The Hierarchical Daisy Architecture for Causal Delivery

Roberto Baldoni[*] Roy Friedman[†] Robbert van Renesse[†]

September 25, 1996

Abstract

In this paper, we propose the *hierarchical daisy architecture*, which provides causal delivery of messages sent to any subset of processes. The architecture provides fault tolerance and maintains the amount of control information within a reasonable size. It divides processes into *logical* groups. Messages inside a logical group are sent directly, while messages that need to cross logical groups' bounderies are forwarded by servers. We proof the correctness of the daisy architecture and discuss possible optimizations.

[*]Dipartimento di Informatica e Sistemistica, Universitá di Roma "La Sapienza", Via Salaria 113, 00198, Roma, Italy. E-mail: `baldoni@dis.uniroma1.it`.

[†]Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA. E-mail: `{roy,rvr}@cs.cornell.edu`.
This work was supported by ARPA/ONR grant N00014-92-J-1866.

1 Introduction

The asynchrony of communication channels is one of the major sources of nondeterminism in distributed systems which in turn is a major cause of problems when implementing distributed applications. *Causal ordering* [3] reduces much of this asynchrony by guaranteeing that whenever a message is delivered to a process, all causally prior messages that were sent to the same process have already been delivered to it. This abstraction yields simplified solutions to many fundamental problems in distributed computing, such as *atomic snapshot* [1], *management of replicated data* [3], and *monitoring of distributed applications* [16].

Several protocols for implementing causal ordering have been proposed [3, 5, 18, 17]. These protocols mainly differ by the assumptions they make on the communication patterns, the topological structure of the underlying network, the amount of control information used to enforce causal ordering, and in how fast the protocol is in sending and delivering messages. For protocols that optimize delivery time, and do not make any assumptions on the topological structure of the network, the following bounds are known:

- If processes only broadcast messages to a group of size n, then the amount of control information that is required to add to messages is $\Theta(n)$[5][1].
- In order to support multicasts to a small number of groups g, each of size n, the amount of required control information is $\Theta(n \cdot g)$ [5].
- In order to allow each process to send each message to an arbitrary set of recipients, taken from a pool of n processes, the amount of required control information is $\Theta(n^2)$ [16].

The bound of $\Theta(n^2)$ for sending messages to arbitrary sets of recipients is prohibitively high for large values of n. Several suggestions have been made to reduce the amount of control information in the common case. For example, for protocols that uses matrix timestamps, e.g., [18], it is possible to send only the difference between the previous matrix timestamp and the current one. It is assumed that in most cases the difference matrix would be very sparse, and therefore can be represented very efficiently. However, in the worst case, a full matrix must be sent, which means $O(n^2)$ integers. Prakash et al. [15] proposed to piggyback on every message explicit information regarding all causally prior messages that are not known to have been delivered to all their destinations, under the assumption that usually there will not be too many such messages. In practice, it is unclear how large this information will be, and even using complex optimizations, it remains $O(n^2)$ in the worst case. Finally, Horus [19] provides the FILTER layer, which translates every send downcall into a broadcast; at the receivers' side, this layer filters out messages that are not intended for the

[1]This control information actually corresponds to a vector of timestamp [13].

local process. This allows to use only a single vector of size n, but is naturally only feasible for relatively small groups equipped with hardware multicast capability.

One of the main problems with most existing protocols for causal ordering is that *a single failure of a process combined with an omission of a single message can prevent the entire system from delivering any additional messages.*[2] Worse yet such failure scenarios are not uncommon in real systems. To overcome this problem, most existing systems adopt conservative techniques that delay transmissions of some messages. This is usually done in a manner unrelated to the control information, which prohibits exploiting these delays in order to reduce the amount of control information.

In this paper we propose a hierarchical architecture that attacks both the problem of reducing the amount of control information and the problem of fault-tolerance at the same time. Our solution splits the participating processes into local groups, and utilizes *causal servers* to disseminate messages across these groups. This adds a delay to messages that need to cross groups' boundaries, as required in any case to guarantee fault-tolerance, but exploits this fact to reduce the amount of control information added to messages. Also, being hierarchical, this solution scales to large numbers of processes, while allowing processes to send each message to any arbitrary set of processes.

Our protocol utilizes a group communication system, e.g., Horus [19], ISIS [3], Transis [7], Totem [14], Phoenix [12], or Relacs [2]. The services of the group communication system we rely on are failure detection and automatic reconfiguration in the event of a failure or a join of a new member, reliable fifo point-to-point delivery, stability detection, and automatic reissuing of messages from failed members that were received by only some members (but not by all of them). This functionality is supported by all the systems we have mentioned. Note that it is possible to implement these functions from scratch. However, since these are well studied problems, assuming a group communication system that provides them simplifies the discussion, and allows us to concentrate on the new things in our architecture and protocol.

In a recent paper, Rodrigues and Verissimo describe an approach which is based on causal separators for reducing the amount of control information in systems that span several network domains [17]. Their approach, however, yields an architecture that is not hierarchical. Also, it does not reduce the amount of control information used within the same subnet domain, even though current LANs can have as many as several hundreds of machines in the same subnet. Our solution does not assume anything about the network topology, although such knowledge can sometimes be used to improve the performance of the system, e.g., by mapping causal servers to routers. Also, Rodrigues and Verissimo do not address the issue of omission failures in [17], so their solution is

[2]Some papers assume reliable delivery of messages and justify this assumption by the use of a point-to-point reliable delivery protocol. As we explain in Section 2.3, unless new messages are not sent until it is guaranteed that older messages have been received, this is not a valid assumption.

less fault-tolerant than ours, as described in Section 2.3.

The rest of this paper is structured as follows: The model and main definitions are introduced in Section 2. The architecture is presented in Section 3 and is shown to be correct in Section 4. We conclude with a discussion in Section 5.

2 Model and Definitions

2.1 Asynchronous Distributed Systems

A distributed system consists of a finite set P of n processes, connected by a *communication network*, or simply *network*, and communicate with each other only by sending and receiving messages through the network. We assume that the network is well connected, but unreliable and asynchronous. That is, the network can delay messages for an arbitrarily long time and may occasionally drop a message completely. However, every message sent from any process to any other process has some positive probability of being delivered after some finite (but unknown) time. In particular, each process can send a message to any set of processes, and this message will be delivered after some finite time to all its recipients with some positive probability, although each recipient may receive the message at a different time. Note that we allow the network to occasionally "drop" a message; we refer to this as an *omission failure*. We assume that processes do not have access to a global clock and there is no bound on their relative speed. In addition to omission failures, processes may fail by crashing [6].

As is done in most papers about causal ordering, we assume that each process consists of an *application level* and a *delivery mechanism*, or *DM* for short, as illustrated in Figure 1. The application level at each process can generate `message-send` events to the DM, and can accept `message-deliver` events from the DM. The DM can generate `net-send` events to the network, and accept `net-receive` events from the network. We also assume that the application level can accept a `crash` event, in which case it is the last event this application level receives.

We say that a message is *sent* when the corresponding `message-send` event is generated; a message is *received* when the corresponding `net-receive` is generated; a message is *delivered* when a `message-deliver` event is generated. We assume that the DM can generate `net-send` events only for messages it accepted `message-send` events for, and can generate `message-deliver` event only for messages it accepted `net-receive` events for. In case the DM generates one `net-send` event for several messages, also known as *piggybacking* or *packing*, we assume, for simplicity reasons, that each of these messages generates a separate `net-receive` event.

We denote a `message-send` event of process p_i sending message m to a subset P_k of processes by $send_i(m, P_k)$ and the `message-deliver` event of a message m at process p_j by $del_j(m)$.

An *execution* of a distributed system P is a collection of `message-send` and `message-deliver`

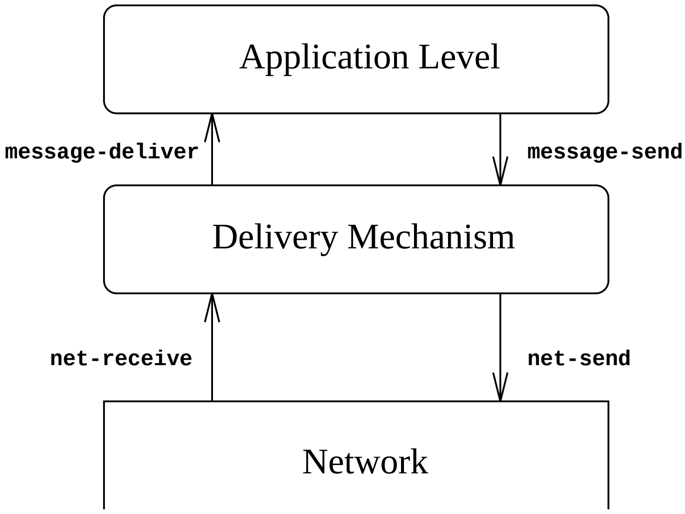

Figure 1: The structure of a process.

events with the following partial order, also known as the *happens before* relation [10], defined on them:

Definition 2.1 (Happens before relation) *An event $a \rightarrow b$ iff*

1. *a and b are two events of the same process and a occurs before b.*
2. *a is a* `message-send` *event $send_i(m, P_k)$, for some p_i, m, and P_k, and b is the corresponding* `message-deliver` *event $del_j(m)$, for some $p_j \in P_k$.*
3. *there exists an event c such that $a \rightarrow c$ and $c \rightarrow b$.*

We say that a process crashes in an execution σ, if the application level of this process accepts a `crash` event during σ. If neither $a \rightarrow b$ nor $b \rightarrow a$, a and b are said to be *concurrent events.*

2.2 Causal Ordering

Introduced by Birman and Joseph in [4], causal ordering states that the order in which messages are delivered to the application must be consistent with the *happened-before* relation of the corresponding sending events. More formally:

Definition 2.2 *An execution of a distributed system σ respects causal ordering if*

1. *for any two messages m_1 and m_2, sent by p_i and p_j, with the same destination p_k such that $send_i(m_1, P_\alpha) \rightarrow send_j(m_2, P_\beta)$ ($p_k \in P_\alpha \cap P_\beta$), $del_k(m_1) \rightarrow del_k(m_2)$.*
2. *for any $send_i(m, P_\alpha)$ event of a process p_i that does not crash in σ and for every process $p_j \in P_\alpha$ that does not crash in σ, the application level of p_j in σ accepts an event $del_j(m)$ from its DM.*

Hence, the problem of implementing causal ordering is the problem of designing a protocol for the DM that will always obey the requirements of Definition 2.2. We say that a message that was received by the DM of a process p_j is *causally deliverable* if all causally prior messages that were sent to p_j have been delivered to the application level of p_j.

The main obstacles facing implementations of causal ordering are the amount of control information required to ensure causal ordering, and overcoming failures of processes and message omissions. The amount of control information used by the protocol is important for the scalability of the solution; if it is too large, then the protocol becomes infeasible for large groups. Also, as we discuss in the following subsection, if the protocol does not explicitly handle message omissions *and* crash failures, then a single failure scenario can block the protocol from delivering messages.

2.3 Fault tolerance

The problem of fault tolerance in causal ordering protocols that allow sending messages to overlapping, but different, sets of recipients has been pointed out by Birman, Schiper, and Stephenson in [5].

Figure 2 illustrates an example of problems caused by a crash of a process combined with an omission failure. Message m_1 is lost by process p_b due to an omission failure. Assuming that the causal ordering protocol releases messages to the network as soon as it receives them from the application, message m_2 is sent from p_a to p_c before p_a notices the omission of m_1. Note that m_2 causally follows m_1. As soon as m_2 is delivered to p_c, it sends message m_3 to p_b. Following this, p_a crashes before noticing that m_1 was lost. Now, when p_b receives m_3, it cannot deliver it, since m_3 causally depends on m_1, which has not arrived yet. On the other hand, since p_a has failed, there is no way to retrieve m_1, and p_b is blocked forever from delivering p_c's messages.

Note that this problem occurs even if processes employ an uncoordinated point-to-point reliable delivery mechanism, such as positive acknowledgements. For example, since m_1 is sent to p_b and m_2 is sent to p_c, such a mechanism would still allow m_2 to be sent and be delivered before the faith of m_1 is determined. The only known solutions to these problems are:

a. Wait for the stability of all previously sent and received messages whenever the subset of recipients changes. This is the solution currently used by ISIS, but it requires delaying such messages for arbitrary long periods of time before sending them.

b. Broadcast every message to all processes, so there is always a way to retrieve m_1. This solution is currently supported by Horus, but it may create too many messages in the network, and is therefore not feasible for large groups.

c. Piggyback on every message all previously unstable messages. This solution was used by early versions of ISIS. However, it may generate extremely large messages.

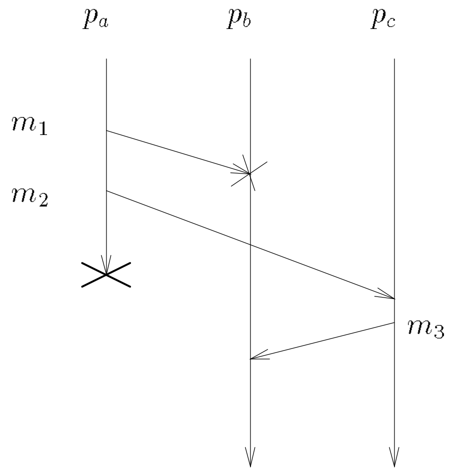

Figure 2: A failure scenario which causes causal protocols to block.

3 The Proposed Architecture

In this section, we describe a hierarchical architecture for implementing causal ordering by a collection of DMs. We start by describing the simple case, in which there are only two levels in the hierarchy, and then discuss how this can be generalized to more levels. Finally, we discuss several optimizations to the general architecture, which can optimize the performance of the system.

3.1 The Base Case

We divide the processes into *local groups*, such that each process is a member of only one local group, as illustrated in Figure 3. Groups are managed by a group communication system, e.g., Horus. In each group, a single process is chosen (deterministically) to be the *causal server* for that group. All causal servers are also members of another group simply called the causal servers' group.

We refer to this as the *daisy configuration.* We assume that the group communication system can detect process failures, and report them to members of the group, and allow new members to join the group. We assume also that inside a group, the system provides a point-to-point fifo, reliable, but uncoordinated, message delivery, e.g., by employing a positive acknowledgment protocol or a negative acknowledgment protocol with periodic updates. Furthermore, we assume that messages delivered within a group are buffered by the group communication system until they are known to have been received everywhere, and that in the event of a failure, the system automatically relays omitted messages that were originated by failed members to members that have not received these messages. All these are standard features in all group communication systems we have mentioned.

In order to send a message, a process does a causal broadcast of this message to all members of its local group, using the conventional causal broadcast protocol, which requires only a vector of

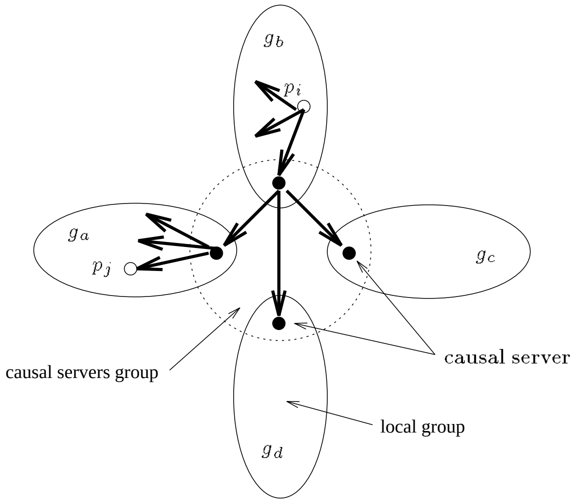

Figure 3: An example of a daisy configuration with a message sent from process p_i to p_j.

integers of the size of the local group [18]. Each member that receives a message that is intended for it, delivers the message locally as soon as the message becomes causally deliverable. If the message is also intended for processes that are not members of the local group, the causal server waits until the message becomes causally deliverable in the local group, and then does a causal broadcast of this message in the causal servers group. Each causal server that receives such a message, waits until it becomes causally deliverable in the causal servers group, and then if some of the intended recipients of this message belong to its local group, does a causal broadcast of this message inside its local group. In this case, again, as soon as the message becomes causally deliverable, each process that was supposed to receive this message delivers it, and all other processes discard it. [3]

An important point to note about this architecture lies in the fact that it dynamically changes according to the number of processes running. For example, to get the daisy architecture depicted in Figure 3, there are some intermediate steps to do which are depicted in Figure 4. At the beginning there is just one causal server, i.e., a singleton group (Figure 4.a). At this point, new processes join the group g_a (Figure 4.b). When the control information becomes to large, or too many messages are received by everyone, a new causal server is created and the processes split into two local groups g_a and g_b (Figure 4.c).[4] If the number of processes in the system continues to grow, we get the configuration of Figure 3. So, it turns out that in this architecture the number of server processes is usually much smaller than the number of processes in a group. If the number of processes decreases, then two groups can be merged back into a single group.

[3]Note that the DM buffers the messages in case it needs to be retransmitted after a chrash.

[4]New group memberships can be determined according to the amount of information exchanged among processes: if two processes communicate frequently they will become members of the same group.

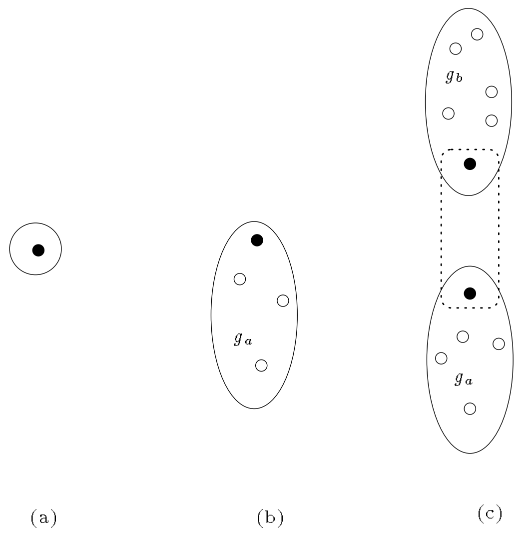

Figure 4: Evolving daisy configuration.

3.1.1 Stability Detection

Recall that for fault-tolerance purposes, processes buffer messages they send and receive. However, there is no point to continue buffering a message after it was delivered to all of its intended recipients. We say that a message is *stable* when all live recipients have received it. Thus, in order to keep the amount of local storage from overflowing, the causal delivery service must employ a stability detection mechanism.

A stability detection mechanism is usually based on the fact that processes inform each other of messages they receive, either by piggybacking this information on messages, or occasionally generating specific messages for this purpose. For efficiency purposes, many protocols employ "gossiping" in the implementation of the stability detection protocol, which shortens the time required for a process to learn that a message is stable, and reduces the message overhead.

All group communication systems we have mentioned in Section 1 provide a stability detection mechanism within a local group. Therefore, we assume the existence of such a mechanism. Note, however, that our hierarchical mechanism requires stability information to traverse across groups, and that the sender of a message should not discard it until it is stable w.r.t. every recipient, and not just within the group. The stability detection mechanisms of some group communication systems (e.g., Horus) require the application at a process to explicitly inform the stability mechanism when it is ok to report that it has received the message in order to achieve end-to-end semantics. We therefore assume this capability as well, and describe how cross-group stability detection is implemented using the intra-group stability mechanism provided by the group communication system.

A causal server does not report a message as stable in the servers group, until it becomes stable in its local group. (If there are no recipients in the local group, the causal server can immediately report the message as stable.) A causal server that belongs to a local group where a message was initiated, does not report the message as stable to its local group until it is stable in the causal servers group. This way, once a message is declared stable in the group that included its original sender, it is guaranteed to have arrived at all its destinations, and it is therefore safe to discard it.

3.1.2 Handling Failures

Note that if a non-server member of the group fails, then the group communication system takes care of resending all messages that originated by that member and were received by only part of the members of the same local group (but not by all of them). This, combined with the fact that every message is broadcast to all members in the local group, and the point-to-point reliable delivery mechanism of the group communication system, guarantee that such a failure will not block the protocol from making progress, as in the examples discussed in Section 2.3. Thus, the only thing that need be addressed explicitly by our protocol is a failure of a causal server.

A failure of a causal server is problematic since it is a member of two groups, and messages in one group do not propagate automatically to the other by the group communication system. In particular, when a causal server fails, the local group it belongs to is reconfigured and another member is elected to be the causal server for that group. However, this server must be updated regarding the messages that were delivered in the servers group, but have not made it to the local group when the previous causal server crashed.

Recall that a causal server does not report a message as stable until it becomes stable in its local group. Thus, whenever a causal server fails, all messages that were still not reported by it as stable in its local group are recorded. When the new causal server joins the servers group, a state update message is sent to it, which includes copies of all such messages. The new causal server can then forward these messages in its local group. A simple duplicate suppression mechanism is used to prevent duplicate messages from being delivered in the local group. Such a mechanism consists of adding a unique identifier to each message, which includes the originator's unique identifier, and the sequential number of this message at this process. Since our protocol guarantee causal ordering, and since causal ordering also implies fifo ordering, each process needs to maintain a single vector with one entry for each process in the system. However, since this vector is kept in memory, and is not added to messages, is does not impose a scalability problem.

3.2 The general case

If we have to cope with a huge number of processes, the basic architecture becomes inadequate, since the size of the control information and the number of messages generated becomes very large. In this case, we can add another daisy configuration and divide processes between the two daisies

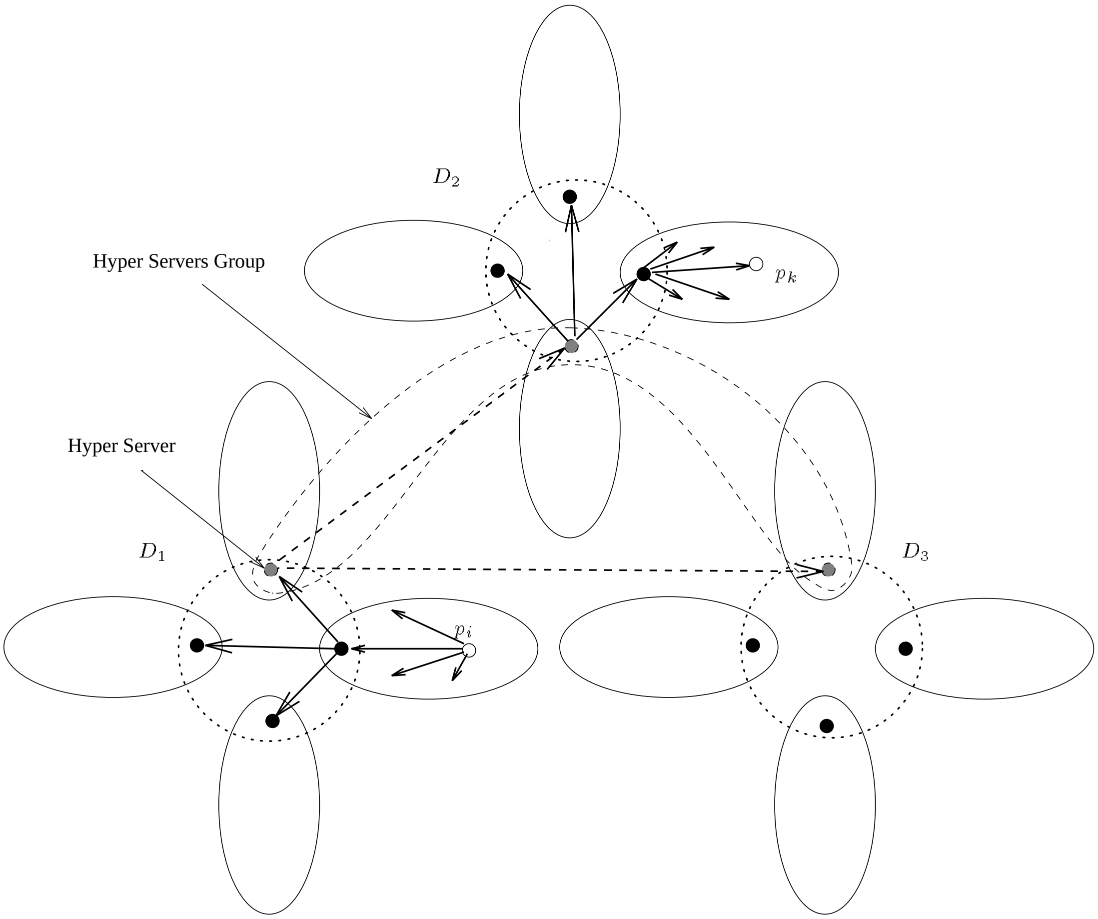

Figure 5: A system with three daisies in which a message is sent from process p_i to p_k.

according, for example, to their communication activity[5]. Processes that communicate rarely can be put on distinct daisies. At this point either one of the causal servers of one of the daisies or a new "ad-hoc" causal server becomes responsible for forwarding messages from one daisy to another. We call this server a *hyper-server*. All hyper-servers form a *hyper-servers' group*. Figure 5 shows a system with three daisies. Each one has selected one causal server for interdaisies communication.

In this general architecture, we assume that interdaisies communication among causal servers are causally ordered. When a causal server receives a message m from p_i to be forwarded to a set of destinations in other daisies, it broadcasts m in the causal servers' group of that daisy. The hyper-server of this daisy then broadcasts m in the hyper-servers' group. Hyper-servers whose daisies include recipients of m broadcast m in the server's group of their daisy, so that causal servers in those daisies whose local groups include recipients of m can broadcast m there.

This can be done several times, to achieve several levels of hierarchy. Of course, in practice, we assume that 2 to 4 levels would be enough for most applications, since the number of levels grows logarithmically with the total number of processes.

[5]Processes can also be divided according to the physical topology of the communication network. Processes on the same LAN segment could be members of the same group.

3.3 Performance

The number of messages required to deliver a message depends on the number of different groups in which there are recipients; we need one broadcast in each group where there are recipients, one broadcast in the group that the originator of the message belongs to, and one broadcast for each servers group the message has to go through. Using packing techniques, as described in [8], it is possible to greatly reduce the number of actual (physical) messages being sent.

Since within a group we are using the causal broadcast protocol presented in [18], a message always carries a single vector of length m, where m is the size of the (local or servers) group the message is being sent in.

3.4 Possible Optimizations

We now outline several optimization that can reduce either the amount of control information, the number of messages, or the delays. However, each of these optimizations involves a tradeoff. Decreasing the amount of control information and the number of messages yield longer delays for messages, and a lower degree of fault-tolerance. On the other hand, reducing message delays requires more control information and slightly more complex rules for delivering messages.

1. A causal server that receives a message from its local group can forward the message to the servers group as soon as it receives the message, and not wait for it to become causally deliverable in the local group. Similarly, a causal server that receives a message in the servers group can forward it immediately inside its local group. However, this requires the message to carry one vector for each group it traverses in, and requires complex rules for deciding when a message becomes causally deliverable at its recipients.

2. It is possible to send messages inside a group only to their intended recipients, instead of broadcasting it to the entire group. This requires the use of matrices of size m^2 for groups of size m, instead of just a vector of length m. Also, in order to have some degree of fault tolerance, messages cannot be delivered until it is known that they were received by at least k more processes, in order to be able to survive k failures.

3. It is possible to send all messages inside a local group point-to-point to the causal server, and have the causal server forward the messages to all recipients within the group on a point-to-point basis. (Recall that we assumed that the group communication system employs a reliable fifo mechanism, such as negative acknowledgements, for messages sent within the group.) Forwarding message in the servers group occurs in this case using the regular causal broadcast algorithm.

This approach reduces the number of messages required to send a message, and the amount of control information carried on messages, since delivering point-to-point messages reliably and in fifo order requires only a single integer to be added to each message. The disadvantage of this approach is that messages sent within the group need to pass through an extra hop. Also, this creates an additional load on the causal server, since it has to forward all messages sent in the local group to their recipients inside this group.

Note that all these ideas can also be implemented at higher levels of the hierarchy. The same trade-offs apply at these levels as well.

4 Correctness

We show that this architecture guarantees causal delivery (*safety property*) and that each message will be eventually delivered (*liveness property*) in a system composed of a two-level daisy. The proof for higher levels is a trivial extension of the proof we give for two levels, and is therefore omitted.

4.1 Safety

Note that omission failures and crash failures cannot alter the safety of our architecture. They can prevent the protocol from delivering messages, but this is a liveness issue. Liveness is shown in the next subsection, so in the rest of this subsection we can ignore failures.

Theorem 4.1 *The protocol presented in Section 3 deliveres messages in causal order.*

Proof Assume, by way of contradiction, that there exists an execution σ of the protocol in which some messages are delivered in an order that violates causal ordering. Let m and m' be two such messages. Denote by p_i the process that sends m, p_j the process that sends m', g_a the local group that p_i is a member of, and g_b the local group that p_j is a member of. Assume, w.l.o.g., that $send_i(m, P_k) \rightarrow send_j(m', P_{k'})$, but $del_q(m') \rightarrow del_q(m)$ for some $p_q \in P_k \cap P_{k'}$.

Clearly, this cannot happen if $g_a = g_b$. Thus, we must assume that $g_a \neq g_b$. Since $send_i(m, P_k) \rightarrow send_j(m', P_{k'})$, m is causally deliverable at the causal server of g_b, denoted by p_b, by the time it forwards m' to the causal servers group. Thus, in the causal servers group, m' gets a vector timestamp that indicates that it causally follows m.

Let g_c be the group of p_q, and denote by p_c the causal server of p_q. Thus, p_q sends m in g_c before it sends m' in g_c, meaning that the vector timestamp of m in g_c indicates that it has to be delivered after m'. Therefore, p_q must deliver m before it delivers m', a contradiction to the assumption that $del_q(m') \rightarrow del_q(m)$. □

4.2 Liveness

Before proving liveness, let us introduce the notion of *persistent causal hole* (PCH). By the definition of causal order, a message cannot be delivered to a destination p_i until all causally prior messages also sent to p_i have been delivered there. Hence, if a message m sent to a process p_i is lost and cannot be retrieved somewhere in the distributed system, all messages m' sent to p_i such that $m \rightarrow m'$ cannot be delivered to p_i. Due to the acyclicity of the happened-before relation, we assume, w.l.o.g., that all messages that are causally prior to m and sent to p_i have been delivered by p_i. In this case, we say that m creates a PCH in p_i. Let us introduce the following lemmas:

Lemma 4.2 *A group communication system (e.g., HORUS) and our protocol ensure that in a single group of processes there cannot be a PCH.*

Proof Assume, by way of contradiction, that a message m, sent by p_j to p_i, creates a PCH in some execution of the protocol. According to the protocol of Section 3.1, each message is actually sent to each member of the local group, and every such member buffers m until it becomes stable (Section 3.1.1). By definition of stability, if m creates a PCH, it cannot be stable.

In the absence of crash failures, the liveness of the causal broadcasting protocol adopted [18] and the reliable fifo point-to-point delivery mechanism of the group communication system guarantees that m will eventually be delivered to p_i. If some process crashes, the group communication system automatically relays all unstable messages among the non-faulty processes. Therefore, m will be eventually received by p_i. A contradiction to the assumption that m creates a PCH. $\square$

Lemma 4.3 *A group communication system (e.g., HORUS) and our protocol ensure that there cannot be a PCH in a group of processes g_a and in the associated causal server group g_c due to a message sent by g_a (resp. g_c) and to be relayed to g_c (resp. g_a).*

Proof Assume, by way of contradiction, that there is a PCH in one of two groups due to a message that should have been relayed from g_a (resp. g_c) to g_c (resp. g_a), and denote by m the message that caused this PCH. In the absence of a crash failure of the causal server of g_a, the liveness of the causal broadcasting protocol adopted [18], Lemma 4.2, the reliable fifo point-to-point delivery mechanism of the group communication system, and the relay mechanism done by the causal server of g_a (Section 3.1) guarantees that any message m broadcast in the group g_a (resp. g_c) and with some recipient out of g_a (resp. in g_a) will be eventually received by any process of g_c (resp. g_a).

Therefore, we must assume that p_a, the causal server of g_a crashes, m is destined to some recipients outside of g_a (resp. inside of g_a) and was not yet relayed by p_a to g_c (resp. g_a). Hence, the intergroup stability mechanism of Section 3.1.1 considers m to be unstable. According to our protocol (Section 3.1.2), another member of g_a, p'_a is elected (in a deterministic way) to be g_a's new causal server. Due to Lemma 4.2, $p_{a'}$ will eventually receive all unstable messages of g_a. Similarly, as soon as p'_a is elected, it will join g_c's group and will receive all unstable messages in g_c, and will relay unstable messages from g_a to g_c and vice versa.

Thus, m will eventually be received by processes in group g_a (resp. g_c), a contradiction to the assumption that m causes a PCH. $\square$

Theorem 4.4 *Every message that is sent by a process that does not crash is delivered to all its recipients that do not crash.*

Proof Since we have shown that the causality relation among messages is guaranteed by our architecture (Theorem 4.1) and we use a causal ordering protocol in each group that has been proven to be live in [18], we will prove liveness of our protocol showing that there cannot be a PCH in any execution of our protocol.

Hence, assume, by way of contradiction, that there exists a message sent by a member of group g_a that creates a PCH in a member of group g_b, and let g_c be the causal server group. Two cases are possible:

- $g_a = g_b$. Lemma 4.2 outrules a PCH in this case.
- $g_a \neq g_b$ (it is possible that $g_b = g_c$). By Lemma 4.2 and by applying Lemma 4.3 repeatedly, first, to the pair g_a and g_c, and then to the pair of groups g_c and g_b, there cannot be a PCH in this case either.

Therefore, in both cases there cannot be a PCH, a contradiction to the assumption that m causes a PCH. $\square$

5 Discussion

In this paper we have described the hierarchical daisy architecture for fault-tolerant causal ordering. This architecture can tolerate both omission and crash failures, while keeping the amount of control information small. In our solution, messages may need to go through one or more intermediate nodes on their way to their destination, but similar delays are needed in any case in order to guarantee fault-tolerance.

A smart utilization of our architecture can make it well suited to large area networks, by mapping processes in the same (or close) domains to the same daisies, and members of distinct domains to separate daisies. This is just a performance optimization, and is not required for the correctness of the proposed implementation.

Since our scheme can tolerate both omission failures and crash failures, it may be possible to combine a simple modification of it (the third optimization described in Section 3.4) with tools like GTS [11], to serve as a causal delivery mechanism for mobile computing, having the causal servers serve as the *mobile support station* and processes of a group as mobile hosts in a cell [9]. This may require some more work on the definition of a *hand-off* protocol when a mobile computer transfers from one cell to another, in order to guarantee exactly one delivery of each message, and is left for future research.

References

[1] S. Alagar and S. Venkatesan. An optimal algorithm for distributed snapshots with causal message ordering. *Information Processing Letters*, 50:311–316, 1994.

[2] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. Technical Report UBLCS–94–15, Department of Computer Science, University of Bologna, June 1994. Revised January 1995.

[3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.

[4] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[6] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Diffusion to Byzantine Agreement. In *Proc. of the 15th International Conference on Fault-Tolerant Computing*, Austin, Texas, 1985.

[7] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[8] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95–1491, Department of Computer Science, Cornell University, March 1995.

[9] T. Imielinski and B. R. Badrinath. Mobile Wireless Computing. *Communication of the ACM*, 37(10):19–28, 1994.

[10] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[11] S. Maffeis, W. Bischofberger, and K. Matzel. A Generic Multicast Transport Service to Support Disconnected Operation. In *Proceeding of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, pages 79–89, Ann Arbor, MI, April 1995.

[12] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A Tollkit for Building Fault-Tolerant Distributed Application in Large Scale. Technical report, Department d'Informatique, Ecole Polytechnique Federale de Lausanne, July 1995.

[13] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[14] L. Moser, P. M. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[15] R. Prakash, M. Raynal, and M. Singhal. An Efficient Causal Ordering Algorithm for Mobile Computing Environment. Technical Report 2680, INRIA, October 1995.

[16] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39:343–350, 1991.

[17] L. Rodrigues and P. Verissimo. Causal Separators and Topological Timestamping: an Approach to Support Causal Multicast in Large-Scale Systems. In *Proc. of the 15th International Conference on Distributed Systems*, May 1995.

[18] R. Schwarz and F. Mattern. Detecting Causal Relations in Distributed Computing: in Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.

[19] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.