Bounded time-stamping in message-passing systems*

Madhavan Mukund† K. Narayan Kumar† Milind Sohoni‡

Abstract

Consider a distributed system running a protocol in which processes exchange information by passing messages. The *gossip problem* for the protocol is the following: Whenever a process q receives a message from another process p, q must be able to decide which of p and q has more recent information about r, for every other process r in the system. With this data, q is in a position to update its knowledge about the global state of the system.

We propose a solution wherein to each message of the protocol, the sender adds information about its current state of knowledge about other processes. We do not add any *new* messages to the underlying computation. The additional information tagged onto each message is *uniformly* bounded if we make reasonable assumptions about the number of undelivered messages present at any time in the system. This means that the overhead of maintaining the latest gossip is a constant, independent of the length of the underlying computation.

Our solution to the gossip problem has several applications in the analysis of distributed systems. Many distributed algorithms rely, either explicitly or implicitly, on the local information available at a process about the global state of the system. Using our scheme, each process can ensure that during a computation it always maintains the best possible information about every other process. At a theoretical level, the gossip problem plays an important role in formal characterizations of finite-state message-passing systems.

1 Introduction

We tackle a natural problem from distributed computing, involving time-stamps. Let $\mathcal{P}$ be a set of computing agents or processes which exchange information by passing messages. The *gossip problem* is the following: Whenever a process p receives a message from another

*A preliminary version of this paper appeared as "Keeping track of the latest gossip in message-passing systems" in *Proc. Structures in Concurrency Theory (STRICT)*, Workshops in Computing Series, Springer-Verlag (1995) 249–263.

†Chennai Mathematical Institute, 92 G.N. Chetty Road, Chennai 600 017, India. E-mail: {madhavan,kumar}@smi.ernet.in

‡Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai 400 076, India. E-mail: sohoni@cse.iitb.ernet.in

process q, p must be able to decide whether q's message contains "fresh" information about r, for every other process r. Once p makes this decision, it can systematically collate this information to maintain, on-line, its "latest gossip" about every other process.

By keeping track of the latest gossip about other agents, each process can consistently update its knowledge about the global state of the system whenever it receives some new information from another process. Since computing global information about the system from local information is a central issue in distributed computing, a solution to the gossip problem would be useful in a wide variety of applications involving distributed systems.

The gossip problem has been investigated in [12] for systems where processes synchronize periodically and exchange information. We extend the solution proposed in [12] to a general message-passing model, where processes communicate by sending messages on point-to-point channels. We assume that the communication medium is reliable—in particular, all messages in the system are eventually delivered. However, we permit indefinite delays in transit. Also, messages need not be received in the order in which they were sent.

In our solution to the gossip problem, processes exchange only a bounded amount of gossip information with each message they send. At the heart of our solution is a protocol for time-stamping messages in the system using a finite set of labels.

Time-stamping is a well-established technique for ordering events in a distributed setting [9, 10]. Time-stamping protocols which use only a bounded set of labels to tag events have attracted a fair amount of attention in recent years. Protocols have been exhibited for systems in which processes communicate via a shared memory [4, 5, 7], as also for systems where processes synchronize periodically and exchange information [2, 3, 12]. However, no such protocols seem to exist for message-passing systems.

One of the main complications introduced by message-passing is that information flows in only one direction. In general, a process needs to know whether the information that it has sent out has been incorporated into the local state of the recipient. In systems with synchronous communication, each transfer of information is implicitly acknowledged. To transport the protocol in [12] from the setting of synchronous communication to the setting of message-passing, we have to introduce an explicit mechanism for collecting acknowledgments, both direct and indirect. It turns out that in a completely general message-passing model, a solution to the gossip problem using bounded time-stamps is not possible. In order to make the problem tractable, we have to restrict computations by placing a bound on the number of unacknowledged messages that can be present in the system at any time.

This restriction is a natural one—for instance, in the context of Message Sequence Charts (MSCs) [14], a popular visual formalism for specifying message-passing systems, it has been shown that regular (finite-state) specifications correspond precisely to specifications which impose a bound on the number of unacknowledged messages [6]. The importance of regularity in message-passing protocols has been established in [11], where it is shown that every robust asynchronous protocol is actually finite-state. (A protocol is said to be robust if its behaviour is insensitive to nondeterminism resulting from differences in relative speeds of the different components and delays in message delivery.) Thus, most practical protocols would satisfy the conditions required for our time-stamping protocol to work.

An important feature of our solution to the gossip problem is that it does not introduce any additional messages—it just adds additional data to each message of the underlying computation. The amount of additional data added to each message is guaranteed to be uniformly bounded. Thus, given any distributed algorithm which conforms to the restricted model we work with, we can enhance the algorithm to also keep track of the latest gossip with only a constant overhead in message complexity.

As we have already mentioned, a solution to the gossip problem is useful in designing distributed algorithms. We discuss some instances of this at the end of this paper. In addition, our time-stamping algorithm also has important applications in automata theory for message-passing systems. In [6], it is shown that finite-state versions of a distributed machine model called message-passing automata capture precisely the regular sets of message-passing specifications described by MSCs. One of the main results of the characterization proved in [6] is a decomposition theorem for automata over regular MSC languages, along the lines of Zielonka's theorem for regular trace languages [18]. This decomposition theorem uses in crucial way the solution to the gossip problem for message-passing systems. Further, we believe that our solution to the gossip problem will play a central role in developing local temporal logics for message-passing systems, just as the solution to the gossip problem in synchronous systems [12] is central to the theory of local temporal logics over Mazurkiewicz traces [13].

The paper is organized as follows. In the next section we introduce our model of computation and formulate the gossip problem in terms of a natural partial order on the events in the system. Section 3 describes ideals, which capture the notion of a partial view of a distributed computation. Sections 4 and 5 describe a protocol to solve the gossip problem. Each process maintains what we call *primary information* about the computation, using potentially unbounded labels to distinguish messages in the system. In Section 6 we show how to convert this protocol to one which uses bounded time-stamps, thereby establishing a solution to the gossip problem. In the final section, we sketch how this protocol may be applied to simplify some classes of distributed algorithms.

2 The Model

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ be a set of processes which communicate with each other through messages. We assume that messages are never inserted, lost or modified—that is, the communication medium is reliable. However, there may be an arbitrary delay between the sending of a message and its receipt. Further, messages need not be received in the order in which they were sent.

We assume that communication is point-to-point. Each message is addressed to a specific process and is not seen by any of the other processes in the system. Thus, each transmission of a message from a process p to a process q consists of two distinct actions; the action $s_{p\to q}$ corresponds to the sending of the message from p to q and the action $r_{q\leftarrow p}$ corresponds to its receipt by q.

We can regard a computation of the system as a word over the alphabet $\mathcal{C} = \mathcal{C}_S \cup \mathcal{C}_R$ where $\mathcal{C}_S = \{s_{p\to q} \mid p, q \in \mathcal{P}\}$ is the set of *send actions* and $\mathcal{C}_R = \{r_{p\leftarrow q} \mid p, q \in \mathcal{P}\}$ is the set of *receive actions*. Since each action in $\mathcal{C}$ is "executed" by a single process, we can also partition $\mathcal{C}$ across processes—for each process p, $\mathcal{C}_p = \{s_{p\to q} \mid q \in \mathcal{P}\} \cup \{r_{p\leftarrow q} \mid q \in \mathcal{P}\}$ is the set of *p-actions* that p participates in directly.

We shall regard a word $u \in \mathcal{C}^*$ of length m as a function $u : [1..m] \to \mathcal{C}$, where $[1..m]$ denotes the set $\{1, 2, \ldots, m\}$ if $m \geq 1$ and is $\emptyset$ if $m = 0$. For $u \in \mathcal{C}^*$ and $c \in \mathcal{C}$, $\#_c(u)$ denotes the number of occurrences of c in u. We can extend this to subsets $X \subseteq \mathcal{C}$: $\#_X(u) = \sum_{c \in X} \#_c(u)$.

Not every word corresponds to a valid computation—for instance, we must insist that messages are received only after they are sent. In addition, since messages need not be received in the order they were sent, to completely specify a computation we need to match each receive event to the corresponding send event. With this in mind, we define computations as follows:

Computations A *computation* over $\mathcal{C}$ is a pair (u, φ) where $u : [1..m] \to \mathcal{C}$ is a word and $\varphi : [1..m] \to [1..m]$ is a partial function such that:

(i) The domain of φ, $dom(\varphi)$ is the set of positions labelled by receive actions—that is, $dom(\varphi) = \{i \mid u(i) \in \mathcal{C}_R\}$.

(ii) φ is injective over $dom(\varphi)$—for each $i, j \in dom(\varphi)$, $i \neq j \Rightarrow \varphi(i) \neq \varphi(j)$.

(iii) For each $i \in dom(\varphi)$, $\varphi(i) < i$.

(iv) If $u(i) = r_{q \leftarrow p}$ then $u(\varphi(i)) = s_{p \to q}$.

If $\varphi(i) = j$, then $u(i)$ is a receive action whose corresponding send action is $u(j)$. By condition (ii), we may also refer to i unambiguously as $\varphi^{-1}(j)$.

EXAMPLE: *Let $\mathcal{P} = \{p, q\}$ and let u be the string $s_{p \to q} s_{p \to q} s_{p \to q} r_{q \leftarrow p} r_{q \leftarrow p}$ and φ be the function where $\varphi(4) = 3$ and $\varphi(5) = 1$. In this computation, the message sent from p to q at $u(1)$ is overtaken by the message sent at $u(3)$. Moreover, the message sent at $u(2)$ has not yet reached q.*

Events and causality The word u imposes a total, temporal order on the actions observed during a computation (u, φ). However, in order to analyze the flow of information between processes, we need a more accurate description of the cause and effect relationship between the different actions in u.

Let (u, φ) be a computation, where $u : [1..m] \to \mathcal{C}$. We associate with (u, φ) a set of events $\mathcal{E}_u = \{(i, u(i)) \mid i \in [1..m]\}$.

Let $e = (i, u(i))$ be an event in $\mathcal{E}_u$. When there is no ambiguity, we shall use e to denote both i and $u(i)$. For instance, $e \in \mathcal{C}_p$ denotes that $u(i) \in \mathcal{C}_p$—in other words, e is a *p-event.* Similarly, if we say $f = \varphi(e)$ we mean that $f = (j, u(j))$ is an event such that $\varphi(i) = j$. We shall also use $\mathcal{E}_u$ and u interchangeably in expressions such as $\#_c(\mathcal{E}_u)$, which denotes $\#_c(u)$.

As we mentioned earlier, u imposes a total, temporal order on the events in $\mathcal{E}_u$. Let $e, f \in \mathcal{E}_u$. Then $e < f$ provided $e = (i, u(i))$, $f = (j, u(j))$ and $i < j$. As usual $e \leq f$ if $e < f$ or $e = f$.

Messages introduce causality across processes. For each pair $(p, q) \in \mathcal{P} \times \mathcal{P}$ such that $p \neq q$, define $\lessdot_{pq}$ to be the ordering

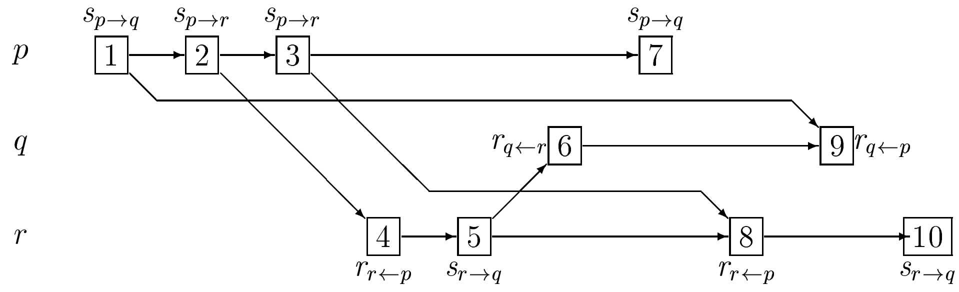

Figure 1: An example

$$e \lessdot_{pq} f \stackrel{\triangle}{=} e \in \mathcal{C}_p, f \in \mathcal{C}_q \text{ and } \varphi(f) = e.$$

In addition, each process p orders the events it participates in. Define $\lessdot_{pp}$ to be the strict ordering

$$e \lessdot_{pp} f \stackrel{\triangle}{=} e < f,\ e \in \mathcal{C}_p,\ f \in \mathcal{C}_p \text{ and for all } e < g < f,\ g \notin \mathcal{C}_p.$$

The set of all p-events in $\mathcal{E}_u$ is totally ordered by $\lessdot_{pp}^*$, the reflexive, transitive closure of $\lessdot_{pp}$.

Define $e \sqsubset f$ if for some $p, q \in \mathcal{P}$ (where p and q need not be distinct), $e \lessdot_{pq} f$ and let $\sqsubseteq$ denote the reflexive, transitive closure of $\sqsubset$. If $e \sqsubseteq f$ then we say that e is *below* f. The partial order $\sqsubseteq$ records the information we require about causality and independence between events in $\mathcal{E}_u$.

Let $e \in \mathcal{E}_u$ be a p-event. The set of events below e is $e{\downarrow} = \{f \mid f \sqsubseteq e\}$. These represent the only actions which are known to p when e occurs.

Latest information Consider a computation (u, φ) and its associated set of events $\mathcal{E}_u$. The $\sqsubseteq$-maximum p-event in $\mathcal{E}_u$ is denoted $max_p(\mathcal{E}_u)$—this is the last event in $\mathcal{E}_u$ in which p has taken part. This quantity is well defined whenever $\#_{\mathcal{C}_p}(\mathcal{E}_u) > 0$, since all p-events in $\mathcal{E}_u$ are totally ordered by $\sqsubseteq$. (Recall that $\mathcal{C}_p \subseteq \mathcal{C}$ is the set of p-actions, so $\#_{\mathcal{C}_p}(\mathcal{E}_u)$ denotes, by convention, the number of p-actions mentioned in the string u.)

Let $p, q \in \mathcal{P}$. If $\#_{\mathcal{C}_p}(\mathcal{E}_u) > 0$, the latest information p has about q in $\mathcal{E}_u$ corresponds to the $\sqsubseteq$-maximum q-event in the set $max_p(\mathcal{E}_u){\downarrow}$, provided the set of q-events below $max_p(\mathcal{E}_u)$ is not empty. We denote this event by $latest_{p\leftarrow q}(\mathcal{E}_u)$. (If there are no q-events in $max_p(\mathcal{E}_u){\downarrow}$, then $latest_{p\leftarrow q}(\mathcal{E}_u)$ is undefined.)

EXAMPLE: *Let* $\mathcal{P} = \{p, q, r\}$. *Consider the computation* (u, φ), *where* $u = s_{p\to q} s_{p\to r} s_{p\to r} r_{r\leftarrow p} s_{r\to q} r_{q\leftarrow r} s_{p\to q} r_{r\leftarrow p} r_{q\leftarrow p} s_{r\to q}$, $\varphi(4) = 2$, $\varphi(6) = 5$, $\varphi(8) = 3$ *and* $\varphi(9) = 1$. *Figure 1 is a picture of* $(\mathcal{E}_u, \sqsubseteq)$. *The arrows in the figure correspond to the basic relations* $\lessdot_{pq}$, *which generate* $\sqsubseteq$.

In this computation, $max_q(\mathcal{E}_u) = (9, r_{q\leftarrow p})$. *Though at* $max_q(\mathcal{E}_u)$, *process* q *hears from process* p, $latest_{q\leftarrow p}(\mathcal{E}_u)$ *does not correspond to* $\varphi(max_q(\mathcal{E}_u))$. *Instead,* $latest_{q\leftarrow p}(\mathcal{E}_u) = (2, s_{p\to r})$—*process* q *hears this information indirectly, via process* r.

The gossip problem

Let p, q and r be processes and (u,φ) a computation such that $latest_{p\leftarrow r}(\mathcal{E}_u)$ and $latest_{q\leftarrow r}(\mathcal{E}_u)$ are both defined. Since both of these are r-events, they must be ordered by $\lessdot^*_{rr}$. Thus, the latest information that p and q have about r will always be comparable.

The gossip problem is the following.

> *Whenever a process p receives a message from another process q, p must be able to decide whether the message from q contains more recent information about r than p already has, for every other process r in the system.*

One way to resolve this problem is as follows. As the computation progresses, each action is assigned a label by the process involved in that action. These labels allow processes to refer to events in an unambiguous manner. Each process then maintains the labels corresponding to its latest information. These labels are passed on with each communication in such a way that the process receiving the message can consistently update its own latest information.

The labels which are assigned to events during a computation are essentially *time-stamps*. A trivial solution to the time-stamping problem is for each process to maintain a local counter and assign strictly increasing counter values to the actions it executes. Along with each message, the sender attaches the largest labels it knows for every other process. Then, when process p receives a message from process q, p can compare its latest information about r with the information that q has sent about r in the message by checking whether p's "latest" r label is larger than the r label recorded in the message from q.

This scheme has the following drawback: As the computation progresses, the time-stamps assigned to events grow without bound. As a result, processes need to send longer and longer messages to transfer the labels corresponding to their latest information.

We seek a solution to the gossip problem where message lengths are bounded. This will ensure that the overhead of maintaining gossip information remains a constant, regardless of the length of the underlying computation.

To achieve this, we need to devise a scheme for labelling events using a bounded set of time-stamps. This means that the same time-stamp will be assigned, eventually, to more than one event. We need to ensure that time-stamps are reused in such a way that the update of latest information is not affected.

In principle, this should be possible. Let $\mathcal{E}_u$ be the events corresponding to the computation (u,φ) and let the number of processes in the system be N. Regardless of the number of events in $\mathcal{E}_u$, at most N^2 of them are relevant for solving the gossip problem—we only need to be able to compare the labels of events of the form $latest_{p\leftarrow q}(\mathcal{E}_u)$ for each pair $p,q\in\mathcal{P}$. In effect, at most N^2 of the events in $\mathcal{E}_u$ constitute "current" gossip. Moreover, once an event becomes "obsolete" its time-stamp can be safely reused—an "obsolete" event can never become "current" at a later stage in the computation.

However, in the completely general model we have considered so far, it is impossible to achieve our goal. Since messages can be delayed indefinitely, a process p may send unboundedly many messages to q without knowing whether any or all of them have reached. Until p receives some confirmation from q that a particular message has reached, that message's time-stamp cannot be reused. Thus, p will potentially need to use an unbounded number of time-stamps to label its messages to q. (Notice that this problem arises even if messages are delivered in the order in which they were sent—the main source of difficulty is the fact that there is no bound on the delay in delivering a particular message.)

B-bounded computations To overcome this problem, we need to restrict the class of computations we permit. Intuitively, we must bound the number of unacknowledged messages between any pair of processes. One way to achieve this is to ensure that p can send a fresh message to q only if, as far as it knows, the number of messages which it has already sent to q and which are as yet undelivered is less than B, where $B \in \mathbb{N}$ is a prespecified bound. More formally, we say that (u, φ) is a *B-bounded computation* provided the following holds:

$$\text{For each event } e = (i, s_{p\to q}) \text{ in } \mathcal{E}_u,\ \#s_{p\to q}(e{\downarrow}) - \#r_{q\leftarrow p}(e{\downarrow}) \leq B.$$

Notice that p need not get direct acknowledgments from q. For instance, r may hear from q that q has received a particular message m from p and p, in turn, may pick up this indirect information about m from r.

Even for B-bounded computations, it is not immediate that the gossip problem has a solution. Suppose process p sends process q a message m. Though p is guaranteed to receive an acknowledgment for this message by the time it sends its next B messages to q, it cannot naïvely reuse m's time-stamp once m is acknowledged. In between, q may have passed on the information in m to another process r, in which case the message m would still constitute "current" gossip for r. The process p has to have some means of recording which of its time-stamps are "in use" in the system at any given time.

For the next four sections, we assume that every computation we deal with is B-bounded.

3 Ideals

Let us fix a computation (u, φ), where $u : [1..m] \to \mathcal{C}$, and the corresponding set of events $\mathcal{E}_u$, which we shall denote as just $\mathcal{E}$ from now on, for convenience.

The main source of difficulty in solving the gossip problem is the fact that the processes in $\mathcal{P}$ need to compute global information about the computation (u, φ) while each process only has access to a local, "partial" view of u. Although partial views of (u, φ) correspond to subsets of $\mathcal{E}$, not every subset of $\mathcal{E}$ arises from such a partial view. Those subsets of $\mathcal{E}$ which do correspond to partial views of (u, φ) are called ideals.

Ideals A set of events $I \subseteq \mathcal{E}$ is called an *order ideal* if I is closed with respect to $\sqsubseteq$—that is, $e \in I$ and $f \sqsubseteq e$ implies $f \in I$ as well. We shall always refer to order ideals as just *ideals*.

The requirement that an ideal be closed with respect to $\sqsubseteq$ guarantees that the observation it represents is "consistent"—whenever an event e has been observed, so have all the events in the computation which necessarily precede e. Clearly the entire set $\mathcal{E}$ is an ideal, as is $e{\downarrow}$ for any $e \in \mathcal{E}$. It is easy to see that if I and J are ideals, so are $I \cup J$ and $I \cap J$.

EXAMPLE: *In Figure 1, the set* $I = \{(1, s_{p\to q}), (2, s_{p\to r}), (4, r_{r\leftarrow p}), (5, s_{r\to q}), (6, r_{q\leftarrow r})\}$ *is an ideal. However, the set* $J = \{(1, s_{p\to q}), (2, s_{p\to r}), (3, s_{p\to r}), (5, s_{r\to q}), (6, r_{q\leftarrow r})\}$ *is not an ideal, since* $(4, r_{r\leftarrow p}) \sqsubseteq (5, s_{r\to q})$ *but* $(4, r_{r\leftarrow p}) \notin J$.

We need to generalize the notion of $max_p(\mathcal{E})$, the maximum p-event in $\mathcal{E}$, to all ideals $I \subseteq \mathcal{E}$.

p-views For an ideal I, the $\sqsubseteq$-maximum p-event in I is denoted $max_p(I)$, provided $\#_{\mathcal{C}_p}(I) > 0$. The p-view of I is the ideal $I_p = max_p(I)\!\downarrow$. Thus, I_p consists of all events in I which p can "see". (By convention, if $max_p(I)$ is undefined—that is, if there is no p-event in I—the p-view I_p is empty.)

EXAMPLE: *In Figure 1, consider the ideal* $I = \{(1, s_{p\to q}), (2, s_{p\to r}), (4, r_{r\leftarrow p}), (5, s_{r\to q}), (6, r_{q\leftarrow r})\}$ *Then* I_p*, the p-view of* I *is* $\{(1, s_{p\to q}), (2, s_{p\to r})\}$ *whereas* I_q*, the q-view of* I*, is the entire ideal* I.

4 Primary information

For processes $p, q \in \mathcal{P}$, we have already defined $latest_{p\leftarrow q}(\mathcal{E})$, the latest information that p has about q after (u, φ). We can extend this definition to arbitrary ideals.

Let $I \subseteq \mathcal{E}$ be an ideal and $p, q \in \mathcal{P}$. Then $latest_{p\leftarrow q}(I)$ denotes the $\sqsubseteq$-maximum q-event in I_p, provided $\#_{\mathcal{C}_q}(I_p) > 0$. Thus, $latest_{p\leftarrow q}(I)$ is the latest q-event in I that p knows about. (As usual, if there is no q-event in I_p, the quantity $latest_{p\leftarrow q}(I)$ is undefined.)

It is clear that for $p \neq q$, $latest_{p\leftarrow q}(I)$ always corresponds to a send action from $\mathcal{C}_q$. However $latest_{p\leftarrow q}(I)$ need not be of the form $s_{q\to p}$; the latest information that p has about q in I may have been obtained indirectly.

To maintain and update the latest information of processes, we need to keep track of an expanded set of events which we call primary information. The primary information of a process contains not only its latest information about every other process but also information about unacknowledged messages in the system.

Message acknowledgments Let $I \subseteq \mathcal{E}$ be an ideal and $e \in I$ an event of the form $s_{p\to q}$. Then, e is said to have been *acknowledged* in I if $\varphi^{-1}(e)$ exists and, moreover, belongs to I_p. Otherwise, e is said to be *unacknowledged* in I.

Notice that it is not enough for a message to have been received in I to deem it to be acknowledged. We demand that the event corresponding to the receipt of the message be "visible" to the sending process.

For an ideal I and a pair of processes p, q, let $unack_{p\to q}(I)$ be the set of unacknowledged $s_{p\to q}$ events in I. The following observation is immediate.

Proposition 4.1 *Let* (v, ψ) *be a B-bounded computation. For every ideal* $I \subseteq \mathcal{E}_v$*,* $unack_{p\to q}(I)$ *contains at most* B *events.*

Proof Suppose that $I \subseteq \mathcal{E}_v$ and $p, q \in \mathcal{P}$ such that $unack_{p\to q}(I)$ contains more than B events. Let e be the maximum $s_{p\to q}$ event in I. Then, it follows that $unack_{p\to q}(e\!\downarrow)$ contains more than B messages. In other words, $\#s_{p\to q}(e{\downarrow}) - \#r_{q\leftarrow p}(e{\downarrow}) > B$, which violates the definition of B-boundedness.

$\square$

Primary information Let $I \subseteq \mathcal{E}$ be an ideal. The *primary information* of I, $\mathit{primary}(I)$, consists of the following events in I:

- The set $\mathit{latest}(I) = \{\mathit{max}_p(I) \mid p \in \mathcal{P}\}$.
- The collection of sets $\mathit{unack}(I) = \{\mathit{unack}_{p\to q}(I) \mid p, q \in \mathcal{P}\}$.

Let $I \subseteq \mathcal{E}$ be an ideal and p a process such that $I_p \neq \emptyset$. Then, $\mathit{primary}(I_p)$ denotes the primary information of p in I—that is, p's primary information is just the primary information of the p-view of I. Clearly, the "latest information" of p after I is contained in its primary information—for every process q, $\mathit{latest}_{p\leftarrow q}(I)$ is just $\mathit{max}_q(I_p)$.

We need to propagate implicit "acknowledgments" so that processes can update their primary information. For an ideal I and processes p, q, let $\mathit{ack\text{-}pending}_{q\leftarrow p}(I)$ denote the set of messages from p to q which have been received by q in I but whose receipt, as far as q knows, is not yet known to the sender p. Formally, an $s_{p\to q}$ event e in I belongs $\mathit{ack\text{-}pending}_{q\leftarrow p}(I)$ if $\varphi^{-1}(e) \in I_q$ but $\varphi^{-1}(e) \notin (I_q)_p$. It is clear that $\mathit{ack\text{-}pending}_{q\leftarrow p}(I) \subseteq \mathit{unack}_{p\to q}(I_q)$ and thus never contains more than B events. As with $\mathit{latest}(I)$ and $\mathit{unack}(I)$, we write $\mathit{ack\text{-}pending}(I)$ to denote the collection $\{\mathit{ack\text{-}pending}_{q\leftarrow p}(I) \mid p, q \in \mathcal{P}\}$.

To compare and update primary information, processes will also need to remember how their primary events are ordered by $\sqsubseteq$.

Primary graph Let $I \subseteq \mathcal{E}$. The *primary graph* of I, $\mathit{primary\text{-}graph}(I)$, is the directed graph (V, E) where:

- $V = (e, \alpha)$ where e is an event in $\mathit{primary}(I)$ and $\alpha \in \{\mathit{latest}, \mathit{unack}\}$ indicates the component of $\mathit{primary}(I)$ to which e belongs. (The flag α is required because e may play multiple roles in $\mathit{primary}(I)$, and these roles may change independent of each other. For instance, an event which is initially both in $\mathit{unack}(I)$ and $\mathit{latest}(I)$ may cease to be in $\mathit{latest}(I')$ for $I \subseteq I'$ but still remain in $\mathit{unack}(I')$.)
- For $v_1, v_2 \in V$, let e_1 and e_2 be the corresponding events from I. Then, $(v_1, v_2) \in E$ iff $e_1 \sqsubseteq e_2$.

As with primary information, the primary graph of a process p in I is just the graph $\mathit{primary\text{-}graph}(I_p)$.

For the moment we shall ignore the issue of assigning bounded time-stamps to events and assume that events are assigned unambiguous labels by some mechanism. For instance, as we mentioned earlier, each process could maintain a local counter and assign an increasing sequence of unique time-stamps to the events that it participates in.

Our first goal is to exhibit a procedure by which processes update their primary graphs *without* relying on the temporal order implicit in the event labels. Processes will only utilize the information about causality recorded in the primary graphs. All comparisons and updates of primary information will based purely on equality of event labels. This feature will allow us to extend the algorithm smoothly to the case where processes reuse labels.

5 Comparing primary information

Let $\mathcal{E}$ be the set of events corresponding to a computation (u, φ). Recall that each ideal $I \subseteq \mathcal{E}$ corresponds to a possible partial computation of (u, φ). Let us assume that at the end of any partial computation I, each process maintains the information *primary-graph*(I_p) and *ack-pending*(I_p).

In general, for an ideal $I \subseteq \mathcal{E}$, each pair of processes p and q will have incomparable information about I. The events known to both p and q lie in the ideal $I_p \cap I_q$. Events lying "above" the intersection are known to one of p or q but not both.

Suppose that q receives a message from p during the computation. Then, we have an event $e_q \in \mathcal{E}$ of the form $r_{q \leftarrow p}$ and a corresponding event $e_p \in \mathcal{E}$ of the form $s_{p \rightarrow q}$ such that $\varphi(e_q) = e_p$.

Let $e'_q = max_q(e_q{\downarrow} \setminus \{e_q\})$—that is, e'_q is the maximum q-event strictly below e_q. Thus, $e'_q{\downarrow}$ represents the state of q's knowledge before receiving this message from p. Let I be the ideal $e_p{\downarrow} \cup e'_q{\downarrow}$.

There are two possibilities for the information contained in the message sent at e_p.

(i) If $max_p(e'_q{\downarrow}) \sqsubset^+ e_p$, then the message sent at e_p has "new" information for q about the state of p, so q, on receiving the message at e_q, has to make some non-trivial updates to its primary information.

(ii) If $e_p \sqsubseteq max_p(e'_q{\downarrow})$, then the message sent at e_p and received at e_q is "stale" and should essentially be ignored by q.

Fortunately, it is easy to determine which of the two situations hold.

Proposition 5.1 *Let e_p be a $s_{p\rightarrow q}$ event such that $\varphi^{-1}(e_p) = e_q$ and let $e'_q = max_q(e_q{\downarrow} \setminus \{e_q\})$. Then $e_p \sqsubseteq max_p(e'_q{\downarrow})$ iff $e_p \in unack_{p\rightarrow q}(e'_q{\downarrow})$.*

It is easy to see that if $e_p \sqsubseteq max_p(e'_q{\downarrow})$ then $e_p{\downarrow} \subseteq e'_q{\downarrow}$. For every other process r, $max_r(e_p{\downarrow}) \sqsubseteq max_r(e'_q{\downarrow})$. It then follows that the only update that q has to make to its local information is to add e_p to the set *ack-pending*$_{q\leftarrow p}(e_q{\downarrow})$. The rest of *ack-pending*$(e_q{\downarrow})$ and all of *primary-graph*$(e_q{\downarrow})$ are inherited from *ack-pending*$(e'_q{\downarrow})$ and *primary-graph*$(e'_q{\downarrow})$, respectively.

For the rest of the section, we concentrate on the non-trivial situation where $max_p(e'_q{\downarrow}) \sqsubset^+ e_p$. Then, $e_p{\downarrow} = I_p$ and $e'_q{\downarrow} = I_q$. Our strategy is to arrange for p to send *primary-graph*(I_p) and *ack-pending*(I_p) along with the message sent at e_p. On the other hand, before receiving this message, q's information consists of *primary-graph*(I_q) and *ack-pending*(I_q). We will establish that q can construct *primary-graph*$(e_q{\downarrow})$ and *ack-pending*$(e_q{\downarrow})$ if it knows *primary-graph*(I_p), *ack-pending*(I_p), *primary-graph*(I_q) and *ack-pending*(I_q).

Our first observation is that if q knows both *primary-graph*(I_p) and *primary-graph*(I_q), it can determine which events in the two primary graphs lie within $I_p \cap I_q$ and which lie outside this intersection.

Lemma 5.2 *Let $I \subseteq \mathcal{E}$ be an ideal and p, q a pair of processes. Then, for each maximal event e in $I_p \cap I_q$, either $e \in latest(I_p) \cap unack(I_q)$ or $e \in unack(I_p) \cap latest(I_q)$.*

Proof First suppose that $I_p \setminus I_q$ and $I_q \setminus I_p$ are both nonempty. Let e be a maximal event in $I_p \cap I_q$. Suppose e is an r-event, for some $r \in \mathcal{P}$. Since $I_p \setminus I_q$ and $I_q \setminus I_p$ are both nonempty, it follows that $r \notin \{p, q\}$. The event e must have $\sqsubset$-successors in both I_p and I_q. However, observe that any event f in $\mathcal{E}$ can have at most two immediate $\sqsubset$-successors—one "internal" successor within the process and, if f is a send event, one "external" successor corresponding to the matching receive event.

Thus, the maximal event e must be a send event, with a $\lessdot_{rr}$ successor e_r and a $\lessdot_{rs}$ successor e_s, corresponding to some $s \in \mathcal{P}$. Assume that $e_r \in I_q \setminus I_p$ and $e_s \in I_p \setminus I_q$. Since the r-successor of e is outside I_p, $e = max_r(I_p)$, so e belongs to $latest(I_p)$. On the other hand, e is an unacknowledged $s_{r\to s}$ event in I_q. Thus, $e \in unack_{r\to s}(I_q)$, which is part of $unack(I_q)$.

Symmetrically, if $e_r \in I_p \setminus I_q$ and $e_s \in I_q \setminus I_p$, e belongs to $unack(I_p) \cap latest(I_q)$.

We still have to consider the case when $I_p \subseteq I_q$ or $I_q \subseteq I_p$. Suppose that $I_p \subseteq I_q$, so that $I_p \cap I_q = I_p$. Let $e = max_p(I_q)$. Clearly, $I_p = e\!\downarrow$. Consider any r-event f in I_p, where $r \neq p$. Since $f \sqsubset^+ e$, f cannot be maximal in I_p. Thus, the only maximal event in I_p is the p-event e. Since e has a successor in I_q, e must be a send event and is hence in $unack(I_p)$. Thus, $e \in unack(I_p) \cap latest(I_q)$. Symmetrically, if $I_q \subseteq I_p$, the unique maximal event e in I_q belongs to $latest(I_p) \cap unack(I_q)$.

□

Thus, when q receives p's primary graph, q can collect together in a set M all the events that lie in $latest(I_p) \cap unack(I_q)$ and $unack(I_p) \cap latest(I_q)$. Clearly $M \subseteq I_p \cap I_q$ and, by the preceding lemma, the events in M subsume the maximal events in $I_p \cap I_q$.

The process q can use M to check whether a primary event $e \in primary(I_p) \cup primary(I_q)$ lies inside or outside the intersection—e lies inside the intersection iff it lies below one of the elements in M. These comparisons can be made using the edge information in the graphs *primary-graph*(I_p) and *primary-graph*(I_q).

Now, it is easy for q to compare the events in $latest(I_p)$ with those in $latest(I_q)$ to determine which of p and q have more recent information about every other process r.

Lemma 5.3 *Let $I \subseteq \mathcal{E}$ be an ideal and p, q a pair of processes. Let $e = max_r(I_p)$ and $f = max_r(I_q)$ such that $e \neq f$. Then, $e \sqsubset^+ f$ iff $f \in I_q \setminus I_p$. Moreover, one can effectively determine whether f belongs to $I_q \setminus I_p$ using the information in primary-graph(I_p) and primary-graph(I_q).*

Proof We first establish that $e \sqsubset^+ f$ iff $f \in I_q \setminus I_p$.
($\Rightarrow$) Suppose that $e \sqsubset^+ f$. If f belongs to I_p then $e \neq max_r(I_p)$, which is a contradiction. On the other hand, f clearly belongs to I_q, so $f \in I_q \setminus I_p$.

($\Leftarrow$) Suppose that $f \in I_q \setminus I_p$. If $f \sqsubset^+ e$, then $f \in I_p$ since I_p is an ideal. This is a contradiction. So it is not the case that $f \sqsubset^+ e$. Since $f \neq e$ and all r-events are totally ordered by $\sqsubseteq$, we must have $e \sqsubset^+ f$.

Finally, we have to show that one can effectively determine whether f belongs to $I_q \setminus I_p$ using the information available in *primary-graph*(I_p) and *primary-graph*(I_q). Observe that $f \in I_q \setminus I_p$ iff $f \notin I_p \cap I_q$. We know that the set of events $M = (\mathit{latest}(I_p) \cap \mathit{unack}(I_q)) \cup (\mathit{unack}(I_p) \cap \mathit{latest}(I_q))$ is contained in $I_p \cap I_q$ and subsumes all the maximal events in $I_p \cap I_q$. Thus, $f \in I_p \cap I_q$ iff f is dominated by some element from M. This can be checked using the edge information in *primary-graph*(I_p) and *primary-graph*(I_q).

$\square$

Once q has compared all events of the form $max_r(I_p)$ and $max_r(I_q)$, it can easily update its sets $\mathit{unack}_{r\to s}(I_q)$, where $r \neq q$. The process which has better information about r also has better information about unacknowledged events of the form $s_{r\to s}$ in I. In other words, q inherits the sets $\mathit{unack}_{r\to s}(I_p)$ for every process r such that $max_r(I_p)$ is more recent than $max_r(I_q)$. On the other hand, if $max_r(I_p)$ is older than $max_r(I_q)$, then q ignores p's sets $\mathit{unack}_{r\to s}(I_p)$ since it already has better information about these events. Formally, we have the following.

Proposition 5.4 *For every pair of processes (r, s) such that $r \neq q$,*

$$\mathit{unack}_{r\to s}(e_q\!\downarrow) = \begin{cases} \mathit{unack}_{r\to s}(I_p) & \text{if } max_r(I_q) \sqsubset^+ max_r(I_p) \\ \mathit{unack}_{r\to s}(I_q) & \text{otherwise} \end{cases}$$

Recall that e_q was the event where q received p's message sent at e_p. At this stage, using the data in *primary-graph*(I_p) and *primary-graph*(I_q), q has updated all of *primary*$(e_q\!\downarrow)$ except for the sets $\{\mathit{unack}_{q\to r}(e_q\!\downarrow)\}_{r\in\mathcal{P}}$. Process q has also yet to update *ack-pending*$(e_q\!\downarrow)$.

We first describe how to construct *ack-pending*$(e_q\!\downarrow)$. We begin by purging from *ack-pending*$_{q\leftarrow p}(I_q)$ any event e such that $\varphi(e)$ no longer appears in $\mathit{unack}_{p\to q}(e_q\!\downarrow)$. We then add the newly sent event e_p to *ack-pending*$_{q\leftarrow p}(I_q)$. More formally, *ack-pending*$_{q\leftarrow p}(e_q\!\downarrow) =$ (*ack-pending*$_{q\leftarrow p}(I_q) \cap \mathit{unack}_{p\to q}(e_q\!\downarrow)) \cup \{e_p\}$. For $s \neq p$, we perform a similar update to obtain *ack-pending*$_{q\leftarrow s}(e_q\!\downarrow)$, except we don't add e_p at the end. In other words, for $s \neq p$, *ack-pending*$_{q\leftarrow s}(e_q\!\downarrow) =$ *ack-pending*$_{q\leftarrow s}(I_q) \cap \mathit{unack}_{s\to q}(e_q\!\downarrow)$. To fill in the rest of *ack-pending*$(e_q\!\downarrow)$, we need the following observation, which we state without proof.

Proposition 5.5 *For every pair of processes (r, s) such that $s \neq q$,*

$$\textit{ack-pending}_{s\leftarrow r}(e_q\!\downarrow) = \begin{cases} \textit{ack-pending}_{s\leftarrow r}(I_p) & \text{if } max_s(I_q) \sqsubset^+ max_s(I_p) \\ \textit{ack-pending}_{s\leftarrow r}(I_q) & \text{otherwise} \end{cases}$$

Process q can now use the information in *ack-pending*$(e_q\!\downarrow)$ to update the sets $\{\mathit{unack}_{q\to r}(I_q)\}_{r\in\mathcal{P}}$ by purging acknowledged events from these lists. Formally, for every process r, we obtain $\mathit{unack}_{q\to r}(e_q\!\downarrow)$ from $\mathit{unack}_{q\to r}(I_q)$ by purging all events which lie in *ack-pending*$_{r\leftarrow q}(e_q\!\downarrow)$.

Having constructed the sets $\mathit{latest}(e_q\!\downarrow)$ and $\mathit{unack}(e_q\!\downarrow)$, we need to add edges between the (annotated) events in these sets to obtain the graph *primary-graph*$(e_q\!\downarrow)$.

Let $f_1 = (e_1, \alpha_1), f_2 = (e_2, \alpha_2) \in \mathit{primary}(e_q\downarrow)$, where $e_1, e_2 \in \mathcal{E}$ and $\alpha_1, \alpha_2 \in \{\mathit{latest}, \mathit{unack}\}$. Recall that we draw an edge from f_1 to f_2 in $\mathit{primary\text{-}graph}(e_q\downarrow)$ iff $e_1 \sqsubseteq e_2$.

If both f_1 and f_2 came from $\mathit{primary}(I_p)$, then $e_1 \sqsubseteq e_2$ iff there was an edge from f_1 to f_2 in $\mathit{primary\text{-}graph}(I_p)$. A symmetric situation applies if both f_1 and f_2 were contributed by $\mathit{primary}(I_q)$.

The only interesting case is when f_1 and f_2 originally came from different processes. Without loss of generality, suppose that f_1 came from $\mathit{primary}(I_p)$ and f_2 from $\mathit{primary}(I_q)$. From the definition of primary information, it is not difficult to argue that the underlying events e_1 and e_2 are different from each other. Our method for comparing primary events then guarantees that e_1 was in $I_p \setminus I_q$ and e_2 was in $I_q \setminus I_p$. Thus, e_1 and e_2 are unordered in $\mathcal{E}$ and there should be no edge in either direction between f_1 and f_2 in $\mathit{primary\text{-}graph}(e_q\downarrow)$.

The following general statement which summarizes the results of this section.

Lemma 5.6 *Let e_p be a $s_{p\to q}$ event in $\mathcal{E}$ such that $\varphi^{-1}(e_p) = e_q$. Let $e'_q = \mathit{max}_q(e_q\downarrow \setminus \{e_q\})$. Then, q can construct $\mathit{primary\text{-}graph}(e_q\downarrow)$ and $\mathit{ack\text{-}pending}(e_q\downarrow)$ from $\mathit{primary\text{-}graph}(e_p\downarrow)$, $\mathit{ack\text{-}pending}(e_q\downarrow)$, $\mathit{primary\text{-}graph}(e'_q\downarrow)$ and $\mathit{ack\text{-}pending}(e'_q\downarrow)$.*

6 Bounded time-stamps

To make the protocol described in the previous section effective, we have to bound the amount of information recorded in the primary graph of each process by limiting the size of the labels used to identify events.

As in the previous section, assume that q receives a message from p at e_q, with $e_p = \varphi(e_q)$ and $e'_q = \mathit{max}_q(e_q\downarrow \setminus \{e_q\})$. When constructing $\mathit{primary\text{-}graph}(e_q\downarrow)$ and $\mathit{ack\text{-}pending}(e_q\downarrow)$, the only events whose labels have to be compared are those which lie in $\mathit{primary\text{-}graph}(e_p\downarrow) \cup \mathit{primary\text{-}graph}(e'_q\downarrow) \cup \{\mathit{ack\text{-}pending}_{q\leftarrow r}(e'_q\downarrow)\}_{r\in\mathcal{P}} \cup \{\mathit{ack\text{-}pending}_{r\leftarrow q}(e'_q\downarrow)\}_{r\in\mathcal{P}}$. In other words, q never needs to compare labels of events in $\mathit{ack\text{-}pending}(e_p\downarrow)$ or sets of the form $\mathit{ack\text{-}pending}_{r\leftarrow s}(e'_q\downarrow), q \notin \{r, s\}$. Call an event e "current" in I if e belongs to $\mathit{primary}(I_p) \cup \{\mathit{ack\text{-}pending}_{p\leftarrow q}(I_p)\}_{q\in\mathcal{P}} \cup \{\mathit{ack\text{-}pending}_{q\leftarrow p}(I_p)\}_{q\in\mathcal{P}}$ for some process p.

Let N be the number of processes in the system. Since the underlying computation is B-bounded, we know that there are at most $N + BN^2$ distinct events in $\mathit{primary}(I_p)$ for process p—there are at most N events in $\mathit{latest}(I_p)$ and for each pair of processes (q, r), there at most B events in the sets $\mathit{unack}_{q\to r}(I_p)$. Similarly, there are at most BN events each in $\{\mathit{ack\text{-}pending}_{p\leftarrow q}(I_p)\}_{q\in\mathcal{P}}$ and $\{\mathit{ack\text{-}pending}_{q\leftarrow p}(I_p)\}_{q\in\mathcal{P}}$. Thus, at any given time, the number of events across the system which are current is bounded by $N((2B+1)N + BN^2)$.

Each send event $s_{p\to q}$ begins by being current—the moment the message is sent, the event is added to the list of unacknowledged messages from p to q. Eventually, q acknowledges this message, p purges it from its unacknowledged list and, finally, q eliminates it from its list of pending acknowledgments. Meanwhile, as the computation progresses, this event may get added to the primary information or pending acknowledgments of other processes. However, gradually it recedes into the past, until it drops out of the primary information and pending acknowledgments of *all* processes. At this time, the label assigned to this event can be reused—the old event with the same label can *never* become current again.

Processes can keep track of which events in the system are current by maintaining one additional level of data, called secondary information.

Secondary information Let I be an ideal. The *secondary information* of I is the collection of (indexed) sets $\mathit{primary}(e\!\downarrow)$ for each event e in $\mathit{primary}(I)$. This collection of sets is denoted $\mathit{secondary}(I)$.

The following lemma says that the only p-events which can be current in the system are those which occur in p's secondary information.

Lemma 6.1 *Let $I \subseteq \mathcal{E}$ be an ideal and e a p-event which belongs to $\mathit{primary}(I_q) \cup \mathit{ack\text{-}pending}_{q \leftarrow p}(I_q)$ for some process q. Then, $e \in \mathit{secondary}(I_p)$.*

Proof We begin with a simple observation, which we state without proof. Let f be an event and I, J be ideals such that $f \in \mathit{primary}(I)$ and $f \in J \subseteq I$. Then, $f \in \mathit{primary}(J)$ as well.

Now, suppose that $e \in \mathit{primary}(I_q)$. Clearly, $e \in I_p \cap I_q$, so, by the preceding observation, $e \in \mathit{primary}(I_p \cap I_q)$. By Lemma 5.2, we know that each maximal element f in $I_p \cap I_q$ belongs to $\mathit{unack}(I_p) \cap \mathit{latest}(I_q)$ or $\mathit{latest}(I_p) \cap \mathit{unack}(I_q)$. Thus, e belongs to $\mathit{primary}(f\!\downarrow)$ for some $f \in \mathit{unack}(I_p) \cup \mathit{latest}(I_p)$, whence e belongs to $\mathit{secondary}(I_p)$.

On the other hand, suppose that $e \in \mathit{ack\text{-}pending}_{q \leftarrow p}(I_q)$. It follows that $e \in \mathit{unack}_{p \rightarrow q}(I_q)$. Once again, e belongs to $\mathit{unack}(f\!\downarrow)$ for some $f \in \mathit{unack}(I_p) \cup \mathit{latest}(I_p)$, so e belongs to $\mathit{secondary}(I_p)$.

□

We will use the preceding result in the following form.

Corollary 6.2 *Let e be a p-event such that $e \notin \mathit{secondary}(I_p)$. Then $e \notin \mathit{primary}(I_q) \cup \mathit{ack\text{-}pending}_{q \leftarrow p}(I_q)$ for any $q \in \mathcal{P}$. In other words, if $e \notin \mathit{secondary}(I_p)$ then e is not current in I.*

Our update procedure does not rely on the temporal order implicit in event labels. So long as all processes which refer to the same label in their primary information are actually talking about the same event, reusing labels should cause no confusion. Therefore, if p knows that no p-event labelled ℓ is currently part of the primary information of any process in the system, it can safely use ℓ to time-stamp the next message which it sends.

Secondary information can be updated in a straightforward manner when we update primary information—if q inherits an event e from p's primary information, it also inherits the secondary information $\mathit{primary}(e\!\downarrow)$ associated with e. Notice that it suffices to maintain secondary information as an indexed set—we do not need to maintain secondary *graphs* as we do primary graphs.

The number of events in $\mathit{secondary}(e\!\downarrow)$ is at most $(N + BN^2)^2$; we have already seen that $\mathit{primary}(e\!\downarrow)$ has at most $N + BN^2$ events and corresponding to each of these primary events, we have $N + BN^2$ secondary events.

This at once gives us a protocol which solves the gossip problem for B-bounded computations.

The gossip protocol

Let $\mathcal{L}$ be a finite set of labels of such that $|\mathcal{L}| > (N + BN^2)^2$. All processes use the set $\mathcal{L}$ to time-stamp messages. Each process p maintains its primary graph $\mathit{primary\text{-}graph}_p = (V_p, E_p)$ where V_p consists of the (indexed) sets of labels latest_p and unack_p. In addition, p also maintains the (indexed) set of labels $\mathit{ack\text{-}pending}_p$.

A typical element of latest_p is a pair of the form (ℓ, q)—this will mean that the maximum q-event known to p is time-stamped ℓ. Elements of unack_p and $\mathit{ack\text{-}pending}_p$ are triples. An entry (ℓ, q, r) in unack_p signifies that, as far as p knows, the $s_{q\to r}$ event labelled ℓ has not been acknowledged. In the same vein, a typical entry (ℓ, q, r) in $\mathit{ack\text{-}pending}_p$ denotes that p knows that the message from q to r time-stamped ℓ has actually been delivered at r but, as far as p knows, r believes that q does not know that this message has been received.

Finally, the process p maintains its secondary information $\mathit{secondary}_p$ as an indexed set of labels. If $\bar{e}$ is a tuple from $\mathit{latest}_p \cup \mathit{unack}_p$, then an event in $\mathit{latest}(\bar{e}{\downarrow})$ will be represented as $(\ell', r, \bar{e})$, indicating that the maximum r-event in $\bar{e}{\downarrow}$ is time-stamped ℓ'. In a similar manner, an entry $(\ell', r, s, \bar{e})$ in $\mathit{unack}(\bar{e}{\downarrow})$ signifies that there is a $s_{r\to s}$ event time-stamped ℓ' which is unacknowledged within $\bar{e}{\downarrow}$.

Initially, for each p, latest_p, unack_p, $\mathit{ack\text{-}pending}_p$ and $\mathit{secondary}_p$ are empty.

Sending a message When p sends a message to q it does the following:

- Choose a label ℓ from $\mathcal{L}$ which does not appear as the first component of any tuple in $\mathit{secondary}_p$.
- Remove the old event (ℓ', p) from latest_p, if it exists. Also remove all associated events from $\mathit{secondary}_p$—that is, tuples of the form (ℓ'', p', ℓ', p) and $(\ell'', p', p'', \ell', p)$.
- Add (ℓ, p, q) to unack_p and (ℓ, p) to latest_p. Add an edge in E_p from each tuple in $\mathit{latest}_p \cup \mathit{unack}_p$ to the new tuples $(\ell, p, q) \in \mathit{unack}_p$ and $(\ell, p) \in \mathit{latest}_p$.
- For each pair (ℓ', p') in latest_p, add (ℓ', p', ℓ, p) to $\mathit{secondary}_p$. Similarly, for each triple (ℓ', p', p'') in unack_p, add $(\ell', p', p'', \ell, p)$ to $\mathit{secondary}_p$.
- Send $\mathit{primary\text{-}graph}_p$ and $\mathit{secondary}_p$ to q.

Receiving a message On receiving a message from p, q does the following:

- Extract the label ℓ of the new message.
- Add the triple (ℓ, p, q) to $\mathit{ack\text{-}pending}_q$.
- If (ℓ, p, q) does not already belong to unack_q then update $\mathit{primary\text{-}graph}_q$, $\mathit{ack\text{-}pending}_q$ and $\mathit{secondary}_q$ by comparing $\mathit{primary\text{-}graph}_p$, $\mathit{ack\text{-}pending}_p$ and $\mathit{secondary}_p$ in the message with $\mathit{primary\text{-}graph}_q$, $\mathit{ack\text{-}pending}_q$ and $\mathit{secondary}_q$ currently maintained by q.

On sending a message, p chooses a label ℓ which is not currently in use in the system and uses ℓ to time-stamp the message. It then replaces the latest p label in $latest_p$ by ℓ and also adds ℓ to the list of unacknowledged $s_{p\to q}$ events. Finally, it places the new event at the "top" of its primary graph and sets the secondary information with respect to the new event to be its overall primary information. It then sends its current data structures *primary-graph*$_p$, *ack-pending*$_p$ and *secondary*$_p$ to q.

When q receives the message labelled ℓ, it first adds this message to its set *ack-pending*$_q$ of messages which have been received but whose receipt is as yet unknown to p. (Extracting the label of the message can be done by looking, for instance, for the E_p^*-maximal event in *primary-graph*$_p$). It then checks whether the message is new. If so, it updates its primary graph and secondary information following the results in Lemmas 5.2 and 5.3 and Propositions 5.4 and 5.5.

Message complexity

Lemma 6.3 *Let N be the number of processes in the system. For each process p, the information in primary-graph$_p$ and secondary$_p$ can be written down using at most $O(B^2N^4(\log B + \log N))$ bits.*

Proof We know that $|\mathcal{L}|$ is $O(B^2N^4)$. So each label in $\mathcal{L}$ can be written down using $O(\log B + \log N)$ bits. Similarly, each process name can be written down using $O(\log N)$ bits. So each tuple in the sets $latest_p$, $unack_p$, *ack-pending*$_p$ and *secondary*$_p$ requires only $O(\log B + \log N)$ bits to write down.

Since there are $O(BN^2)$ entries in *primary*$_p$, the edge relation E_p of *primary-graph*$_p$ can be represented in terms of an adjacency matrix, using $O(B^2N^4)$ bits. Hence, all of *primary-graph*$_p$ can be written down in $O(B^2N^4)$ bits.

The real bottleneck turns out to be *secondary*$_p$. For each of the $O(BN^2)$ elements in $\bar{e} \in$ *primary*$_p$, we have to maintain *primary*$(\bar{e}\downarrow)$, which requires $O(BN^2(\log B + \log N))$ bits. Overall *secondary*$_p$ requires $O(B^2N^4(\log B + \log N))$ bits to write down.

□

Putting together all the results we have proved so far, we can state the following theorem

Theorem 6.4 *The protocol we have described solves the gossip problem for B-bounded computations with only a bounded amount of additional information being added to each message of the underlying computation.*

7 Applications in distributed algorithms

The time-stamping protocol we have described can be used to implement natural solutions to several standard problems in distributed computing.

Our first example considers the problem of recording consistent global states. The global state of a message-passing system consists of the local state of each process, together with the state of each channel (that is, information about messages that are as yet undelivered). The problem of recording consistent global states is the following: each process in the system should record information locally about the computation such that this local information can be collated to arrive at a "legal" global state which could *potentially* have been reached during the computation.

In [1], Chandy and Lamport describe an algorithm to generate such a *distributed snapshot* of the system. In their algorithm, when a process decides that a snapshot of the system state is required, it records its local state and sends a special marker message to each process in the system. Across the system, other processes may also spontaneously record their state. Once a process records its state, it begins recording the sequence of messages received on each incoming channel. It can be shown that the local states and local channel information recorded at each process at the end of the protocol constitutes a legal global state.

The Chandy-Lamport algorithm requires an additional protocol, with its own messages, to be run alongside the main computation. In contrast, Yadulla [17] describes a scheme for recording global states which runs in the background of each computation and uses the time-stamping protocol described in this paper. In Yadulla's proposal, each process p uses the information contained in the sets $latest(I)$, $unack(I)$ and $ack\text{-}pending(I)$ to reconstruct the global state of I_p. Given this information about I_p for each $p \in \mathcal{P}$, it is easy to reconstruct the global state of the system at I. It is also worth noting that the Chandy-Lamport algorithm assumes that messages are delivered in fifo order, while Yadulla's solution can also be applied to systems where messages may get reordered in transit. More details are available in [17].

Another example of the use of time-stamping in distributed algorithms is in implementing *causal ordering*. Suppose that individual channels in a message-passing system behave in a fifo fashion. This does not prevent information from arriving out of order globally. For instance, p may send a message m_1 to q, followed by a message m_2 to r. After this, r may send a message m_3 to q. Causal order is violated if the message m_3 from r is received by q before the message m_1 from p.

One way to solve this problem is to add a time-stamp to each message and maintain a local buffer at each process. Whenever a message is received, its time-stamp is checked. The time-stamp should contain enough information to determine whether the arrival of this message has violated causal order. If there is no violation of causal order, the main message may be read by the recipient. Otherwise, the message is added to the local buffer to be read later.

The first time-stamping protocol to ensure causal ordering was proposed in [16]. Establishing the correctness of the time-stamping scheme of [16] is rather complicated and a protocol which uses a simpler time-stamping scheme was introduced in [15]. It turns out that the bounded time-stamping protocol described here can be used to derive a much more natural time-stamping scheme for causal ordering [8]. Moreover, unlike the time-stamps of [16] and [15] which can grow arbitrarily large, the time-stamping scheme of [8] guarantees the use of only bounded time-stamps for all B-bounded computations.

References

[1] K.M. Chandy and L. Lamport: Distributed snapshots: Determining global states of distributed systems, *ACM Trans. Comput. Sys.* **3**(1) (1985) 63–75.

[2] R. Cori and Y. Metivier: Approximations of a trace, asynchronous automata and the ordering of events in a distributed system, *Proc. ICALP '88*, Springer LNCS **317** (1988) 147–161.

[3] R. Cori, Y. Metivier and W. Zielonka: Asynchronous mappings and asynchronous cellular automata, *Inform. and Comput.*, **106** (1993) 159–202.

[4] D. Dolev and N. Shavit: Bounded concurrent time-stamps are constructible, *Proc. ACM STOC* (1989) 454–466.

[5] C. Dwork, O. Waarts: Simple and efficient bounded concurrent time-stamping or bounded concurrent time-stamps are comprehensible, *Proc. 24th ACM STOC* (1992) 655–666.

[6] J.G. Henriksen, M. Mukund, K. Narayan Kumar and P.S. Thiagarajan: Towards a theory of regular MSC languages, Report RS-99-52, BRICS, Computer Science Department, Aarhus University, Denmark (1999).

[7] A. Israeli and M. Li: Bounded time-stamps, *Proc. 28th IEEE FOCS* (1987) 371–382.

[8] S. Krishnamurthy and M. Mukund: Implementing Causal Ordering with Bounded Time-stamps, *Report TCS-95-7*, Chennai Mathematical Institute, Chennai, India (1995).

[9] L. Lamport: Time, clocks and the ordering of events in a distributed system, *Comm. ACM* **17**(8) (1978) 558–565.

[10] L. Lamport and N. Lynch: Distributed computing: Models and methods, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science: Volume B*, North-Holland, Amsterdam (1990) 1157–1200.

[11] M. Mukund, K. Narayan Kumar, J. Radhakrishnan and M. Sohoni: Robust asynchronous protocols are finite-state, *Proc. ICALP '98*, Springer LNCS **1443**, (1998) 188–199.

[12] M. Mukund and M. Sohoni: Keeping track of the latest gossip in a distributed system, *Distributed Computing*, **10**, 3, (1997) 137–148.

[13] M. Mukund and P.S. Thiagarajan: Linear Time Temporal Logics over Mazurkiewicz Traces, *Proc. MFCS '96*, Springer LNCS **1113**, (1996) 32–62.

[14] E. Rudolph, P. Graubmann and J. Grabowski: Tutorial on message sequence charts, in *Computer Networks and ISDN Systems—SDL and MSC*, Volume 28 (1996).

[15] 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.

[16] 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.

[17] S. Yadulla: *Global states of distributed systems.* M. Tech thesis, Department of Computer Science and Engg, Indian Institute of Technology Bombay (1999).

[18] W. Zielonka: Notes on finite asynchronous automata. *R.A.I.R.O.—Inf. Théor. et Appl.*, **21** (1987) 99–135.