}_q[p] = T_p(M')$. Since $T_p(M') < T_p(M)$, we know that $M' \rightarrow M$. Suppose that $deliver(M) \in e_q{\downarrow}$. Since M' is the most recent message from p to q delivered at q, it must be the case that $deliver(M) \rightarrow deliver(M')$. But this contradicts the fact that Protocol 2 is safe (Theorem 3.6). □

Theorem 3.8 *Every message in the system is delivered within a finite amount of time.*

Proof Let $M : p \Rightarrow q$ be a message which is *never* delivered. Let U denote the set of *all* messages sent to process q which are never delivered. Let $M_s : s \Rightarrow q$ be a message in U such that $send(M_s)$ is minimal with respect to the strict partial order $\rightarrow$ on the set of events $\{send(M) \mid M \in U\}$.

Since M_s is never delivered, there must be some $r \in \mathcal{P}$ such that whenever any message is delivered at q, the comparison of GP_{M_s} and GOSSIP_q yields $best(r) = s$ and $ST_{M_s}[r, q] > \mathsf{DELIV}_q[r]$.

Let $ST_{M_s}[r, q] = T_r(M_r)$. By Theorem 3.2 we know that $send(M_r)$ is the maximum event in the set $\{send(M') \mid M' : r \Rightarrow q \textit{ and } M' \rightarrow M_s\}$.

Since $\mathsf{DELIV}_q[r] < T_r(M_r) = ST_{M_s}[r, q]$ throughout the computation, we have from Lemma 3.7 that $deliver(M_r) \notin e_q{\downarrow}$, for any q-event e_q. In other words, M_r is also a message sent to q which is *never* delivered. So $M_r \in U$. But $M_r \rightarrow M_s$, contradicting the assumption that M_s was a minimal element of U. □

3.4 Complexity

Each message M carries the following control information: $T_p(M)$, GP_M and ST_M. Since GP_M and ST_M are both of size $N \times N$, the control information consists of $O(N^2)$ time-stamps.

4 Lexicographic time-stamps

In this section we describe a further modification of Protocol 2. The new protocol is almost identical to Protocol 2, except for the values that are used for the time-stamps. The new time-stamps are pairs $\langle e, t \rangle$, where e and t are both non-negative integers. The two components of e and t are called the *epoch* and *time* components respectively.

We call these pairs *lexicographic time-stamps*. If $T_p = \langle e, t\rangle$ is a time-stamp in our new notation, we use $T_p.e$ and $T_p.t$ to denote its two components. The relation $\leq$ over lexicographic time-stamps is defined as follows. Let T_1 and T_2 be lexicographic time-stamps. $T_1 \leq T_2$ if either $T_1.e < T_2.e$ or $T_1.e = T_2.e$ *and* $T_1.t \leq T_2.t$. Observe that this is a total order, so all events occurring on a particular process are totally ordered by their lexicographic time-stamps. We set $T_p(null) = \langle 0, 0\rangle$.

We can modify Protocol 2 to work with lexicographic time-stamps instead of the linear time-stamps used earlier. The only new feature is deciding when to increment the epoch component of the time-stamp.

The n^{th} epoch of a process $p \in \mathcal{P}$ is defined as the set of all p-events e_p such that $T_p(e_p).e = n$. Every other process q in the system has some knowledge of the epoch p is in through the epoch components of time-stamps recorded in the array GOSSIP_q. In our new protocol, a process p increments its epoch from n to $n+1$ when p becomes aware that every other process in the system "knows" that p's current epoch is n.

A precise description of how processes generate lexicographic time-stamps is defined in Protocol 3.

Protocol 3 *Lexicographic time-stamps to implement causal ordering*

1. *Sending a message $M : p \Rightarrow q$*
   - *Increment $T_p.t$.*
   - $\mathsf{GOSSIP}_p[p, p] := T_p(M)$.
   - *As in Protocol 1, let GP_M be an $N \times N$ associative array of non-negative integers indexed by $\mathcal{P} \times \mathcal{P}$ and set $GP_p := \mathsf{GOSSIP}_p$.*
   - *As in Protocol 2, let ST_M be an $N \times N$ associative array of non-negative integers indexed by $\mathcal{P} \times \mathcal{P}$ and set $ST_M := \mathsf{SENT}_p$.*
   - *The actual message sent is the quadruple $(M, T_p(M), GP_M, ST_M)$.*
   - *After sending the message, update $\mathsf{SENT}_p[p, q] := T_p(M)$.*
2. *Receiving a message $M : p \Rightarrow q$*
   - *Compute the function best : $\mathcal{P} \rightarrow \{p, q\}$ as follows:*

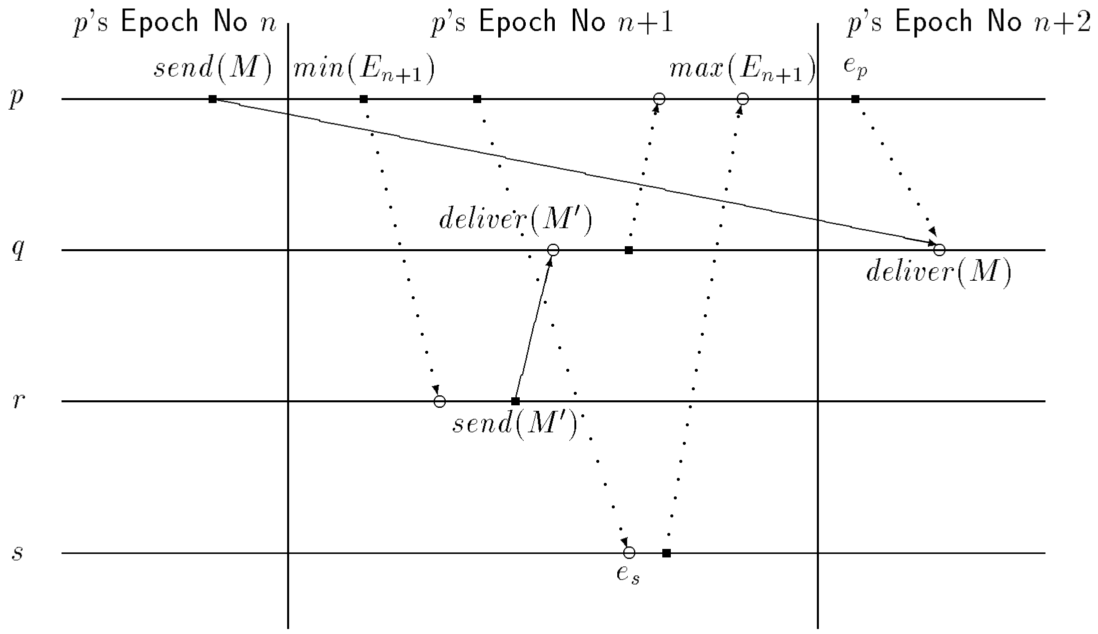
     $$\forall r \in \mathcal{P},\ best(r) = \begin{cases} p & \text{if } GP_M[p, r] > \mathsf{GOSSIP}_q[q, r] \\ q & \text{otherwise} \end{cases}$$

   - *Deliver the message only if:*

     $$\text{For each } r \in \mathcal{P} \text{ such that } best(r) = p,\ ST_M[r, q] \leq \mathsf{DELIV}_q[r]$$

   - *If the message is delivered, update process q's data structures as follows:*
     - *For each $r \in \mathcal{P}$ such that $best(r) = p$:*

       $$\begin{aligned} \mathsf{GOSSIP}_q[q, r] &:= GP_M[p, r]. \\ \forall s \in \mathcal{P}, \mathsf{GOSSIP}_q[r, s] &:= GP_M[r, s]. \\ \forall s \in \mathcal{P}, \mathsf{SENT}_q[r, s] &:= ST_M[r, s]. \end{aligned}$$



$\cdots\blacktriangleright$ Represents the "happened before" relation ($\rightarrow$) between events

$\longrightarrow$ Represents a message in the computation

Figure 2: Messages are bounded within two epochs

– $\mathsf{DELIV}_q[p] = \mathsf{SENT}_q[p,q] = GP_M[p,p]$.

- *Apply the delivery condition to all undelivered messages in the buffer.*
- *If* $\mathsf{GOSSIP}_q[r,q].e = T_q.e$ *for all* $r \in \mathcal{P}$, *update* T_q *as follows:*

$$\begin{aligned} T_q.e &:= T_q.e + 1 \\ T_q.t &:= 0 \end{aligned}$$

Notice that all the results proved for Protocol 2 in Section 4 hold for Protocol 3 as well. In other words, this protocol is also safe and live and adds $O(N^2)$ time-stamps of control information to each message.

Partially bounded time-stamps

We now establish a lemma that will help in bounding these lexicographic time-stamps. The bound that we describe is partial, in the sense that only the epoch component of the time-stamps is bounded. This lemma depends on the safety of Protocol 3 which is inferred from Theorem 3.6.

Lemma 4.1 *Let* $M : p \Rightarrow q$ *be a message such that* $T_p(M).e = n$. *Then, for every* p*-event* e_p *where* $T_p(e_p).e \geq n+2$, $e_p \not\rightarrow deliver(M)$.

Proof Let e_p be a p-event such that $T_p(e_p).e \geq n{+}2$ and $e_p \rightarrow deliver(M)$. Let E_{n+1} be the set of p-events $\{e \mid T_p(e).e = n{+}1\}$. The events in E_{n+1} are totally ordered by $\sqsubseteq^*$—let $min(E_{n+1})$ and $max(E_{n+1})$ denote the minimum and maximum elements of this set *(see*

Figure 2). From Protocol 3 we know that for every process $s \in \mathcal{P} \setminus \{p\}$, there is a *deliver* event e_s on s such that $min(E_{n+1}) \rightarrow e_s \rightarrow max(E_{n+1})$. This is because p increments its epoch from $n{+}1$ to $n{+}2$ only if it sees $\mathsf{GOSSIP}_p[s,p] = n{+}1$ for every other process s, which means that all other processes should have received messages that were sent and delivered between the occurrences of $min(E_{n+1})$ and $max(E_{n+1})$. Specifically, for q, there must be a message $M' : r \Rightarrow q$ from some process r such that $min(E_{n+1}) \rightarrow send(M') \rightarrow deliver(M') \rightarrow max(E_{n+1})$ Since $T_p(M).e = n$, we also have $send(M) \rightarrow min(E_{n+1})$. In other words, $send(M) \rightarrow min(E_{n+1}) \rightarrow send(M')$ and $deliver(M') \rightarrow max(E_{n+1}) \rightarrow e_p \rightarrow deliver(M)$. So $send(M) \rightarrow send(M')$ and $deliver(M') \rightarrow deliver(M)$, which contradicts the fact that Protocol 3 is safe. □

What Lemma 4.1 effectively states is that any message sent by a process in its n^{th} epoch is delivered before the process enters its $(n+2)^{nd}$ epoch. In other words, we will never have to compare messages whose time-stamps differ by more than one epoch.

This means that we need only three distinct values—say 0, 1 and 2—to record the epoch component of the time-stamp. We can then redefine the $\leq$ relation over lexicographic time-stamps as follows. Let T_1 and T_2 be lexicographic time-stamps. $T_1 \leq T_2$ if either $T_2.e = (T_1.e{+}1 \bmod 3)$ or $T_1.e = T_2.e$ *and* $T_1.t \leq T_2.t$. Using this definition of $\leq$, we can cycle through the three values for the epoch by modifying the last line in Protocol 3 to read:

If $\mathsf{GOSSIP}_q[r,q].e = T_q.e$ *for all* $r \in \mathcal{P}$, *update* T_q *as follows:*

$$\begin{aligned} T_q.e &:= (T_q.e{+}1 \bmod 3) \\ T_q.t &:= 0 \end{aligned}$$

5 Bounded time-stamps

In the previous section we demonstrated the use of lexicographic time-stamps and showed how they could be, in a sense, *partially bounded.* Unfortunately it is not so easy to ensure that lexicographic time-stamps, or any other kind of time-stamps for that matter, can be completely bounded for arbitrary computations.

It is easy to construct pathological cases that will lead to the breakdown of most bounded time-stamping protocols. For instance, suppose we have a computation where one process keeps sending messages to another process which, in turn, never sends any messages at all. Then there is no way for the first process to know which (if any) of its messages have been received, so that it can reuse the corresponding time-stamps.

However such computations represent extreme cases and it is possible to impose reasonable restrictions on the system so that we can bound time-stamps systematically. We place the following restrictions on the distributed system *and* the underlying computation to ensure that time-stamps can be completely bounded:

1. The labelled directed graph that describes the topology of the distributed system is strongly connected.

2. No process sends more than B messages in an epoch, where B is a non-negative integer.

The first condition restricts the physical structure of the system, independent of the actual computation. However this is essential to prevent the isolation of a process which would lead to the kind of pathological case described earlier.

The second condition specifies the actual limit on the storage for time-stamps. This bound can be easily incorporated into Protocol 3 to implement a protocol with uniformly bounded lexicographic time-stamps, as described in Protocol 4. The value B is a parameter that denotes the number of messages that a process can send in an epoch. The only difference in the new protocol is that a process is not allowed to send more messages in an epoch if B messages have already been sent. Any such additional messages are preserved in a separate buffer and are sent when the next epoch starts.

It is easy to see that the value of B affects the performance of our protocol. If B is too small, processes will have to wait too long to change epochs. If B is too large, then messages will become too bulky and efficiency is reduced.

In extreme cases, if some processes send messages very sporadically the bound B could lead to a system deadlock. So, this protocol is usable only when the computation progresses somewhat uniformly on all fronts, allowing all processes to change epochs regularly.

Protocol 4 *Bounded time-stamps to implement causal ordering*

1. *Sending a message* $M : p \Rightarrow q$

   - *Proceed with the protocol only if* $T_p.t \leq B$
   - *Increment* $T_p.t$.
   - $\mathsf{GOSSIP}_p[p,p] := T_p(M)$.
   - *As in Protocol 1, let* GP_M *be an* $N \times N$ *associative array of non-negative integers indexed by* $\mathcal{P} \times \mathcal{P}$ *and set* $GP_p := \mathsf{GOSSIP}_p$.
   - *As in Protocol 2, let* ST_M *be an* $N \times N$ *associative array of non-negative integers indexed by* $\mathcal{P} \times \mathcal{P}$ *and set* $ST_M := \mathsf{SENT}_p$.
   - *The actual message sent is the quadruple* $(M, T_p(M), GP_M, ST_M)$.
   - *After sending the message, update* $\mathsf{SENT}_p[p,q] := T_p(M)$.

2. *Receiving a message* $M : p \Rightarrow q$

   - *Compute the function best* : $\mathcal{P} \rightarrow \{p, q\}$ *as follows:*

$$\forall r \in \mathcal{P},\ \ best(r) = \begin{cases} p & \text{if } GP_M[p,r] > \mathsf{GOSSIP}_q[q,r] \\ q & \text{otherwise} \end{cases}$$

   - *Deliver the message only if:*

$$\textit{For each } r \in \mathcal{P} \textit{ such that } best(r) = p,\ ST_M[r,q] \leq \mathsf{DELIV}_q[r]$$

   - *If the message is delivered, update process q's data structures as follows:*
     - $\forall r \in \mathcal{P}$ *such that* $best(r) = p$:

$$\begin{aligned} \mathsf{GOSSIP}_q[q,r] &:= GP_M[p,r]. \\ \forall s \in \mathcal{P}, \mathsf{GOSSIP}_q[r,s] &:= GP_M[r,s]. \\ \forall s \in \mathcal{P}, \mathsf{SENT}_q[r,s] &:= ST_M[r,s]. \end{aligned}$$

- $\mathsf{DELIV}_q[p] = \mathsf{SENT}_q[p,q] = GP_M[p,p]$.

- *Apply the delivery condition to all undelivered messages in the buffer.*
- *If* $\mathsf{GOSSIP}_q[r,q].e = T_q.e$ *for all* $r \in \mathcal{P}$, *update* T_q *as follows:*

$$\begin{array}{lcl} T_q.e & := & (T_q.e{+}1 \bmod 3) \\ T_q.t & := & 0 \end{array}$$

Also despatch all unsent messages in the buffer.

Notice that all the results proved for Protocol 2 in Section 4 hold for Protocol 4 as well. This protocol too is *safe* and *live* and incorporates $O(N^2)$ time-stamps of control information with each message. However, we know that each time-stamp can be described using $O(\log B)$ bits. So, overall Protocol 4 adds only $O(N^2 \log B)$ bits of control data to each message, *regardless* of the length of the computation.

Discussion

As we mentioned earlier, Protocol 4 works for computations which proceed more or less uniformly across the system. We feel that the restrictions we impose are not unreasonable in any system where the processes are coordinating their efforts to achieve a joint goal.

An alternative approach is to implement Protocol 4 *without* explicitly incorporating the bound B in the algorithm. Then, the protocol correctly implements causal ordering for *all* systems, with the additional feature that the time-stamps generated by the protocol are uniformly bounded whenever the process graph is strongly connected and the underlying computation is "well behaved".

It appears possible to slightly weaken the conditions under which the time-stamps used by our protocol remain uniformly bounded. In [4], a bounded time-stamping protocol is described for keeping track of gossip information in arbitrary message-passing systems with reliable (but possibly non-fifo) channels which satisfy the following requirement: the number of *unacknowledged* messages between any pair of processes p and q is bounded by a constant B—a message $M : p \Rightarrow q$ is said to be *acknowledged* if the event $deliver(M)$ is visible to p; i.e., $deliver M \sqsubseteq^* e_p$ for some p-event e_p.

This is a weaker requirement than the one we have imposed on our protocol here. For instance, the requirement of [4] would permit one process to go to sleep and never transmit any more messages, provided no other process ever sends it more than B messages during the computation.

We could modify our protocol to work under this less restrictive assumption, as follows. Instead of maintaining a single clock T_p for each process p, we could maintain a clock T_{pq} for each pair of processes p and q. The clock T_{pq} is used to time-stamp messages sent from p to q. So, each process actually maintains upto N independent clocks.

Each process now needs to remember the latest values of all N clocks of every other process. So, each process maintains N^2 primary time-stamps and N^3 secondary time-stamps. We can then easily modify Protocol 1 to correctly update these N^3 values. Notice that the revised protocol would require an extra $O(N^3 \log N)$ bits to be added to each message, which is more than the $O(N^2 \log N)$ bits added in the protocol presented here.

This revised protocol would use only a bounded set of time-stamps provided each process p resets the epoch of each of its clocks T_{pq} within a bounded number of messages B. Since the length of an epoch in T_{pq} is precisely equal to the number of unacknowledged messages sent from p to q since the beginning of the epoch, this is equivalent to the restriction that there are no more than B unacknowledged messages from p to q at any time.

An additional benefit of the revised protocol is that it would be more resilient to stopping failures, where a process dies at some point and does not participate in the rest of the computation. Since each channel uses a separate set of time-stamps, the reuse of time-stamps between those processes which are still functioning is not affected by the fact that acknowledgments are no longer received from the dead process.

It seems difficult to design reasonable implementations of causal ordering which are robust in the presence of arbitrary system failures. The Isis system of [1] implements causal ordering even in the presence of channel errors, but at the cost of tagging each message with the entire message history of the system. Our protocol, like the protocols of [5] and [6], is very sensitive to channel errors, as discussed in [6]. It appears difficult to overcome this problem in general, but it would be interesting to see to what extent these simplified protocols can be enhanced to cope with specific kinds of process and channel failures.

References

[1] K. Birman and T. Joseph: Reliable communications in the presence of failures, *ACM Trans. Comput. Sci*, **5** (1) (1987), 47–56.

[2] L. Lamport: Time, clocks and the ordering of events in a distributed system, *Comm. ACM*, **21**,7 (1978) 558–565.

[3] F. Mattern: Time and global states of distributed systems, in *Proc. Int. Workshop on Parallel and Distributed Algorithms*, Bonas, France, North-Holland (1988) 215–226.

[4] M. Mukund, K. Narayan Kumar and M. Sohoni: Keeping track of the latest gossip in message-passing systems, in *Proc. Int. Workshop on Structures in Concurrency Theory*, Berlin, Springer (1995) *(to appear)*.

[5] M. Raynal, A. Schiper and S. Toueg: The causal ordering abstraction and a simple way to implement it, *Inform. Proc. Letters*, **39** (1991), 343–350.

[6] A. Schiper, J. Eggli and A. Sandoz: A new algorithm to implement causal ordering, in *Proc. 3rd Int. Workshop on Distributed Algorithms*, Nice, Springer LNCS **392**, (1989), 219–232.