Causal separators and topological timestamping: an approach to support causal multicast in large-scale systems[1]

Luís Rodrigues
ler@inesc.pt

Paulo Veríssimo
paulov@inesc.pt

Technical University of Lisboa
IST - INESC [2]

Abstract

In recent years there has been a growing interest in developing communication systems that are able to deliver messages respecting potential causality. Unfortunately, causal delivery cannot be provided without costs: extra delays may be induced on message delivery or processes may be required to maintain and exchange records of causal relations. In this paper we present an extension to previous work on compression of causal information using knowledge about the topology of the communication structure.

In order to make practical use of this result, we present a methodology to model the communication system. The technique exploits the physical structure of existing networks, in particular its hierarchical nature, to create a communication graph where causal separators match the underlying physical and administrative organization. We show that this approach can be applied to existing large-scale systems, providing the means for using topological timestamping with negligible overhead.

Contents

Chapter 1

Introduction

In a distributed system, a process is able to obtain a view of the system evolution by exchanging information with other processes. For a given process, the "natural" ordering of messages is an order that respects the cause-effect relations in the system. Thus, it is interesting to develop communication infrastructures that deliver messages in causal order, such that processes can have a correct view of system evolution. Since it is usually impossible to know, *a priori*, which events are causally related, the communication subsystem should take the conservative approach of (partially) ordering all events that are *potentially* related in a causal way (unrelated events are referred to as *concurrent*). When a protocol constrains (by construction) the cause-effect relations in the system and, in consequence, is able to deliver messages ordered according to causality, we say that the protocol enforces *logical precedence*[1]:

> **Logical precedence:** *In a distributed system, in which information is exchanged only by transmitting messages, a message m is said to* precede *or to be potentially causally related to a message n, represented as $m \rightarrow n$, only if: (i) m and n were sent by the same process and n was sent after m or; (ii) m had been delivered to the emitter of n before n was sent or; (iii) x exists such that $m \rightarrow x$ and $x \rightarrow n$.*

Experience has shown [19, 1, 13, 16] that the design of distributed applications can be simplified if messages are received in order of logical precedence. Since extra complexity would be added to such applications, should the communication subsystem not provide causal delivery, several algorithms have been proposed to implement this ordering discipline [8, 20, 1, 5, 13, 7]. Nevertheless, despite its advantages, the use of causal communication has been somehow limited by the overhead incurred by existing implementations. A major cost of protocols that preserve logical precedence is the size of "history" information that needs to be stored and exchanged to maintain causality, specially in large-scale systems where group addressing is used.

In this paper we extend previous results on causal history compression using knowledge on the topology of the communication structure. Our compression technique uses the concept of a *causal separator*, a set of nodes of the communication graph that can be used to filter causal information. An implementation of this optimization is presented. Additionally, we show the applicability of this approach to real-life large-scale

networks, given their hierarchical nature: we present a methodology to model the communication system as a graph, where causal separators that match the underlying physical and administrative organization are clearly identified.

The paper is organized as follows. Related work is surveyed in Chapter 2. Causal separators are introduced in Chapter 3. Our representation of causal histories is presented in Chapter 4. Chapter 5 describes how our implementation can be applied to large-scale systems. The possibility of providing incomplete causal orderings is discussed in section 6. Chapter 7 presents performance results obtained through simulation. Concluding remarks appear in Chapter 8.

Chapter 2

Related work

Early implementations of logical precedence (also described as the "happened before" relation) were based on logical clocks [8], a technique that introduces a systematic delay in message delivery and that orders more messages than those potentially causally related [20]. To avoid the disadvantages of logical clocks, a new set of algorithms has been proposed, where the information required to precisely define causal relations is piggybacked on the messages exchanged. These approaches are based on causal histories or vector clocks [1, 13, 7]. While some work in this area assumed that all messages were addressed to all processes in the system [19, 13], recent systems allow messages to be addressed exclusively to a subset of the existing processes [1, 7, 12]. These subsets may be structured in groups that may sometimes overlap. Group addressing may improve the system performance, preventing processes from spending resources on messages of no concern to them. On the other hand, group addressing increases the complexity of the algorithms required to preserve logical precedence.

An early implementation of causal histories was proposed by Birman and Joseph [1] to implement ISIS's CBCAST primitive. In this implementation, causal histories included the entire messages: every time a message is locally delivered or sent, a copy is added to the local causal history; additionally, the causal history of the sender is piggybacked in every message. The advantage of this technique is that the delivery of a message is never delayed, since a message carries all the preceding messages with it. The algorithm used in PSYNC [13] is very similar to that of CBCAST. However, since PSYNC is built on top of a reliable transport layer, only message identifiers are stored in the causal history (additionally, PSYNC optimizes the size of the message timestamps). Vector clocks are a particular representation of causal histories, where message identifiers are stored in a vector, and where each entry is allocated to a specific process. Nevertheless, the size of vector clocks still grows linearly with the number of processes [3]. For instance, the protocol in [23] uses a vector clock per each group of processes; however they have shown that if the graph of groups contains cycles, a process has to keep (and exchange) the vectors from all groups. Thus, it is important to use techniques to reduce the size of vector clocks. Unfortunately, there is a trade-off between the amount of information that needs to be maintained to preserve causality and the degree of concurrency that is achieved [23, 12]. In the next chapter we introduce a technique that uses topological information, exploiting points were messages are already (physically) serialized, to reduce the timestamp size with negligible impact on the system parallelism.

Chapter 3

Causal separators

It has been shown that when communications graphs have a process that acts as a gateway, it is possible to decrease the amount of information required to determine temporal relationships [10]. Optimizations based on the communication patterns of processes have also been presented in [23]. Our paper extends these results to arbitrary communication structures, making the following contributions:

- we show that, even in graphs that contain cycles, it is possible to reduce the size of the information exchanged by defining *causal separators*, a set of nodes of the communication graph that can be used to filter causal information.

- we present a methodology to model the communication system such that one can make practical use of the previous result.

These two aspects will be dealt with considerable detail in the next two chapters. In the present chapter, we provide a global overview of our approach.
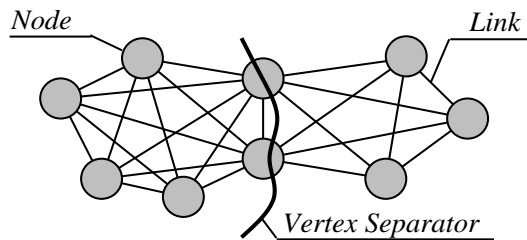

Figure 3.1: Vertex Separators.

We assume that each process is able to communicate directly only with a given subset of system processes $\mathcal{L}_p \subseteq \mathcal{P}$. Two processes not directly linked can communicate inderectly through a chain of packets (usually, automatically created by a routing algorithm). The complete communication topology can thus be represented by a graph $G(\mathcal{P}, E)$, where processes are the vertices and the communication links between them the edges: there is an edge incident to $\{p, q\}$ if p can send messages to q. We assume that the graph is connected. A set of processes, S, is called a (F_S, B_S) vertex separator, where the sets F_S and B_S are called, respectively, *forward* and *backward* sets, iff $F_S \cap B_S =$

$\emptyset \wedge F_S \cup B_S \cup S = \mathcal{P}$ and $\forall f \in F_S, \forall b \in B_S$ every path connecting f and b passes through at least one vertex of S. In the context of causal communication, we called such vertex separators *causal separators* (see Figure. 3.1).

In the next chapter we will show that a causal separator can work as a barrier that filters all information concerned with messages exclusively addressed to the backward set and "reported" to all members of the separator ("reported" will be defined precisely). Let a set of processes bounded by one or more causal separators be defined as a *causal zone*. An interesting corollary of the previous observation is that the causal information concerned with the elements of a causal zone may never need to be propagated outside the boundaries of that causal zone, if *reported* to all members of the separator from which a causal relation with the outside is established.

In chapter 5 we will show that it is possible to model the communication system in such a way that causal separators can be effectively used. The technique exploits the physical structure of existing networks, in particular its hierarchical nature, to create a communication graph where causal separators match the underlying physical and administrative organization.

Chapter 4

Extended causal histories

In this chapter we present an implementation of causal histories that allows to reduce the causal information exchanged using information about causal separators.

4.1   Definitions

In the following text we will assume that the system is composed of a collection of processes, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ with disjoint memory spaces. We assume that a unique identifier is associated with each process $p \in \mathcal{P}$ (for convenience, we will use the same notation to refer to the process and its identification). We also assume that an order relation, $\prec$, can be defined between process identifiers. Processes are able to communicate by exchanging messages: the identification of the sender, $s_m \in \mathcal{P}$, and the set of destination processes, $\mathcal{A}_m \subseteq \mathcal{P}$, are always associated with each message, m. We do not impose any restriction on the destination addresses: a message can always be sent to any set of processes in $\mathcal{P}$. Additionally, we assume that each sender assigns a locally generated integer value, c_m, to each message, such that if m is sent before n then $c_m < c_n$ (this can be trivially obtained by using a local counter [1] c_p at each process p). Although the pair (s_m, c_m) uniquely identifies a message, for convenience we define the message's *unique identifier* also including the destination address, that is, $uid_m \equiv (s_m, c_m, \mathcal{A}_m)$.

As in most works in causal multicast protocols, in addition to the *send* event, we introduce two distinct events: *receive* and *deliver*. The receive event identifies the message arrival at a given process and is not visible at the application level. The deliver event identifies the message arrival at the application level. In order to enforce causal delivery, the protocol may have to delay the delivery of received messages. Since we are exclusively concerned with techniques to enforce causal delivery, we assume that the underlying message passing subsystem is reliable, in the sense that messages sent are always received by all correct addressed participants. This assumption is consistent with recent implementations of causal multicast protocols, which are based on a reliable transport layer (for instance, MUTS [16]). We do not make any assumptions about the

---

[1] Since this counter is stored in a bit array with limited capacity, we cannot have indefinitely growing counters. However, techniques exist to overcome this problem [9].

order in which messages are received. Due to the reliable multicast assumption, our causal histories *do not need* to include the messages themselves, but *only* their unique identifier, uid_m. However, for the sake of clarity, when referring to the contents of causal histories we will just use the word "message" instead of the more precise but longer expression "message's unique identifier".

4.2   Using carbon-copies

We now introduce our representation of causal histories. The interested reader will notice that there are no deep fundamental differences between our representation of causal histories and alternative approaches described in the literature [1, 13]. However, it provides the ground for the implementation of optimizations based on causal separators, the main contribution of our work. Our representation of the causal history, that we called *extended causal history*, stores causal information in three different entities:

- a *causal history*, $\mathcal{H}_p$, a list with the messages that precede the next message to be sent by p;

- the *delivery history*, $\mathcal{D}_p$, a list with the messages that have already been delivered at p and;

- a *carbon-copy* history, $\mathcal{C}_p$, that keeps track of to where causal information has been "reported" (the carbon-copy history is used for compression of causal information, and its use will be detailed later).

The delivery history maintains a record of all messages that were delivered to a given process. The causal history maintains a record of all messages that precede the next message to be sent by a given process. Although the causal history contains the delivery history, different compression rules will be later applied to each, thus we decided to explicitly maintain this information in two different entities (explicit separation between causal and delivery histories is also used in other approaches, for instance [15]). These histories are used in the following way:

Every time a message m is sent by process p, it is timestamped with its sender's causal history, $\mathcal{H}_p$. All messages in $\mathcal{H}_p$ are then said to be "*reported*" to all recipients of m. This information is kept in carbon-copy history, $\mathcal{C}_p$. When a message is received by process q, the recipient compares the message timestamp with its own delivery history and checks whether or not all preceding messages have been already delivered locally: it then delivers or delays the received message accordingly. When a message is delivered, the recipient delivery and causal histories are updated accordingly.

More precisely, causal delivery can be enforced using the delivery and causal histories if the following rules are applied (for clarity, we will defer the use of the carbon-copy history until rule 6 is introduced):

R1 (Initial state): When p starts execution, $\mathcal{H}_p$, $\mathcal{D}_p$, and $\mathcal{C}_p$ are empty. Also, $c_p = 0$.

R2 (Timestamping): Before being sent by process p, a new uid is assigned to message m by incrementing the local counter c_p. Next, m is timestamped with p's causal history, that is, $\mathcal{H}_m = \mathcal{H}_p$.

R3 (Causal delivery): On receipt of message m sent by process p and timestamped with a causal history $\mathcal{H}_m$, process $q \in \mathcal{A}_m$ delays the delivery of m until all messages in $\mathcal{H}_m$ that were addressed to q have been delivered at q [2]. More precisely, q delays the delivery of m until the following condition is true: $\forall(i \in \mathcal{H}_m : q \in \mathcal{A}_i)$ i *is-in* $\mathcal{D}_q$; where the *is-in* relation is here defined as: m *is-in* $\mathcal{D} \iff m \in \mathcal{D}$.

R4 (Record maintenance): When process p sends m it atomically adds m to $\mathcal{H}_p$ and to $\mathcal{D}_q$. When a message, m, is delivered at $q \neq s_m$, m's timestamp, $\mathcal{H}_m$, is added to $\mathcal{H}_q$. Additionally, m is added to $\mathcal{H}_q$ and $\mathcal{D}_q$.

Theorem 1: Rules 1-4 enforce message causal delivery (see appendix for proofs).

Rules 1-4 above are enough to enforce causal delivery of messages (see [18] for a proof). However, this solution suffers from a serious drawback as, unless some measures are taken to garbage-collect redundant elements, the causal histories continue to grow indefinitely. In the next paragraphs we present an extra set of rules that allow the garbage-collection of the extended causal history.

We start by compressing the delivery history. The compression rule exploits the fact that messages from the same sender must always be delivered in the order they were sent. From the causal delivery rule, if a message m from process p is delivered to q, then all previous messages from p, addressed to q, were already delivered at q. As a result of this rule, delivery histories do not need to keep more than one message from each sender. More precisely,
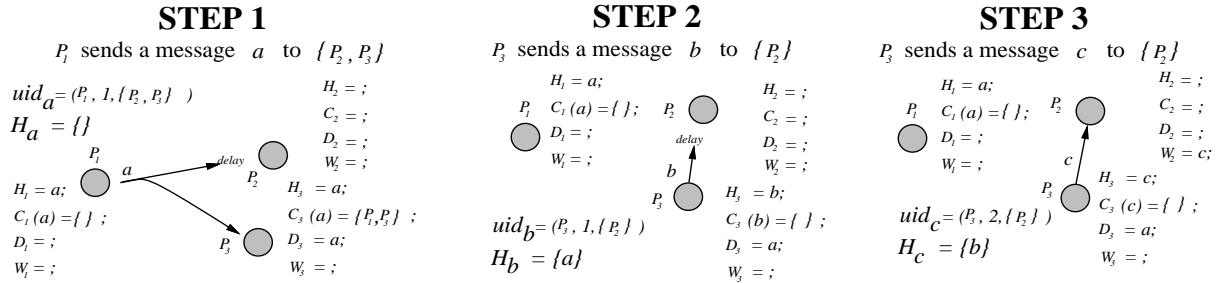

Figure 4.1: An example.

R5 (FIFO Delivery): At most one unique message identifier needs to be stored from each sender in the delivery history. When a message m from process p is added to $\mathcal{D}_q$, m replaces the previous message from p delivered at q.

---

[2] A message can always be delivered without delays to its sender.

Naturally, since some elements are deleted from the delivery history as new members are added, the definition of "is in $\mathcal{D}$" must be slightly changed. We now say that a message m *is-in* $\mathcal{D}$ if and only if there exists a message in $\mathcal{D}$, from the same sender, with a higher or equal identifier. More precisely, m *is-in* $\mathcal{D} \iff \exists_{n \in \mathcal{D}} : c_m \leq c_n$.

Theorem 2: Rules 1-5 still enforce message causal delivery (see appendix for proofs).

We now garbage-collect the causal history. The idea is to remove from this history all the elements not strictly required to preserve causal delivery. In doing so, we discard information about the past. The method is an extension of the "last send" and "last update" vectors proposed by Singhal and Kshemkalyani [21], that were also suggested in [23] to optimize the use of vector clocks on overlapping groups. We simply extend this method to arbitrary addressing schemes. The optimization can be informally presented as follows: before being sent, message m is timestamped with its sender's causal history $\mathcal{H}_p$. It will then be delivered to $\mathcal{A}_m$, after all messages in $\mathcal{H}_m$. Any message $n : \mathcal{A}_n \subseteq \mathcal{A}_m$ that carries m in its timestamp, does not need to carry $\mathcal{H}_m$ since it will be delivered after m, thus after all messages in $\mathcal{H}_m$. However, this requires some bookkeeping of whom the messages were reported to. This information can be kept in an additional history, called the *carbon-copy* history, $\mathcal{C}$. The carbon-copy history contains a field for each message in the causal history, storing the list of processes to which the associated message was already "reported" within the timestamp of another message. The carbon-copy history should be updated using the following rule:

R6 (Carbon-copy): Each process, p, keeps a carbon-copy history, $\mathcal{C}_p$, that contains an element $\mathcal{C}_p(m)$ for each message m in $\mathcal{H}_p$. These elements are used according to the following rules:

R6.1 - Extended timestamping (optional): When a message m is timestamped, in addition to $\mathcal{H}_p$, it may also by timestamped with $\mathcal{C}_p$. We refer to the carbon-copy field of m's timestamp as $\mathcal{C}_m$.

R6.2 - Send update: After sending a message m, and before inserting m in $\mathcal{H}_p$, update all fields of $\mathcal{C}_p$ as follows: $\forall (i \in \mathcal{H}_p)$ let $\mathcal{C}_p(i) = \mathcal{C}_p(i) \cup \mathcal{A}_m \cup \{s_m\}$. Then insert m in $\mathcal{H}_p$ and initialize $\mathcal{C}_p(m) = \emptyset$. These updates should be performed in a single atomic operation.

R6.3 - Deliver update: After delivering a message m, processor $q \neq s_m$ updates the carbon-copy fields of previous messages from the same sender as follows:

$$\forall (n \in \mathcal{H}_q : s_n = s_m \wedge c_n < c_m) \text{ let } \mathcal{C}_q(n) = \mathcal{C}_q(n) \cup \mathcal{A}_m$$

Then, it adds all elements n of $\mathcal{H}_m$ to $\mathcal{H}_q$. The carbon-copy fields of these messages are initialized as follows (if R6.1 is not used, use $\mathcal{C}_m(n) = \emptyset$):

10

$$\mathcal{C}_q(n) = \mathcal{C}_m(n) \cup \mathcal{A}_m \cup \{s_m\} \cup \bigcup_{i \in \mathcal{H}_q : s_i = s_n \wedge c_i > c_n} \mathcal{A}_i$$

If $n \in \mathcal{H}_m$ is already in $\mathcal{H}_q$ it just updates the existing carbon-copy field, merging $\mathcal{C}_q(n)$ with the result of the previous expression. Finally, q inserts m in $\mathcal{H}_q$ and initialize $\mathcal{C}_q(m) = \{s_m, q\}$. These updates should be performed in a single atomic operation.

As noted above, messages may also be timestamped with their sender's carbon-copy history, improving the accuracy of the contents of all carbon-copy histories. This is achieved at the cost of increasing message sizes. Thus, there is a tradeoff between the efficiency of the garbage collection and the size of the message's timestamps. When increasing the size of the timestamps is undesirable, the carbon-copy history can be maintained using exclusively information concerning messages locally sent and received. The carbon-copy history is used to compress the causal histories in the following way: (1) messages do not have to be included in a timestamp, if they have already been included in a timestamp of another message sent to the same destination; and (2) when the carbon-copy field of a message completely includes the message's address, that message can be safely removed from the causal history as it has already been reported to all relevant processes. More precisely,

R7 (Timestamp Redundancy): When timestamping a message m, processor p only includes in $\mathcal{H}_m$ the elements of its causal history $i \in \mathcal{H}_p$ not reported to $\mathcal{A}_m$, according to p's knowledge, i.e.: $\mathcal{H}_m = \bigcup i \in \mathcal{H}_p : \mathcal{A}_m \nsubseteq \mathcal{C}_p(i)$.

R8 (History Redundancy): In a causal history, $\mathcal{H}_p$, if there exists a message, m, such that $\mathcal{A}_m \subseteq \mathcal{C}_p(m)$, m can be removed from $\mathcal{H}_p$ and $\mathcal{C}_p$.

Theorem 3: Rules 1-8 enforce message causal delivery (see appendix for proofs).

In Figure 4.1 we give a small example that illustrates the use of garbage collection rules. We consider three processes, P_1, P_2, P_3. The figure shows the contents of their causal, carbon-copy, and delivery histories. The list of messages waiting to be delivered at process P_i is denoted W_i. P_1 sends a message a to $\mathcal{A}_a = \{P_2, P_3\}$ which is received and delivered at P_3 but not received at P_2 due to a network delay. Note that due to rule R6.3, $\mathcal{C}_3(a) = \{P_1, P_3\}$. Next, P_3 sends a message b to P_2 which is also delayed by the network. Note that due to R6.2, $\mathcal{C}_3(a) = \{P_1, P_3\} \cup \{P_2\} = \{P_1, P_2, P_3\}$. Since, $\mathcal{A}_a \subset \mathcal{C}_3$, according to R8, message a is removed from $\mathcal{H}_3$ and $\mathcal{C}_3$. Next, P_3 sends another message c to P_2. Message c will be queued at P_2 until $b \in \mathcal{H}_c$ is delivered (which in turn, must wait for a).

4.3 Topological timestamping

Causal separators can be exploited to reduce the size of message timestamps as follows. When a member of the causal separator timestamps a message addressed to processes exclusively located in the forward set, it can omit in the timestamp all elements of its causal history that were addressed exclusively to members of the backward set and that were already reported to the other members of the causal separator. More precisely,

> **R9** (Topological timestamp): Processor p is sending a message m. All messages $n \in \mathcal{H}_p$ for which exists a (F_S, B_S) causal separator[3], S, such that: $p \in S \wedge \mathcal{A}_m \subseteq F_S \wedge \mathcal{A}_n \subseteq B_S \wedge S \subseteq \mathcal{C}_p(n)$, do not need to be inserted in $\mathcal{H}_m$

> **Theorem 4:** Rules 1-9 enforce message causal delivery (see appendix for proofs).

The compression achieved with the topological timestamping rule can be further improved at the cost of reporting causal information to all the members of the causal separator. In fact, remember that the carbon-copy fields can always be forced to a given desired value just by sending a message to the relevant processes.

There are a number of difficulties associated with the use of our topological timestamping scheme. In first place, an arbitrary network can have a large number of causal separators: topological timestamping can be applied to all separators or just to a subset of them.

In second place, causal separators need to be computed before the topological timestamping rule can be applied. There are a number of algorithms to identify vertex separators in a graph (for instance, see [4]). However, these can be too expensive to be executed frequently during normal system operation. Thus, our method is better suited for applications where the topology is relatively static or can be computed at compile or configuration time. In this case causal separators may be computed in advance, and the correspondent S, F_S and B_S sets be prepared and loaded on all causal separator members to allow a fast execution of the topological timestamping rule. A particular case of a relatively static topology, is the one defined by the network infrastructure that connects individual nodes of a distributed system. This is a sufficiently important case – in fact, this special case gave us the motivation for this work – thus, it will be dealt with in its own section, later in the paper. The other difficulty associated with the use of causal separators is that any change to the topology can alter the membership of the causal separators. Thus, special care must be taken when processes join the system. This issue will be the concern of the next paragraphs.

4.4 Handling of joins

We considered that the failure of a process is permanent and that if the process later recovers it is considered a new process. Thus, there is only one scenario where the failure

---

[3]Where F_S and B_S are respectively the forward and backward sets of the separator(see chapter 3)

of a process may endanger the delivery of messages according to causal precedence. This happens when the failed process is the unique member of a causal separator and its failure disconnects the associated backward and forward set. In this case, when both sets are reconnected, through the creation of a new causal separator, it must be ensured that none of the precedence information that was stored at the old causal separator is lost. Thus, we distinguish four different scenarios for a joining process:

- (i) The joining process does not modify existing causal separators in the graph.

- (ii) The joining process creates a new causal separator connecting two sub-graphs never connected in the past.

- (iii) The joining process expands an already existing causal separator, but does not creates new separators.

- (iv) The joining process creates a new causal separator connecting two sub graphs that have been connected in the past by another causal separator.

We will analyse the last two cases. The first and second cases are trivial as causal separators are not created or modified (thus, the new process is not required to perform any special action). In the third case, the process is required to obtain the causal history of the other separator members before starting to receive and send causal messages. For sake of simplicity, we assume that in each separator, the membership changes are performed in a serial way, according to some distributed membership algorithm (for a survey, see [11]). Our approach requires the membership protocol to be slightly expanded in order to provide to each joining member the causal history of all the current members of the separator. Finally, in the fourth case, when two sub-graphs are reconnected after a complete failure of a previous separator, three solutions are possible:

- Require the new process to perform a global checkpoint to obtain the causal histories of all members of the backward set. This method may be too expensive to be applied in practice.

- Provide each process in the separator with a Uninterruptible Power Supply unit (as used in [7]) or with non-volatile RAM. This would allow the causal history of the separator to be preserved in stable memory. In this case, the two causal zones could only be reconnected by a process able to read the stable memory of the last member to fail (using for instance the method introduced by Skeen [22]) in order to obtain the causal history of the previous separator.

- Prevent this case from happening, by ensuring that causal separators contain at least $f + 1$ members, in order to tolerate f faults. This could be done by adding members to the causal separator, whose purpose would be exclusively to store the separator's causal history. Those extra members would act as *witnesses* of the traffic flowing through the separator. In this case, the backward and forward sets could be reconnected by any process able to load the causal history from a witness process.

4.5 Further improvements

There are a number of techniques that allow further compression of timestamps. Most of these techniques can also be applied to our extended causal histories. For instance, causal histories can be additionally compressed using information about the *delivery* of messages: if a message is known to be already delivered to all participants (*fully delivered*), its identifier can be deleted from the causal history. In synchronous systems, the passage of time can provide assurance of message delivery. For instance, in the xAMp [17], there is a well-known worst case dissemination time, Δ. Since every message is *fully delivered* within Δ time, this interval is also used to remove message identifiers from causal histories. Another way to obtain information about message delivery is to analyse incoming timestamps: a message M is fully delivered if its identifier is received in a timestamp coming from all of M's recipients [7]. It is interesting to notice that these tests are dual, respectively, to the Δ and *Fifo* stability tests for synchronised and logical clocks [19] (although in the context of causal histories they are not used to reduce message latency but only to save storage).

In this chapter we have shown that an extra carbon-copy history can be used to garbage-collect timestamps. This was attained at the cost of higher storage costs, since auxiliary information needs to be maintained at every process (in our case, the carbon-copy information). Related approaches, suffer from the same drawback [21, 23].

Chapter 5

Using the communication topology

Until now, we have used the term *processes* to identify our computational entity in a quite generic way. A possible implementation could apply the algorithm to the communication between real (operating system) processes. In such cases, the communication graph would reflect the application-level structure.
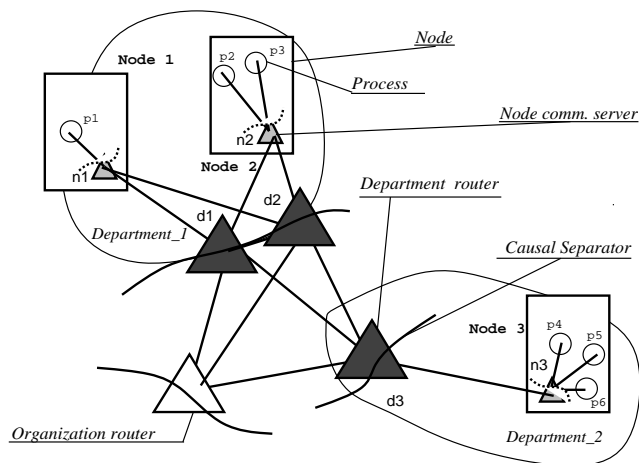

Figure 5.1: Communication architecture.

Consider however how a large-scale network is set up. The view of a fully-connected graph among all processes in the system is an abstraction that matches the service but not the structure of the underlying network. With regard to this, the global network structure[1] has very important features: (i) it is not homogeneous, featuring, besides the mesh of long-haul point-to-point links, technologically interesting "modules" such as LANs, MANs and, still to come, ATM fabrics; (ii) most of the sites are located inside private, organization-specific domains, which are often separated from the rest of the network by private front-end routers or gateways; (iii) in large organizations, this separation may be enforced again at department level; (iii) in many systems, there exists in each node a special communications process, driver or dedicated communication board responsible for all interactions of that node with the network. When this is the case, all messages from processes on the same node are naturally

---

[1] Called Internet in the case of the TCP/IP based internetwork.

serialized when they are physically transmitted. Such a scenario is depicted in Figure 5.1. This structure will last long enough to deserve some systematization, which will hopefully yield more efficient inter-process communication in the large [25].

As an example, we propose the following structuring principles: the global network is a WAN-of-LANs; the WAN is generally uncontrollable from the viewpoint of algorithms – only standard protocols may be run by the WAN routers (e.g. IP); an exception is made for the part of it behind the border of an organization – department routers can run specific protocols; organization routers can also run specific protocols, running over the standard protocols of the global WAN (e.g. Multicast IP). Based on this realistic structure, we propose a solution for the provision of causal communication in large-scale systems based on the following methodology:

- the communication entity that connects a node to the network assumes the role of a router process in the communication structure. This process is a causal separator that connects the machine to the network.

- routing of messages between nodes is also implemented by special processes, usually placed in dedicated router nodes, able to execute topological timestamping as they forward the messages.

Using this methodology, one can build an *extended communication graph* that takes into consideration the organization of the underlying infrastructure. In this graph, new nodes are added, corresponding to communication server and router processes (see an example in Figure 5.2). Promoting the communication entities to nodes of the (extended) communication graph has several advantages:
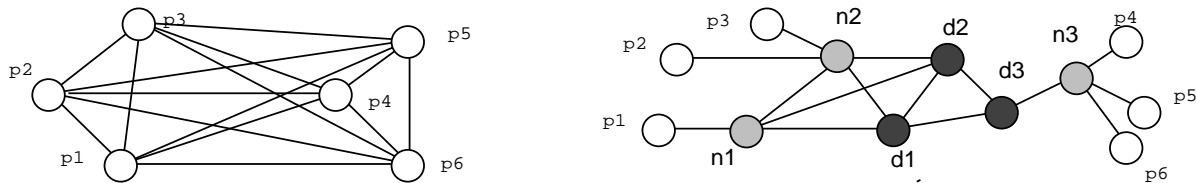
Logical (application-level) communication graph.　　*Extended communication graph.*

Figure 5.2: Mapping the communication structure onto the communication graph.

- It provides a useful way of mapping the abstract communication graph on to the existing physical and administrative communication structure. This gives the means for the practical and useful identification of causal separators (for instance, node's communication server, department and organization routers, etc.).

- The physical communication structure (i.e., the servers and routers, not the application processes) is usually relatively static. This feature provides the basis for an efficient exploitation of topological timestamping and makes the costs associated with dynamic changes of the structure almost negligible.

- It provides a practical way of exploiting communication locality. Using topological timestamping on the structure obtained by adding communication and router

processes, it is possible to prevent local information from being propagated on the network. Messages exchanged between processes of the same node are filtered out by the node router, messages exchanged between processes in the same department by the department router, and so on.

Of course, there are also a number of disadvantages related to this approach. However, we believe that these are minimal and will become even less important as the technology advances, as discussed below.

- The communication servers and routers must implement the topological timestamping rules. Thus, specialized routers need to be used and this may be a disadvantage compared with approaches based on standard communication entities. However, "custom" routers are within the reach of engineers and organizations today: communication protocols (TCP/IP, ISO) are accessible on machines with excellent price/performance ratio. On the other hand, the IP task forces are today very active in the study of new routing protocols, group management and high quality-of-service protocols. Some of that work is based on structuring principles akin to those just mentioned above, so we believe that proprietary "closed" routers will evolve in this direction as well.

- By adding processes to the communication structure, the degree of concurrency of the system is reduced. It should be noted that this theoretical limitation of parallelism has little practical impact, since new processes are being added in places where messages are physically serialized. Considering that the error rate of modern communication links is becoming smaller, the probability of a message being forced to "wait" for another non-causally related message will be almost negligible.

- Routers may become a bottleneck in the system, and topological timestamping increases the overhead of computation that needs to be performed to route each message. However, the processing power has being growing incredibly fast (and becoming cheaper) in the last years. Thus, the cost/benefit ratio of having powerful machines dedicated to routing of messages will decrease in the future.

In conclusion, we believe that the physical and administrative organization of the physical communication structure can be exploited to support the efficient implementation of topological timestamping. In order to achieve this goal, some communication elements must be promoted to nodes of the communication graph. Although this method reduces the parallelism of the abstract communication structure it has little practical impact, as it matches the properties of existing networks [2]. Thus, this technique can be a viable alternative to support causal communication in large-scale systems.

---

[2]Naturally, it should always be possible to bypass the causality mechanisms for applications not wanting to pay the overhead.

Chapter 6

Incomplete orders

Ordering all messages that are *potentially* causally related usually involves ordering more messages that those that actually need to be ordered. In a real system, only a subset of potential relations of causality are transformed, by the system evolution, in effective relations of cause and effect. In many systems, relations between processes have restrictions that result from the way the system was designed and that can be known *a priori*. In order to provide a higher degree of concurrency and parallelism to the communication system, some protocols use knowledge about the way the applications are built to avoid imposing ordering constraints on all messages in the system. This way, messages are split in subsets, which are ordered independently, in what is called an *incomplete order* [24]. Another reason to support incomplete orders is that they allow the implementation of different levels of priorities in communications. Messages of the same priority could be delivered in causal order, but a message of high priority would not need to be delayed by a low-priority message.

It is interesting to notice that in earlier systems where incomplete orders were provided, communication was confined within *groups* [1, 17] or *conversations* [13], that were used as a basis to avoid global total order, global causal order and broadcast addressing. Recent systems [2, 14] tend to use different techniques to provide incomplete causal and total orders.

The simplest way of providing incomplete causal orders is to use closed groups, i.e., to enforce order only on messages exchanged between a subset of processes. This is equivalent to running several instances of any of the previously mentioned protocols, one for each group. However, the impossibility of establishing causal delivery between messages sent in different groups reduces the interest of this approach. The most user-friendly technique consists of assigning *labels* to each message and then enforcing the ordering discipline only on messages with the same label. However this is not always effective. When vector clocks or causal histories are used in practice, this method requires a context to be stored for each label in use, which may consume more resources than desirable. Additionally, it may be very difficult to determine, *a priori*, which label should be used. Another approach is to delegate to the upper layer the task of manipulating causal histories, as suggested in [7]. This seems to be the most versatile approach. This approach can be applied to any kind of causal histories, as long as they can be manipulated as an abstract data type exporting a single "merge" operation. Additionally, it does not suffer from the problem of spending more system

resources, since a single history needs to be maintained at each process. Due to this reason, we believe this is the best way to provide incomplete orders in a system using our extended causal histories.

Chapter 7

Performance

In order to evaluate the performance of our approach we resorted to simulation. For that purpose we used the MIT LCS Advanced Network Architecture group's network simulator, NETSIM [6]. We measured the average size of timestamps obtained using our extended causal histories, with and without applying R9 (topological timestamp). We then compared these values with the size required by a non-optimized clock-matrix approach [15].

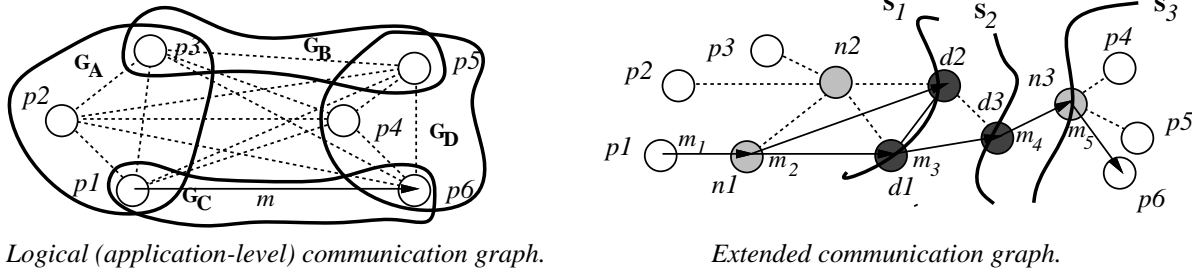*Logical (application-level) communication graph.*   *Extended communication graph.*

Figure 7.1: Topology used in the simulations.

The results presented here were obtained using the same network as in Figure 5.2. At application level, four communication groups are defined, as illustrated in Figure 7.1. Processes can send and receive messages addressed to the groups they belong (for instance, process p_1 can send messages to groups G_A e G_C). A message sent in the logical graph is mapped into a chain of messages in the extended communication graph (for example, message m sent at the logical level from p_1 to $G_C = \{p_1, p_6\}$ is mapped on a chain of messages, m_1, m_2, m_3, m_4, m_5, with the following addresses, $\mathcal{A}_{m_1} = \{n_1\}$, $\mathcal{A}_{m_2} = \{d_1, d_2\}$, $\mathcal{A}_{m_3} = \{d_3\}$, $\mathcal{A}_{m_4} = \{n_3\}$, $\mathcal{A}_{m_5} = \{p_6\}$). Extended causal histories are maintained for the messages exchanged in the extended communication graph.

In this example, with 6 processes, a non optimized matrix clock would require $6x6 = 36$ elements in the history. If a vector clock was kept for each group, the size of timestamps would be of $3 + 3 + 2 + 2 = 10$ elements. We simulated our method and measured the average size of message timestamps. The results presented in table 7.1 were obtained using the following parameters: message rate of $10 msg/s$ at each process and average network delay of $50 ms$. As it can be seen, even without the topological

Scenario	$n = 6$	$n = 10$
Matrix clock	36	100
Vector clocks	10	14
Extended causal histories		
No R9	3.55	3.46
R9 at S_2	2.70	3.09
R9 at S_1, S_2, and S_3	2.10	2.76

Table 7.1: Average timestamp size.

timestamping rule, the average size of message timestamps is much smaller than the obtained with non-optimized versions of the matrix/vector clock approaches. In this scenario, applying the topological timestamping rule at causal separator $S_2 = \{d_3\}$ improves the results by approximately 20%. Applying the rule also at separators $S_1 = \{d_1, d_2\}$ and $S_3 = \{n_3\}$ reduces the average to 2.1 elements per timestamp. Figure 7.2 presents an histogram of the causal history sizes at selected nodes of the network (more precisely, nodes p_3, d_2, and d_3) for the three scenarios. A histogram for message timestamp sizes for the three scenarios is given in Figure 7.3.

The table also shows values obtained with a network where two more members were added to group G_A (at node n_1) and another two to group G_D (at node n_3), resulting in a network of 10 nodes overall. A histogram for message timestamp sizes for the three scenarios with this new topology is given in Figure 7.4. As it can be seen, the average size of message timestamps increases comparatively less than with other approaches. We have experimented the same topology with different network delays without obtaining significantly different values (less than 5% difference).

The advantages of causal separators are more evident if, instead of measuring average values on the global system, we measure the impact of the technique on local communication. Consider for instance the network of figure 7.5, where a small group of nodes interacts with a larger group through a single gateway. Without using topological timestamping, the causal information from G2 is propagated to to group G1, inducing an average timestamp size of 32 elements. If the gateway is used as a separator, the average timestamp size of group G1 is reduced down to less than 5 elements which corresponds approximately to the influence of the local group elements plus that of groups G3 and G4.

The interesting feature of causal separators is the the same type of gains can be achieved if more than one gateway exists. Consider now the network of figure 7.6, consisting of three interconnected broadcast networks. If the communication infrastructure is constructed in such a way that, in each broadcast network, both gateways receive all out-going traffic (since these are connected to a broadcast network this can be achieved with negligible cost), the gateways can filter causal associated with local communication.

The impact of applying topological timestamping is this case is illustrated in figure 7.7. The figure shows the distribution of message timestamp size in group $G1$

with and without separators. The figure shows also the distribution of the host history size on process $p1$ with and without separators. It can be seen that, without topological timestamp, the causal history of process $p1$ is much larger, due to the un-filtered information regarding remote groups that needs to be stored locally.
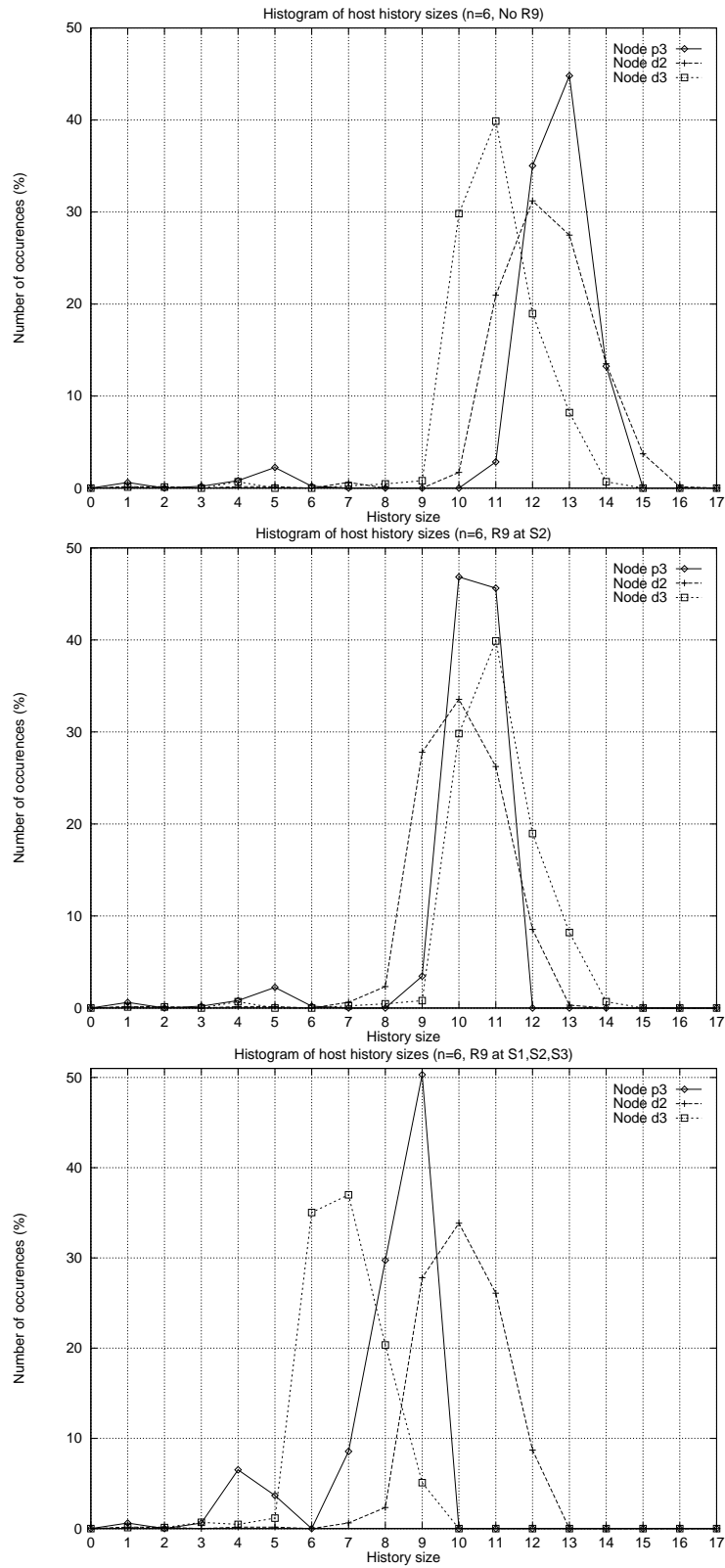
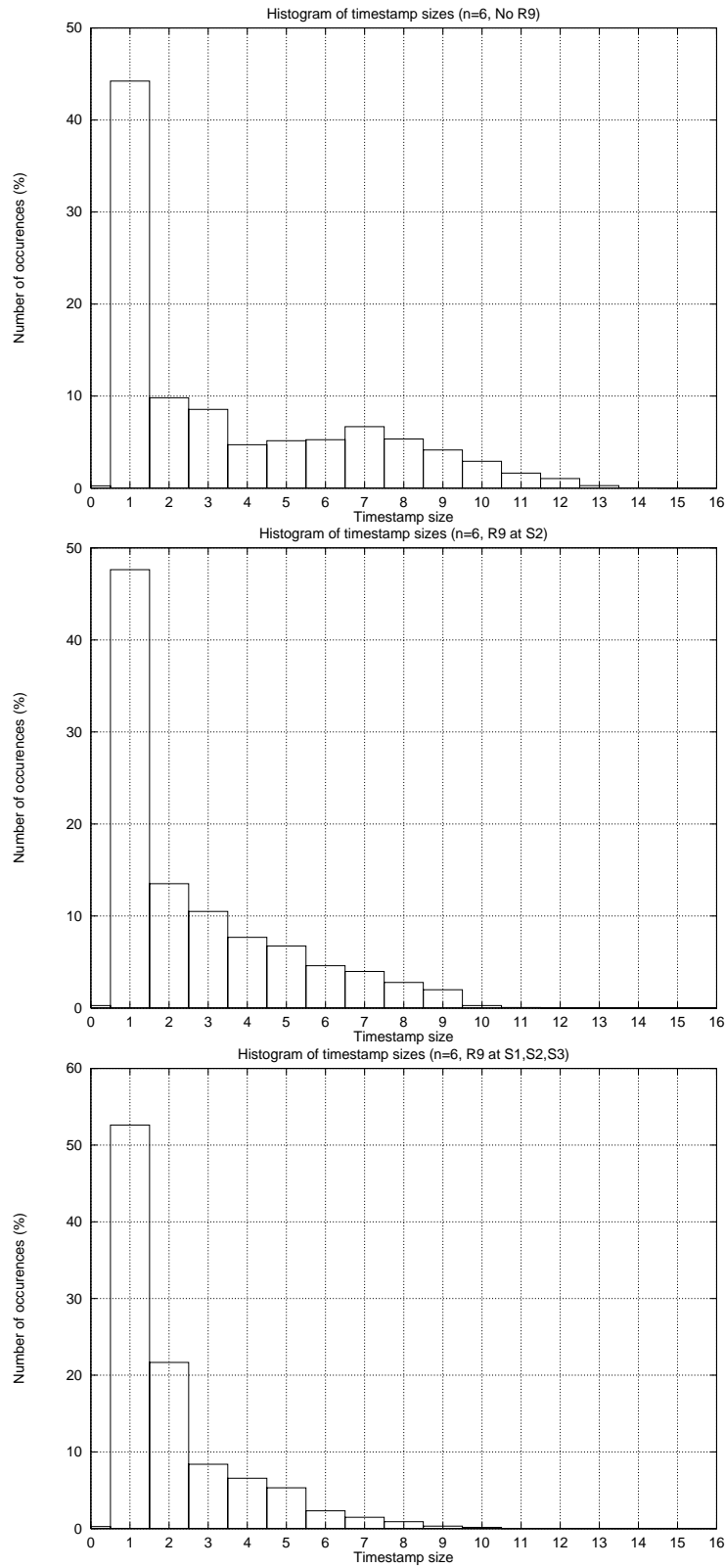Figure 7.2: Hosts history size (topology with 6 processes).

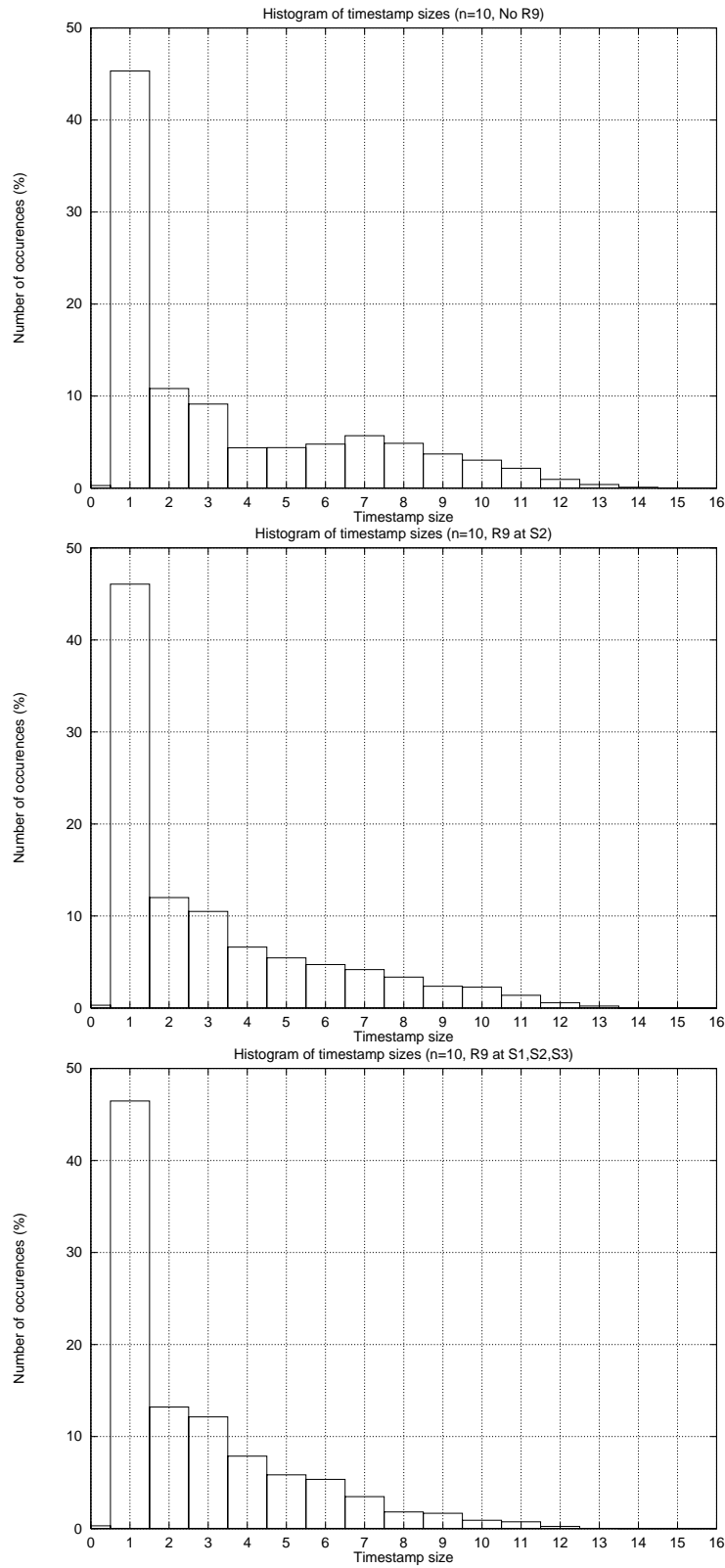Figure 7.3: Message timestamp size (topology with 6 processes).

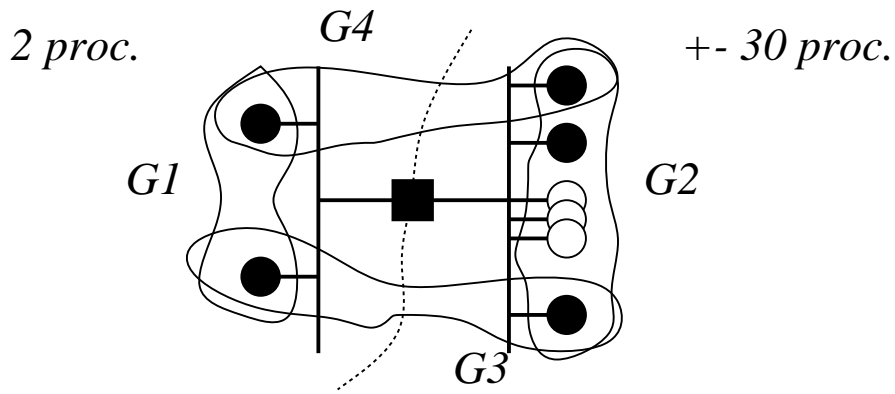Figure 7.4: Message timestamp size (topology with 10 processes).

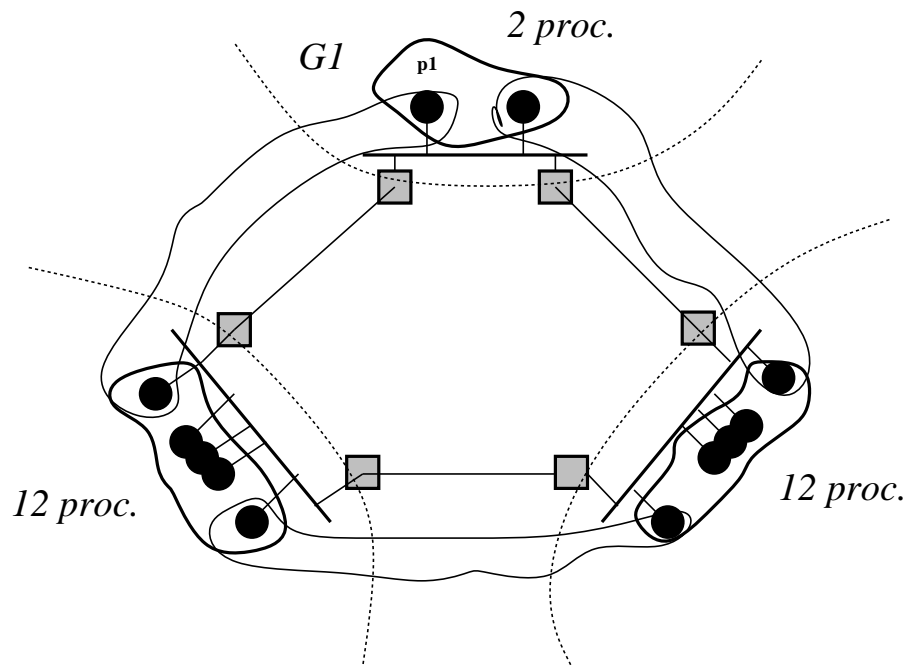Figure 7.5: A small group which interacts with a large group.

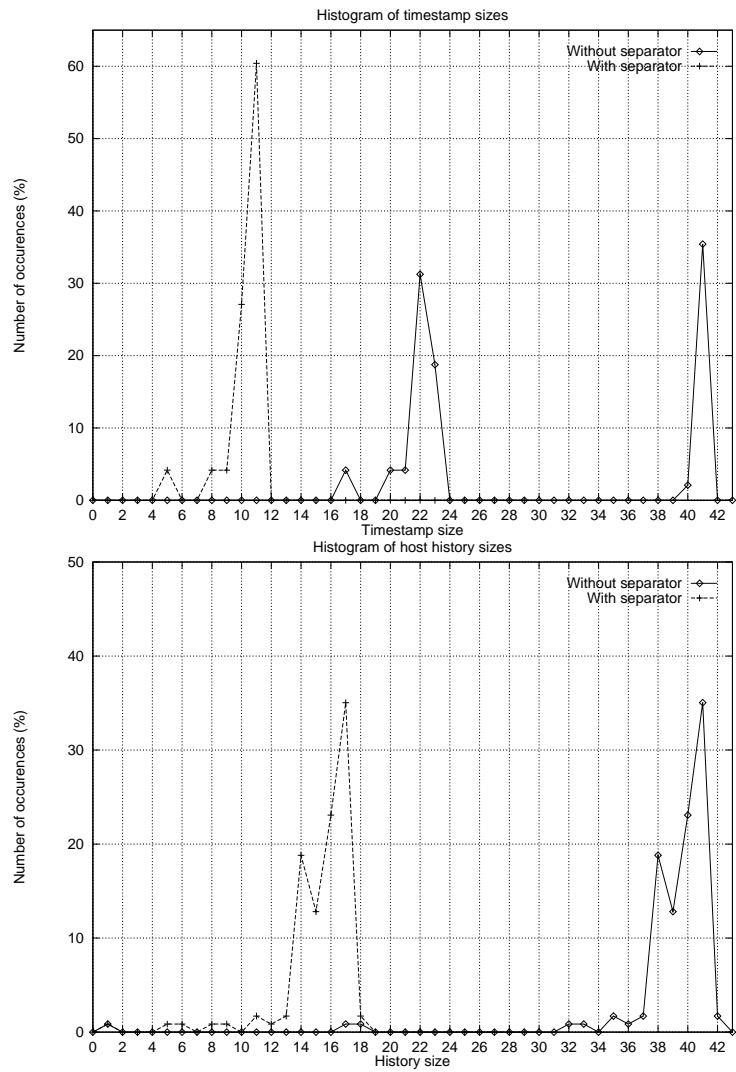Figure 7.6: Three interconnected broadcast networks.

Figure 7.7: Impact on group $G1$.

Chapter 8

Conclusions

In this paper, we have studied the problem of ensuring causal delivery in large-scale systems. We started with a generic representation of causal histories that does not impose restrictions on message addressing. Then, we showed that this representation could be used to improve previous results on compression of causal histories using knowledge about the topology of the communication structure. The technique is based on identifying *causal separators*, i.e. vertex separators of the communication graph, and filtering the causal information that crosses their elements. In order to make practical use of this result, we have presented a technique that exploits the physical structure of existing networks, in particular its hierarchical nature, to create a communication graph where causal separators match the underlying physical and administrative organization. We have shown that this approach can be applied to existing large-scale systems, providing the means for using topological timestamping with little practical overhead.

Acknowledgements

Appendix A

Proofs of theorems

A.1 Overview

This appendix provides the proof for some of theorems presented in the report. It is intended to be relatively self-contained. Although it is assumed that the report has been previously read, for the convenience of the reader we repeat here the relevant definitions and rules required to follow the proofs.

For sake of conciseness, we do not present the proofs of all theorems. However, we do present the proof for what we believe to be the most innovative contribution of our work, the use of causal separators to compress causal information. In this proof, we deliberately remove all rules that compress causal histories except the topological timestamping rule. This substantially reduces the size of the proof and highlights the main contribution of our work. We intend to present the complete proofs in an future version of the report.

A.2 Assumptions

For self containment, we repeat here assumptions made about our system. We assume that the system is composed of a collection of processes, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ with disjoint memory spaces. We assume that a unique identifier is associated with each process $p \in \mathcal{P}$ (for convenience, we will use the same notation to refer to the process and its identification). We also assume that an order relation, $\prec$, can be defined between process identifiers. Processes are able to communicate by exchanging messages: the identification of the sender, $s_m \in \mathcal{P}$, and the set of destination processes, $\mathcal{A}_m \subseteq \mathcal{P}$ are always associated with each message, m. We do not impose any restriction on the destination addresses: a message can always be sent to any set of process in $\mathcal{P}$. Additionally, we assume that each sender assigns a locally generated integer value, c_m, to each message, such that if m_1 is sent before m_2 then c_{m_1} is less than c_{m_2}.

Three local events are relevant to us:

- *Send* event, denoted by SEND(m). Send is a local event that occurs at a single process, the sender of m. We could explicitly indicate the sender node, SEND$_p(m)$ where $p = s_m$ for instance. However, since this information is redundant we simply omit the index.

- *Receive* event, denoted by REC$_q(m), q \in \mathcal{A}_m$, that represents the reception of a message m at a node q. To avoid violation of causal order, the *delivery* of the message to the application can be delayed.

- *Deliver* event, denoted $DELIV_q(m), q \in \mathcal{A}_m$, that represents the delivery of message m to the application at node q.

We do not assume the existence of a global time frame. Thus, we don't assume the pre-existence of any mechanism able to order events that occur at different processes. However, we assume that there is a total order on the events that occur in the same process.

Definition 1 *Let A and B be two local events at some process p. We define an order relation, denoted $<$, between two local events as follows: $A < B$ iff A occurs before B at p.*

We further assume the existence of a reliable multicast primitive with the following property:

Assumption 1 *If $SEND(m)$ occurs, then, at all processes $q \in \mathcal{A}_m$, eventually either $REC_q(m)$ or q fails.*

Using the local order relation, we define *logical precedence* as follows:

Definition 2 *Let $SEND(m)$ denote the sending of message m at s_m. Let $DELIV_p(m)$ denote the delivery of m at some process p. We define* direct logical precedence, *denoted $\hookrightarrow$, as:*

$$m \hookrightarrow m' \stackrel{\mathrm{def}}{=} SEND(m) < SEND(m') \vee DELIV_p(m) < SEND(m')$$

We further define logical precedence, *denoted $\rightarrow$, as the transitive closure of $\hookrightarrow$.*

Definition 3 *We say that a set of rules ensure causal delivery, if the following condition is verified:*

$$\forall_{m,m'} : m \rightarrow m' \Rightarrow DELIV_p(m) < \forall_p : p \in \mathcal{A}_m \cap \mathcal{A}_{m'} \ DELIV_p(m')$$

Our representation of the causal history, that we called *extended causal history*, stores causal information in three different entities:

- a *causal history*, $\mathcal{H}_p$, a list with the messages that precede the next message to be sent by p;

- the *delivery history*, $\mathcal{D}_p$, a list with the messages that have already been delivered at p and;

- a *carbon-copy* history, $\mathcal{C}_p$, that keeps track of to where causal information has been "reported" (the carbon-copy history is used for compression of causal information, and its use will be detailed later).

The delivery history maintains a record of all messages that were delivered to a given process. The causal history maintains a record of all messages that precede the next message to be sent by a given process. Although the causal history contains the delivery history, different compression rules will be later applied to each, thus we decided to explicitly maintain this information in two different entities (explicit separation between causal and delivery histories is also used in other approaches, for instance [15]). These histories are used in the following way:

Every time a message m is sent by process p, it is timestamped with its sender's causal history, $\mathcal{H}_p$. All messages in $\mathcal{H}_p$ are then said to be *"reported"* to all recipients of m. This

30

information is kept in carbon-copy history, $\mathcal{C}_p$. When a message is received by process q, the recipient compares the message timestamp with its own delivery history and checks whether or not all preceding messages have been already delivered locally: it then delivers or delays the received message accordingly. When a message is delivered, the recipient delivery and causal histories are updated accordingly. The relevant rules to update these histories are presented in the sections were they are used.

A.3 Proof of theorem 1

For self containment, we repeat here the rules that are used for Theorem 1.

R1 (Initial state): *When p starts execution, $\mathcal{H}_p$, $\mathcal{D}_p$, and $\mathcal{C}_p$ are empty. Also, the local counter c_p is set to zero, i.e. $c_p = 0$.*

R2 (Timestamping): *Before being sent by process p, a new uid is assigned to message m by incrementing the local counter c_p. Next, m is timestamped with p's causal history, that is, $\mathcal{H}_m = \mathcal{H}_p$.*

R3 (Causal delivery): *On receipt of message m sent by process p and timestamped with a causal history $\mathcal{H}_m$, process $q \in \mathcal{A}_m$ delays the delivery of m until all messages in $\mathcal{H}_m$ that were addressed to q have been delivered at q [1]. More precisely, q delays the delivery of m until the following condition is true:*

$$\forall_{I \in \mathcal{H}_m : q \in \mathcal{A}_I} : I \text{ is-in } \mathcal{D}_q$$

where the is-in relation is here defined as:

$$m \text{ is-in } \mathcal{D} \iff m \in \mathcal{D}$$

R4 (Record maintenance): *When process p sends m it atomically adds m to $\mathcal{H}_p$.*

When a message, m, is delivered at q, m's timestamp, $\mathcal{H}_m$, is added to $\mathcal{H}_q$. Additionally, m is added to $\mathcal{H}_q$ and $\mathcal{D}_q$.

Lemma 1 *When rules R1 to R4 are used, m is-in $\mathcal{D}_p$ only if m has been delivered at p.*

Proof: The proof follows directly from the rules that update the delivery history. There is only one rule that allows a message to be added to the delivery history, this is rule R4. As stated in the rule, the message is added to the delivery history only after being delivered. □

Lemma 2 *When rules R1 to R4 are used, $m \in \mathcal{H}_{m'}$ only if $m \to m'$.*

Proof: We consider first that $s_m = s_{m'}$ and then $s_m \neq s_{m'}$.

$s_m = s_{m'}$: m and m' were sent by the same process p. According to rule R2, m must be already in $\mathcal{H}_p$ when m' is sent, in order for $m \in \mathcal{H}_{m'}$. By rule R4, messages are never added to causal histories before being sent. Thus, m is not inserted in any causal history before being sent by p. Then, it follows directly that m must have been sent by p before m', or $m \to m'$.

$s_m \neq s_{m'}$: m and m' were sent by different processes. The proof will consist in building a causal chain $m \to \ldots \to m^2 \to m^1 \to m'$. Let m^1 be the first message sent by $s_{m'}$ such

---

[1] A message can always be delivered without delays to its sender.

31

that $m \in \mathcal{H}_{m^1}$ (i.e., either $m^1 = m'$ or m^1 was sent before than m' by the sender of m'). Before sending m^1, $s_{m'}$ must have delivered m^2 such that $m \in \mathcal{H}_{m^2}$. By definition, $m^2 \to m'$. The same reasoning can now be applied to m^2 and so on. In each interaction, a new sender process is introduced, different from all previous. As the number of processes is finite we will finally end up with $s_{m^n} = s_m$, which was considered under case 2.1. Thus the causal chain $m \to m^n \to m^1 \to m'$ has been built. $\square$

Lemma 3 *When rules R1 to R4 are used, if m is delivered at p then m is-in $\mathcal{D}_p$.*

Proof: *According to rule R4, when m is delivered at p it is added to $\mathcal{D}_p$ thus, $m \in \mathcal{D}_p$, or m is-in $\mathcal{D}_p$.*

Theorem 1 *Rules R1-R4 ensure message causal delivery.*

Proof: We prove the correctness of the algorithm in two stages. We first prove safety (i.e., that causal delivery is never violated) and then we prove liveness (i.e., that messages are never delayed indefinitely).

Safety: Consider the actions of a process p that receives two messages m and m' such that $m \to m'$. By rule R3 and Lemma 1, m' will only be delivered after all messages in $\mathcal{H}_{m'}$ addressed to p have been delivered. Thus, we just have to prove that if $m \to m'$ then $m \in \mathcal{H}_{m'}$.

If $m \to m'$ there is a possibly empty series of messages $m^0 \ldots m^k$ such that $m \hookrightarrow m^0 \hookrightarrow \ldots \hookrightarrow m^k \hookrightarrow m'$. We will now prove $m \in \mathcal{H}_{m'}$ by induction.

- Base case: If $m \hookrightarrow m^0$, $m \in \mathcal{H}_{m^0}$.

    - $s_m = s_{m^0}$: $\exists_q : \text{SEND}(m) < \text{SEND}(m^0)$. By rule R4 process q atomically adds m to $\mathcal{H}_q$ after sending it.
    - $s_m \neq s_{m^0}$: $\exists_q : \text{DELIV}_q(m) < \text{SEND}(m^0)$. By rule R4, m is atomically added to $\mathcal{H}_q$ after q delivering it.

    In both cases, by rule R2, m^0 will be timestamped with $\mathcal{H}_q$ (now containing m).

- Inductive step:. If $m^k \hookrightarrow m^{k+1}$ and $m \in \mathcal{H}_{m^k}$ then $m \in \mathcal{H}_{m^{k+1}}$.

    - $s_{m^k} = s_{m^{k+1}}$: $\exists_q : \text{SEND}(m^k) < \text{SEND}(m^{k+1})$. m^k was stamped by q. Thus, in order for $m \in \mathcal{H}_{m^k}$ we must have $m \in \mathcal{H}_q$.
    - $s_{m^k} \neq s_{m^{k+1}}$: $\exists_q : \text{DELIV}_q(m^k) < \text{SEND}(m^{k+1})$. By rule R4, $\mathcal{H}_{m^k}$ is atomically added to $\mathcal{H}_q$ after delivering m^k. Since $m \in \mathcal{H}_{m^k}$, then $m \in \mathcal{H}_q$.

    In both cases, by rule R2, m^{k+1} will be timestamped with $\mathcal{H}_q$ (now containing m^k).

Liveness: Suppose there exists a broadcast message m sent by process p that can never be delivered to process q. From Assumption 1, if q does not fails, eventually $\text{REC}_q(m)$. Thus, according to rule R3, in order for the delivery of m to be delayed at q, there must exist one or more messages $m' \in \mathcal{H}_m : q \in \mathcal{A}_{m'} \wedge \neg(m' \textit{ is-in } \mathcal{D}_q)$. Let m^s be the earliest of those messages according to the precedence relation, $\to$. To be more precise, remember that $\to$ defines a *partial* order. Thus, the "earliest" message according to this relation may not be unique. In fact, there may exist a set of concurrent messages that are the "earliest" according to $\to$. In this case, let m^s be one of those earliest messages. m^s has been sent by some process r and received at q. If delivered, m^s *is-in* $\mathcal{D}_q$, by Lemma 3. In order for m^s delivery to be delayed at q, there must exist one or more messages $m'' \in \mathcal{H}_{m^s} : q \in \mathcal{A}_{m''} \wedge \neg(m'' \textit{ is-in } \mathcal{D}_q)$. However, all messages in $\mathcal{H}_{m^s}$ are earlier than m^s according to $\to$ as proven by Lemma 2. Thus, m^s is not (one of) the earliest messages not delivered at q which shows the contradiction. $\square$

32

A.4   Proof of theorem 2

For theorem 2, the following additional rule is used:

R5 (FIFO Delivery): *At most one unique message identifier needs to be stored from each sender in the delivery history. When a message m from process p is added to $\mathcal{D}_q$, m replaces the previous message from p delivered at q.*

Naturally, since some elements are deleted from the delivery history as new members are added, the definition of "is in $\mathcal{D}$" must be slightly changed. We now say that a message m is-in $\mathcal{D}$ if and only if there exists a message in $\mathcal{D}$, from the same sender, with a higher or equal identifier. More precisely,

$$m \text{ is-in } \mathcal{D} \iff \exists_{N \in \mathcal{D}} : c_m \leq c_N$$

Lemma 4 *When rules R1 to R5 are used, m is-in $\mathcal{D}_p$ only if m has been delivered at p.*

Proof: The proof for Rules R1-R4 still applies. □

Lemma 5 *When rules R1 to R5 are used, $m \in \mathcal{H}_{m'}$ only if $m \to m'$.*

Proof: The proof for Rules R1-R4 still applies. □

Lemma 6 *When rules R1 to R5 are used, if m is delivered at p, m is-in $\mathcal{D}_p$.*

Proof: The proof is by induction.

Base case: According to rule R4, when m is delivered at p it is added to $\mathcal{D}_p$ thus, $m \in \mathcal{D}_p$. According to definition, m *is-in* $\mathcal{D}_p$.

Inductive step: If m *is-in* $\mathcal{D}_p$ before another message m' is delivered at p, m *is-in* $\mathcal{D}_p$ after m' is delivered at p. By definition, if m *is-in* $\mathcal{D}_p$, then $\exists_{m'' \in \mathcal{D}} : c_m \leq c_{m''}$. m' replaces m'' in $\mathcal{D}_p$ iff $s_{m'} = s_{m''}$. However, from the safety proof of theorem 1, $m'' \to m'$, thus $c_{m''}$ is less than $c_{m'}$. Thus, after the replacement we still have c_m less than $c_{m'}$, that is, m *is-in* $\mathcal{D}_p$. □

Theorem 2 *Rules R1-R5 ensure message causal delivery.*

Proof: The safety proof of the theorem is not affected in any way by rule R5. The liveness proof is also very similar: just change reference to Lemma 3 by reference to Lemma 6. □

A.5   Causal separators: proof

To prove the correctness of the causal separator rules, the following additional rules are used (note that all rules not strictly required to this proof are omitted):

R6 (Carbon-copy): *Each process, p, keeps a carbon-copy history, $\mathcal{C}_p$, that contains an element $\mathcal{C}_p(m)$ for each message m in $\mathcal{H}_p$. These elements are used according to the following rules:*

R6.2- Send update: *After sending a message m, and before inserting m in $\mathcal{H}_p$, update all fields of $\mathcal{C}_p$ as follows:*

$$\forall_{I \in \mathcal{H}_p} \ let \ \mathcal{C}_p(I) = \mathcal{C}_p(I) \cup \mathcal{A}_m$$

Then insert m in $\mathcal{H}_p$ and initialise $\mathcal{C}_p(m) = \emptyset$. These updates should be performed in a single atomic operation.

R6.3- Deliver update: After delivering a message m, when processor q adds $\mathcal{H}_m$ to $\mathcal{H}_q$, it should update carbon-copy fields as follows:

$$\forall_{N \in \mathcal{H}_m : N \notin \mathcal{H}_q} : \mathcal{C}_q(N) = \emptyset$$

Then insert m in $\mathcal{H}_q$ and initialise $\mathcal{C}_q(m) = \emptyset$. These updates should be performed in a single atomic operation.

We assume that each process is able to communicate only with a given subset of system processes $\mathcal{L}_p \subseteq \mathcal{P}$. The complete communication topology can thus be represented by a graph $G(\mathcal{P}, E)$, where processes are the vertices and the communication links between them the edges: there is an edge incident to $\{p_1, p_2\}$ if p_1 can send messages to p_2. We assume that the graph is connected. A set of processes, S, is called a (F^S, B^S) vertex separator, where the sets F^S and B^S are called, respectively, *forward* and *backward* sets, iff $F^S \cap B^S = \emptyset \land F^S \cup B^S \cup S = \mathcal{P}$ and $\forall p_f \in F^S, \forall p_b \in B^S$ every path connecting p_f and p_b passes through at least one vertex of S. In the context of our extended histories, we called such vertex separators *causal separators*.

Causal separators can be exploited to reduce the size of message timestamps as follows. When a member of the causal separator timestamps a message addressed to processes exclusively located in the forward set, it can omit in the timestamp all elements of its causal history that were addressed exclusively to members of the backward set and that were already reported to the other members of the causal separator. More precisely,

R9 (Topological timestamp): Processor p is sending a message m. All messages $N \in \mathcal{H}_p$ for which exists a (F^S, B^S) causal separator, S, such that:

$$p \in S \land \mathcal{A}_m \subseteq F^S \land \mathcal{A}_N \subseteq B^S \land S \subseteq \mathcal{C}_p(N)$$

do not need to be inserted in $\mathcal{H}_m$

Some of the lemmas previously defined, are not affected by this new set of rules. We can re-state these lemmas:

Lemma 7 When rules R1-R5, R6.2, R6.3 and R9 are used, m is-in $\mathcal{D}_p$ only if m has been delivered at p.

Proof: The proof for Rules R1-R4 still applies. $\square$

Lemma 8 When rules R1-R5, R6.2, R6.3 and R9 are used, $m \in \mathcal{H}_{m'}$ only if $m \to m'$.

Proof: The proof for Rules R1-R4 still applies. $\square$

Lemma 9 When rules R1-R5, R6.2, R6.3 and R9 are used, if m is delivered at p, m is-in $\mathcal{D}_p$.

Proof: The proof for Rules R1-R5 still applies. $\square$

Definition 4 *We say that a message m is cited by another message m' through a causal-separator $\mathcal{S}$, and we denote this relation as $m \overset{\mathcal{S}}{\triangleright} m'$, if m is in the causal history of m' or, if for each member of the separator, $x \in \mathcal{S}$, there is in the history of m' a message, m^x addressed to x that contains m in its history. Formally:*

$$m \overset{\mathcal{S}}{\triangleright} m' \overset{\text{def}}{=} \quad (m \in \mathcal{H}_{m'}) \vee \tag{A.1}$$
$$\mathcal{A}_m \subseteq \mathcal{B}^S \wedge \mathcal{A}_{m'} \subseteq \mathcal{F}^S \wedge \forall_{x \in \mathcal{S}} \exists_{m^x} : m \in \mathcal{H}_{m^x} \wedge m^x \in \mathcal{H}_{m'} \wedge x \in \mathcal{A}_{m^x} \tag{A.2}$$

Lemma 10 *If $m \in \mathcal{H}_p$ and $\exists_q : q \in \mathcal{C}_p(m)$ then $\exists_{m^1 \in \mathcal{H}_p} : m \in \mathcal{H}_{m^1} \wedge q \in \mathcal{A}_{m^1}$.*

Proof: m can be inserted in $\mathcal{H}_p$ as a result of rules R6.2 or R6.3. In any case, $\mathcal{C}_p(m)$ is set to $\emptyset$. The only rule that updates $\mathcal{C}_p(m)$ is rule R6.3. In this case, p must have sent m^1 such that $q \in \mathcal{A}_{m^1}$. From rule R2, $m \in \mathcal{H}_{m^1}$ and from rule R4, $m^1 \in \mathcal{H}_p$. $\square$

Lemma 11 *If $m \in \mathcal{H}_p$ and process $p \in \mathcal{S}$ sends a message m' then, $m \overset{\mathcal{S}}{\triangleright} m'$.*

Proof: Assume that $\mathcal{A}_{m'} \nsubseteq \mathcal{F}^S \vee \mathcal{A}_m \nsubseteq \mathcal{B}^S \vee \mathcal{S} \nsubseteq \mathcal{C}_p(m))$. In this case, m is not covered by rule R9 and, according to rule R2, is added to $\mathcal{H}_{m'}$. Thus, from clause A.1, we have $m \overset{\mathcal{S}}{\triangleright} m'$.

If the above condition is false, m is covered by rule R9 and will not be added to $\mathcal{H}_{m'}$. However, from lemma 10, for all x such that $x \in \mathcal{C}_p(m)$, exists m^x in $\mathcal{H}_p$ such that $m \in \mathcal{H}_{m^x} \wedge x \in \mathcal{A}_{m^x}$. Since $x \in \mathcal{S}$, message m^x will not be covered by rule R9 and will be added to $\mathcal{H}_{m'}$. Thus, from clause A.2, we have $m \overset{\mathcal{S}}{\triangleright} m'$. $\square$

Lemma 12 *If process $p \in \mathcal{S}$ delivers a message m' such that $m \overset{\mathcal{S}}{\triangleright} m'$, then $m \in \mathcal{H}_p$.*

Proof: Assume that $m \in \mathcal{H}_{m'}$. From rule R4, all elements of $\mathcal{H}_{m'}$ are added to $\mathcal{H}_p$. Thus $m \in \mathcal{H}_p$. If $m \notin \mathcal{H}_{m'}$ then, by definition, exists m^x in $\mathcal{H}_p$ such that $m \in \mathcal{H}_{m^x} \wedge x \in \mathcal{A}_{m^x}$. By rule R3, m' will only delivered after m^x and from rule R4, all elements of $\mathcal{H}_{m^x}$ are added to $\mathcal{H}_p$. Thus $m \in \mathcal{H}_p$. $\square$

We now prove the following theorem:

Theorem 8 *Rules R1-R5, R6.2, R6.3 and R9 ensure message causal delivery .*

Proof: The liveness proof given for Theorem 2 still applies here. In fact, rule R9 does not affect the proof in any way. This matches the intuitive notion that the elimination of elements from the causal history cannot compromise liveness. However, it can compromise safety:

Consider the actions of a process p that receives two messages m and m' such that $m \to m'$. By rule R3, if $m \in \mathcal{H}_{m'}$ then $\text{DELIV}_p(m) < \text{DELIV}_p(m')$. If $m \to m'$, there is a chain of messages, from the sender of m, s_m, to process p:

$$s_m \overset{m^0}{\hookrightarrow} p_1 \ldots p_i \overset{m^i}{\hookrightarrow} \ldots p_{j-1} \overset{m^{j-1}}{\hookrightarrow} p_j \ldots p_n \overset{m^n}{\hookrightarrow} p$$

In this chain, we consider that all processes are different, that is, we consider only the shortest path (note, that in this proof we do not consider rules to eliminate elements from the causal history of processes).

Thus, we have to prove that if $m \in \mathcal{H}_{s_m}$, then $m \in \mathcal{H}_{m_n}$ for any number of separators crossed in the path. Also note that, in order for rule R9 to be applied to m, due to a causal separator $\mathcal{S}$, process $p \in \mathcal{A}_m$ must be in the backward set of $\mathcal{S}$. Thus if rule R9 is applicable to m, and later in the path, some message is addressed to p, the chain of messages must cross the causal separator *twice*. The proof is by induction on k the number of separators in the path:

Base case ($k = 0$): If there are no separators in the path, then the proof from theorem 1 still applies, thus we will have $m \in \mathcal{H}_{m_n}$.

Inductive step: Assume that the assumption is true if $k = l - 1$ separators are crossed in the chain. We will prove that the relation is preserved when there are $k = l$ separators in the chain. Consider $\mathcal{S}$ the first separator crossed in the chain. Since the separator is crossed twice, we must have two processes $p_i \in \mathcal{S}$ and $p_j \in \mathcal{S}$ in the path. Since $\mathcal{S}$ is the first separator crossed, there are no separators between s_m and p_i, thus $m \in \mathcal{H}_{p_i}$. From lemma 11, either $m \in \mathcal{H}_{m_i}$ or $\exists_{m^j \in \mathcal{H}_{m_i}} : j \in \mathcal{A}_{m^j} \wedge m \in \mathcal{H}_{m_j}$. In any case, either m or m^j will be added to p_{i+1}.

In the first case, since there are $k = l - 1$ separators in the path till p_n, by assumption we will have m in $\mathcal{H}_{m_n}$.

In the second case, for m^j to be "retained" at some other separator, i.e., for rule R9 to be applied to m^j, p_j must be in that separator backwards set. Again, in order for p_j to be reached later in the path, that other separator must be crossed twice. Since there are at most $l - 1$ separators in the path till p_j, by assumption we will have m^j in $\mathcal{H}_{m_{j-1}}$. In this case, m will be added to $\mathcal{H}_{p_j}$. Thus, $m \in \mathcal{H}_{m_j}$, and since there are no more than $l - 1$ separators in the path to p, $m \in \mathcal{H}_{m_n}$. $\square$

Bibliography

[1] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM, Transactions on Computer Systems*, 5(1), February 1987.

[2] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[3] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.

[4] Simon Even. *Graph Algorithms*. Computer Science Press, 1979.

[5] C. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 1988.

[6] Andrew Heybey. The network simulator version 2.1. Technical report, M.I.T., September 1990.

[7] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Lazy Replication: Exploiting the Semantics of Distributed Services. Technical Report MIT/LCS/TR-84, MIT Laboratory for Computer Science, 1990.

[8] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, July 1978.

[9] William S. Lloyd and Phil Kearns. Bounding sequence numbers in distributed systems: a general approach. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 312–319, Paris, France, May 1990. IEEE.

[10] SIgurd Meldal, Sriram Sankar, and James Vera. Exploiting locality in maitaining potential causality. In *Procedings of the 10th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 231–239, 1991.

[11] H Garcia Molina. Elections in Distributed Computer Systems. *IEEE Transactions on Computers*, C-31:48–59, 1982.

[12] Achour Mostefaoui and Michel Raynal. Causal multicast in overlapping groups: Towards a low cist approach. In *Proceedings of the fourth workshop on Future Trends of Distributed Computing Systems*, Lisboa, Portugal, September 1993. IEEE.

[13] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–146, August 1989.

[14] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer Verlag, November 1991.

[15] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information processing letters*, 39(6):343–350, September 1991.

[16] R. van Renesse, Ken Birman, Robert Cooper, Bradford Glade, and Patrick Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, pages 269–283, Seattle, Washington, April 1992.

[17] L. Rodrigues and P. Veríssimo. xAMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 112–121, Houston, Texas, October 1992. IEEE. INESC AR/66-92.

[18] L. Rodrigues and P. Verissimo. Causal separators and topological timestamping: an approach to support causal multicast in large-scale systems. Technical report, IST - INESC, Lisboa, Portugal, 1994.

[19] F. B. Schneider. The state machine approach: a tutorial. In *Proceedings of the Workshop on Fault-tolerant Distributed Computing*, Lecture Notes in Computer Science. Springer-Verlag, 1988.

[20] A. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In search of the Holy Grail. Technical report, Departement of Computer Science, University of Kaiserlautern, P.O. Box 3049, D-6750 Kaiserslautern, Fed. Rep. of Germany, 1991.

[21] M. Singhal and Kshemkalyani A. An Efficient Implementation of vector clocks. Technical report, Ohio State University, Departement of Computer and Information Science, October 1990.

[22] D. Skeen. Determining the last process to fail. *ACM Trans. on Computer Systems*, 3(1), February 1985.

[23] Pat Stephenson. *Fast Causal Multicast*. PhD thesis, Cornell Univ., February 1991.

[24] P. Veríssimo and L. Rodrigues. Order and synchronism properties of reliable broadcast protocols. Technical Report RT/66-89, INESC, Lisboa, Portugal, December 1989.

[25] P. Veríssimo and W. Vogels. The changing face of technology in distributed systems. In *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, pages 119–127, Lisboa, Portugal, September 1993. also as INESC AR/15-94.