A Sequencing Service for Group Communication

Tim Kindberg
Dept. of Computer Science
Queen Mary & Westfield College, University of London
Mile End Rd., London E1 4NS
email:timk@dcs.qmw.ac.uk
WWW: http://www.dcs.qmw.ac.uk/~timk/

Key words: group communication, multicast, total order, causal order, reconfigurable service.

1 Introduction

This paper is about the design of a sequencing service for totally- and causally-ordered group communication in a distributed system. A group communication service provides facilities for managing groups of processes and for multicasting messages to all the members of a group. The primary role of the sequencing service is to impose a total order on multicast messages, but we shall show how to build a combined causal order from it, and also a new hybrid order which we call cautal ordering. This design addresses the issue of sequencing many groups, which may overlap, and of using several sequencer processes to do so.

In Section 2 we briefly explain the requirement for group communication involving overlapping groups, in the context of multi-user applications; the section also introduces the concept of a multiple-peer server, which we use as a framework for the sequencing service. Section 3 outlines sequencer-based protocols for totally-ordered multicast to a single group. Section 4 describes extensions to these protocols to handle multiple groups, and to split the sequencing load between several sequencers whenever possible. Section 5 shows how to obtain causal and cautal ordering from total ordering. Section 6 compares our protocols to related work, and Section 7 summarises and concludes the paper.

2 Background

The context of the design is, firstly, an investigation into multicast communication requirements for groupware, including shared editors such as Grove [Ellis et al 1991], and applications presenting a single 'virtual reality' shared between several users (e.g. MASSIVE [Benford et al 1994], DIVE [Carlsson and Hagsand 1993]). In these applications, users share a common environment (for example, a document or 3D scene), through which they interact and observe one another's effects. In principle, the entire collaborative activity of an organisation such as a Computer Science department can be represented using a single, extended metaphorical environment through which members navigate and interact. In practice users need observe only part of a large environment at a given time. A user's view may or may not overlap with other users' views. Users may change parts of the environment, and others observing them must see these changes. The major requirements are for timely communication and consistent views.

Objects need to be replicated at the participating computers if satisfactory performance is to be achieved. The set of processes containing replicas of a shared object forms a group, a *replica group*, to which updates must be propagated. One approach would be to replicate all objects at all users' computers. However, this is wasteful in the case of objects that are not observed at some workstations. If a user is editing a paragraph in a document, and if this is off-screen at a particular workstation, then there is no need to propagate the characters to that station as they are typed; it is preferable to fetch an up-to-date copy of the paragraph when it becomes visible. A computer need store an up-to-date version of an object only if it is observable there, or if it is a sub-object (a 'helper', such as a renderer) of an observable object.

Operation	Function
createGroup() → groupId	Create group, learn its identifier, but don't join it.
joinGroup(groupId)	Caller joins named group
leaveGroup(groupid)	Caller leaves named group
multicast(groupId, message)	Deliver message to members of group

Table 1. Group communication operations.

Objects in a shared environment may react to events generated by user actions or by other objects. For example, in a virtual environment objects change their appearance when a user switches on a light; objects may collide. The set of processes containing objects that must be notified when an event of a certain type occurs naturally forms a group, of a type called a *subscription group*.

Note that for both replica groups and subscription groups, the membership may vary over time, sometimes rapidly. A strong requirement is that group membership changes should not incur latencies that are noticable to the user.

The second background component to this paper is previous work on multiple peer servers [Kindberg 1992]. These are sets of server processes that share client workload between them. Unlike servers in multicast groups, each peer satisfies requests from a subset of the clients by itself. As the client population varies, so may the number of server peers used to service them. Also, clients or groups of clients may be transferred between server peers as client requirements or load changes. These reconfigurations of the service are transparent to the clients. The relevance to this work is that, while it is always possible to use a single sequencer to sequence messages sent to all process groups, it is desirable on performance grounds to divide sequencing between several peer sequencers. However, the requirement for total message ordering constrains the division of groups between sequencers, as the next section shows, and it is sometimes necessary to transfer clients between them.

3 Sequencer-based multicast protocols

3.1 Group communication semantics

Let a *group* be a set of processes – the group's *members* – to which messages are delivered. We consider a group communication service that provides the operations given in Table 1.We now define the conventional multicast semantics of FIFO, total and causal ordering of message delivery, and then propose a hybrid, called cautal ordering. For a message m, let $dest(m)$ be the group to which m is addressed, and let $sender(m)$ be the process that sent m. We distinguish between the event of a message being *received* at a destination site, when it may or may not be ready for consumption by the local process, and the message being *delivered* to the process, when the process consumes the message. We also define $send_p(m)$ to be the event of a process p sending the message m, and $deliver_p(m)$ the event of m being delivered to process p. Finally, '$\rightarrow$' denotes Lamport's happened-before relation [Lamport 1978].

FIFO ordering. A multicast is said to be *FIFO-ordered* if $deliver_q(m) \rightarrow deliver_q(m')$ whenever $send_p(m) \rightarrow send_p(m')$. That is, a message sent by process p is not delivered to q unless all previous messages sent by p have also been delivered to q.

Total ordering. Under total ordering, if G and G' are groups (not necessarily distinct) and $G \cap G' \neq \emptyset$, then if $dest(m) = G$ and $dest(m') = G'$, then m and m' are delivered to all processes in $G \cap G'$ in the same order, even when $sender(m) \neq sender(m')$. Totally ordering

multicast messages enables group members to maintain replicated data, since they apply the same series of operations to their state.

Causal ordering. Let $send_q(m) \rightarrow send_r(m')$, with messages m and m' sent to G and G' as before. A multicast is causally ordered if $\forall p \in G \cap G'$: $deliver_p(m) \rightarrow deliver_p(m')$. That is, m' is delivered to process p only when all causally prior messages have been delivered to p. Causally ordering multicast messages enables group members to maintain data that meet application requirements for compatibility (between the members' data), but which are not necessarily identical.

Total ordering is too strong in some applications that contain overlapping groups. For example, consider an application containing replicated objects. Operations upon individual replicated objects are totally ordered, to keep the replicas consistent. However, it is not always necessary for operations on two different replicated objects to be ordered consistently at all sites; often, a causal ordering will suffice. Consider a collection of brokers who receive financial information concerning various stocks and commodities. The set of brokers receiving information about a particular stock or commodity forms a group. Changes to the price of any one item should be totally and causally ordered. Moreover, if a broker should sell stocks in the Computer Manufacturing Company because he has seen the price of memory chips triple, thus reducing the value of the former, then brokers observing both price movements should see the changes in causal order. In general, brokers need to see changes to information about different stocks or commodities in causal but not total order.

We therefore propose a hybrid ordering semantics, *cautal ordering*, which guarantees causal ordering across all overlapping groups, but only totally orders messages sent to each individual group.

Cautal ordering. A multicast obeys a cautal order if it is causally ordered and if two messages m and m' sent to the same group G are delivered to all processes in G in the same order.

We take up the question of implementing cautal and causal multicast in Section 5. In the meantime, we consider only total ordering.

3.2 Sequencer-based protocols

Sequencers are employed in the Amoeba group communication protocol [Kaashoek & Tanenbaum 1991] and the ISIS *ABCAST* protocol [Birman et al 1991]. These protocols impose a total order on messages by assigning a unique sequence number from a continuously increasing series to each message sent to a given group. Messages are delivered to group members in order, according to the sequence number they bear. Each multicast message is sent to the sequencer; the sequencer's role is to assign sequence numbers in the order in which it receives the messages.

In ISIS, when the *ABCAST* multicast primitive is invoked the protocol appends a unique identifier to the message and sends it using the causally-ordered *CBCAST* protocol to the group members; however, the message is not yet delivered. A designated member site acts as the sequencer: it collects a list of one or more pairs *<messageIdentifier, messageSequencerNumber>* of the messages it has received, and reliably multicasts this 'sets-order' message to the group. On receipt of a sets-order message, the communication module at a group member delivers received messages in order of their sequence number in the list.

The Amoeba protocol does not employ an underlying causally-ordered protocol, but directly uses the broadcast and point-to-point packet transmission facilities of a local area network such as the Ethernet. In the most basic variation on this protocol, a message to be multicast is first sent point-to-point to a sequencer. The sequencer increments the sequence number, attaches it to the message and uses hardware broadcast (or multicast) to send the message to all the members of the group. The multicast message is sent to the sender, thus serving as an acknowledgement. The receiving communication module marks an arriving

multicast message as deliverable if it bears the next expected sequence number; otherwise, if the arriving sequence number is larger than expected, it sends a request to the sequencer to obtain the missing message(s), which it delivers before the out-of-sequence message. The sequencer keeps messages that are not yet known to have been received by all members of the group, in a so-called *history buffer*. This negative acknowledgement scheme is efficient on a network on which packets are rarely dropped (a LAN or a local internetwork). Kaashoek & Tanenbaum [1991] describe a more reliable variant on this protocol.

Note that an Amoeba sequencer totally orders changes in the group's membership with respect to message delivery, thus giving rise to virtually synchronous executions [Birman & Joseph 1987]. Also, group membership changes are relatively inexpensive in this protocol: for a process to join or leave a group requires only a single RPC to a sequencer.

3.3 Limitations of existing protocols

Over a local area network with a multicast capability, a sequencer-based protocol produces relatively low latencies. However, there is a tendency for a single sequencer process to become a bottleneck as the usage of group communication increases. If groups are disjoint, then separate sequencers may be used with different groups. But to facilitate large-scale sharing of common environments, a protocol must address the following points:

- many groups may co-exist
- groups may overlap – that is, have members in common
- many messages may be sent – in applications that require good response times to be effective.

Two groups that are initially disjoint may later acquire common members as the pattern of sharing changes, and therefore group membership changes. It is equally true that overlapping groups may become disjoint.

Both the Amoeba and ISIS group communication systems deal only with *closed groups* (or *peer groups*), in which the sender of a message is also a member of the destination group. However, the example of event notification given in Section 2 shows that this is not always a convenient assumption. The notifier of an event does not normally expect to receive its own notification (although it may choose to do so, if other processes notify it of their own events and it needs to know in what position in the total order its event came).

4 The reconfigurable sequencing service

We now describe a sequencing service implemented by several sequencer processes executing at distinct nodes of a distributed system. Each sequencer orders messages sent to one or more groups. We shall describe protocols that give total and causal, and cautal orderings. The first protocol entails reconfigurations which involve the transfer of the sequencing activity for a particular group from one sequencer to another.

A sequencer holds the following data and resources for the purposes of multicast to a particular group:

- The group's identifier, G
- The last sequence number used for the group
- A membership list $<(member_1, address_1), .., (member_n, address_n)>$
  (Only one address entry is necessary if the sequencer uses a low-level, unordered multicasting service such as IP multicast [Deering and Cheriton 1990], which employs a single address per group.)
- A history buffer
- A communication port for receiving requests.

Request	Contents	Response	Contents
CREATE-GROUP		GROUP-CREATED	groupId, portId
JOIN-GROUP	groupId, processId, groupList	GROUP-JOINED	groupId
LEAVE-GROUP	groupId, processId, groupList	GROUP-LEFT	groupId
DO-MULTICAST	groupId, msgId, context, msg	ACCEPT	groupId, msgId, context, msg
FETCH-MSGS	groupId, fromSeq, toSeq	msg1<, msg2,..>	requested messages

Table 2. Sequencing service messages.

Clients communicate with the sequencing service using port identifiers. Each such identifier refers to some port owned by one of the sequencers; however the client is unaware of which sequencer owns the port and therefore processes its requests. Furthermore, ports may migrate between sequencers (as is possible, for example, in Equus [Kindberg et al 1994], Mach and Chorus [Coulouris et al 1994]), and port migration is transparent to the clients. The service maintains a port for each group, and it also maintains a port for requests to create new groups. We assume that a name service provides clients with port identifiers for well-known groups, and for contacting the service to create new groups. The client library maintains a table mapping group identifiers to port identifiers, and the application programmer is not concerned with them when using the operations given in Table 1 above.

The set of request messages that a sequencer recognises and the messages that it emits in response are given in Table 2. A client wishing to send a message to a group transmits a *DO-MULTICAST* message to the corresponding port. The sequencer owning that port examines the message's identifier and, if the message is fresh, responds by multicasting an *ACCEPT* message to the group members; if the sender is not a group member, then the sequencer additionally sends a separate, point-to-point *ACCEPT* message to it, which acts as an acknowledgement. The group members queue the message body for delivery to the application, assuming that the sequence number is the one they expect. Clients use the last of the request types, *FETCH-MSGS*, to fetch missed messages.

The parameter *context* in *DO-MULTICAST* and *ACCEPT* messages is discussed below, as is the *groupList* parameter for *JOIN-GROUP* and *LEAVE-GROUP* messages. Clients append a unique identifier *msgId* to their *DO-MULTICAST* messages. It is convenient for them to construct this as a combination of their unique process identifier and an integer representing the number of distinct *DO-MULTICAST* messages it has addressed so far for the group in question. The history buffer contains pairs of identifiers and sequence numbers of *ACCEPT*-ed messages, for detecting and acknowledging (in a point-to-point *ACCEPT* message) repeated *DO-MULTICAST* requests. As explained above, it is also used for saving messages not yet known to be *group-stable*, where group-stable messages are ones that have been received at all members of the destination group (defined according to membership at the time when the sequencer first received the message).

Note that each message body is transmitted twice in this protocol, once in a *DO-MULTICAST* message and once in an *ACCEPT* message. Kaashoek and Tanenbaum describe an alternative protocol, in which clients multicast the message body to the group members (including the sequencer), and the sequencer's *ACCEPT* message does not contain the message body, but simply assigns the sequence number of the corresponding message. The reduction in bandwidth utilisation is traded off against the need for group members to receive two messages before the message body can be delivered to the application, and against the need for the sequencer to adjust the group membership lists held by multicasting processes. It is possible to adapt this protocol to our multiple-sequencer service, but for the sake of clarity we omit this here.

4.1 Sequencing multiple groups with one sequencer

Multicasts sent to disjoint groups can straightforwardly be sequenced by one or more sequencers. At one extreme, one sequencer could sequence all the groups; at the other extreme, we could assign a sequencer for each group, to gain better performance. Where a sequencer sequences multicasts for several groups, it must keep a separate sequence number for each group.

Sequencers cannot so straightforwardly order multicasts addressed to intersecting groups. Even a single sequencer sending to two intersecting groups must take special measures to sequence the messages sent to them. Suppose a single sequencer is used, and that it appends pairs <*groupId*, *sequenceNumber*> to its messages. How can a member of several groups know whether it hasn't missed some messages destined for other groups of which it is a member, between the last message it received and a newly arrived one? For example, if it receives <group 1, 46>, and then <group 1, 47>, it may be that a message bearing <group 2, 157> was dropped, but should be delivered between the two messages sent to group 1.

The solution is for the sequencer to append a generalised sequence label called a *context* to *ACCEPT* messages, constructed from the sequence numbers pertaining to individual groups, which allows receivers to determine the total order based upon the groups of which they are a member.

A context C is a set of pairs of group identifiers and sequence numbers (elements of *Nat*, the natural numbers), with just one pair for any group: if $\langle G, n\rangle \in C$ and $\langle G, n'\rangle \in C$ then $n = n'$. We define a component-taking operation, a restriction operation, a merge operation and inequality between contexts as follows:

$C.G = n$, if $\langle G, n\rangle \in C$

$= \perp$, otherwise, where $\forall n \in Nat: \perp < n$.

$C|S = \{\langle G, n\rangle: \langle G, n\rangle \in C \wedge G \in S\}$.

$\text{merge}(C_1, C_2) = \{\langle G, n\rangle: (C_1.G \neq \perp \vee C_2.G \neq \perp) \wedge n = \max\{C_1.G, C_2.G\}\}$.

$C_1 \leq C_2$ *iff* $\forall G: C_1.G \leq C_2.G$.

$C_1 < C_2$ *iff* $C_1 \leq C_2 \wedge \exists G: C_1.G < C_2.G$.

The component-taking operation returns the sequence number for a given group in the context, if such exists. The restriction operations selects just those items in a context that pertain to a given set of groups. The merge operation selects the entries with the largest sequence numbers from two contexts.

Let *sequencer*(G) be the unique sequencer that sequences a group G at a particular time, let *seq#*(G) be G's latest sequence number. The sequencer keeps a context C for all the groups it sequences, such that $C.G$ = *seq#*(G) for each of these groups. The sequencer handles a *DO-MULTICAST* message by incrementing the group sequence number in its context. It does not send the whole context in the *ACCEPT* message, but restricts it to entries for groups that intersect with the destination group:

on *DO-MULTICAST*(G, *id*, C_m, m) at sequencer S:

    if $\exists C'$: $\langle id, C'\rangle \in$ *historyBuffer* **then**
        send *ACCEPT*(G, *id*, C') point-to-point to sender;
    else
        $C.G := C.G + 1$;
        multicast *ACCEPT*(G, *id*, $C|$intersect(G), m);
        where intersect(G) = $\{G': G \cap G' \neq \emptyset \wedge$ *sequencer*(G') = $S\}$
    endif

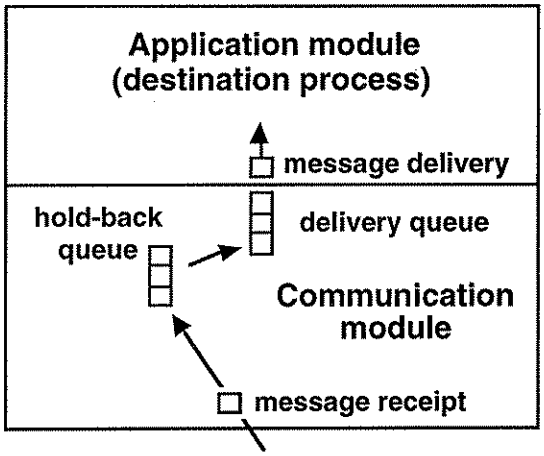

Figure 1. Message receipt and delivery

end

Note that for now we ignore the sender's context parameter. The next section explains the restriction of the intersection set to groups sequenced by the same sequencer.

The communication module at a destination process p keeps a *delivery context* D_p. Initially $D_p = \varnothing$. The communication module places arriving messages on the *hold-back* queue when they are not yet ready for consumption by p; it places deliverable messages on the *delivery queue* while they wait for p to consume them (see Figure 1). When it receives an *ACCEPT* message, the communication module checks the delivery context against the context of the *ACCEPT*-ed message.

The conditions for placing a message m with context C_m addressed to group G on the delivery queue are:

$$C_m.G = D_p.G + 1; \qquad \text{(T1)}$$

$$C_m | \hat{G} = D_p | \hat{G}, \text{ where } \hat{G} = \{G' : G' \neq G, p \in G'\} \qquad \text{(T2)}$$

– that is, for groups of which p is a member and which have entries in C_m, only the sequence number for G itself may differ between C_m and D_p, and then it must be one more than the last sequence number for G. (C_m may contain entries for groups of which p is not a member, but which intersect with groups containing p; we shall also see in a later extension that C_m does not necessarily include entries for all groups of which p is a member.) If a message becomes deliverable, the communication module updates the delivery state:

$$D_p := \text{merge}(D_p, C_m).$$

A process can detect missing messages by comparing the message's context against the delivery context, and request them using the group's port identifier.

Note that contexts are reasonably short in the majority of cases – that is, unless a group overlaps with many other groups. We shall shortly see that different sequencers may sometimes sequence intersecting groups.

4.2 Sequencing multiple groups with several sequencers

We turn now to the problem of how to sequence a set of groups, which may intersect, using several peer sequencers. Our goal is only to sequence two groups with the same sequencer if it is necessary to do so, and to balance the load between sequencers. The obvious constraint to apply is that two sequencers may separately sequence two groups only if they have no members in common:

$$\forall G, G' : \mathit{sequencer}(G) \neq \mathit{sequencer}(G') \Rightarrow G \cap G' = \varnothing \qquad \text{(S1)}$$

However, condition S1 is not necessary. Under total ordering, two groups may have one member in common and still be sequenced by different sequencers, because there is no other process with which to disagree about the relative ordering of two messages sent to the two different groups. The precise condition is:

$$\forall G, G'\text{: } sequencer(G) \neq sequencer(G') \Rightarrow |G \cap G'| \leq 1 \qquad \text{(S2)}$$

We call two groups with two or more common members *strongly intersecting*. Condition S2 states that strongly intersecting groups must be sequenced by the same sequencer.

In order to meet the specification S2, each sequencer needs to monitor *JOIN-GROUP* and *LEAVE-GROUP* requests to determine whether satisfying them will cause groups to become, or cease to be, strongly intersecting. An individual process p may be a member of groups sequenced by different sequencers, so no single sequencer can straightforwardly maintain the set of groups of which p is a member. Fortunately, we can surmount this difficulty by making each client include the set of groups containing it in *JOIN-GROUP* and *LEAVE-GROUP* requests. This is the *groupList* parameter in Table 2 above.

A sequencer checks the list of groups when processing a *JOIN-GROUP* message. If it does not sequence one or more of the groups, it must negotiate with the other sequencers involved as to which will sequence the groups affected[1]. A potential race condition arises, since two processes that are members of two groups sequenced by different sequencers may simultaneously apply to join the other group. Furthermore, it is possible for several pairs of groups to become strongly intersecting as a result of one process joining a group[2]. However, for simplicity of exposition we shall assume that at most one pair can become strongly intersecting. The following algorithm is for deciding whether a conflict occurs and handling it when processing a request to join a group:

```
on JOIN-GROUP(group G, process p, group list L) at S, where S = sequencer(G)
    if ∃G' ∈ L: S' ≠ S, where S' = sequencer(G') then
        if ∃q: q ≠ p ∧ q ∈ G ∩ G' then          // becoming strongly intersecting
            resolve(S', G, G')                  // agree same sequencer for G, G'
        else
            assert inIntersection(p, G, G')     // record overlap in table
        endif
    endif
end
```

The condition $q \in G \cap G'$ is tested by first consulting a local table to see if any process is known to belong to $G \cap G'$ (the statement 'assert *inIntersection*(p, G, G')' is to include an entry for G and G' in such a table). If this check proves negative, then the sequencer S requests S' to check whether it already knows of a process in $G \cap G'$ (two sequencers may communicate with one another simultaneously).

The procedure *resolve*(S', G, G') exchanges information between the two sequencers, in order to decide which shall sequence both the groups G and G'. The sequencers must transfer any groups that are strongly intersecting to whichever of G and G' they choose, in order to preserve the co-location of strongly intersecting groups. We require a load metric suitable for deciding whether to transfer G and its strongly intersecting groups to S', or G' and its strongly intersecting groups to S; but this is not addressed in this paper.

---

[1]The sequencer multicasts to its peers, and only sequencers that manage the other group(s) concerned respond.

[2]For example, let $G_1 = \{p,q\}$, $G_2 = \{p,r\}$, $G = \{q,r\}$. Now let p join G.

Returning to the definition of *intersect*(G) in the algorithm for handling a *DO-MULTICAST* message in Section 4.1, we see now why it was necessary to limit this set to intersecting groups sequenced by the same sequencer: a sequencer does not know the sequence numbers of other intersecting groups. We can make this restriction harmlessly, because sequence numbers of groups not in this set are irrelevant to the recipients, by the argument for using condition S2.

Note that we apply a similar procedure to that for *JOIN-GROUP* when handling a *LEAVE-GROUP* message: in this case, however, we decide whether two groups that were previously strongly intersecting can now be sequenced by different sequencers.

4.3 Transferring groups between sequencers

Each group is associated with some data needed for a sequencer to manage the group, and a port for the sequencer to receive requests pertaining to the group. Each group has its own port. To transfer a set of strongly intersecting groups between source and destination sequencers the following steps are necessary:

1. Suspend receipt of all requests related to the groups.
2. Transfer data pertaining to the management of the groups –

   membership list and address(es);

   sequence number;

   table of intersecting groups;

   identifiers and sequence numbers of last messages sent by clients.
3. Migrate the ports used to accept requests (one per group), to the destination sequencer.
4. Resume receipt of requests for the groups, at the destination sequencer.

After the transfer, the destination sequencer has all the information it needs to manage the group consistently with respect to its former implementation at the source. Note that the transfer procedure only sends enough information from the groups' history buffers to detect and acknowledge repeated *DO-MULTICAST* messages. If the group members do not miss any messages, as is likely on most LANs, then they will not require access to stored message bodies in the history buffer. If, however, the destination sequencer receives a *FETCH-MSGS* request that refers to messages still held at the source, then it may forward the request to the source. The destination must inform the source sequencer when it can delete the transferred groups' history buffers[3].

Senders experience a once-only cost when a port migrates. After a migration, the computer that originally possessed the port holds a forwarding pointer. When a process sends a message using the old port address, the message is forwarded to the new address. Moreover, the communication layer delivers an acknowledgement message (which may be piggy-backed on user data) to the sender, informing it of the port's new address. Thus the forwarding pointer is not used by that sender thereafter. Senders may multicast to the group of sequencers to locate the port, should they discover that the computer at the port's (stale) known address has become unreachable.

5 Causal and cautal ordering

We use the result of Florin and Toinard [1992] to adapt the totally ordered multicast protocol of Section 4.1 to give us hybrids of causal and total multicast ordering, including cautal ordering. Florin and Toinard showed that multicasts are causally ordered if:

each message m is stamped with a unique sequence number $n(m)$ (C1)

multicasts are FIFO ordered (C2)

[3] A sequencer may delete a message when it knows that all destinations have received it. For this purpose, group members occasionally send the current sequencer the highest sequence number they have received.

$$deliver_p(m) \rightarrow send_p(m') \Rightarrow n(m) < n(m') \quad \text{(C3)}$$

$$n(m) < n(m') \Rightarrow \forall p \in dest(m) \cap dest(m'): deliver_p(m) \rightarrow deliver_p(m') \quad \text{(C4)}$$

This result assumes use of a single series of sequence numbers and therefore a single sequencer, so we need to find a suitable interpretation for $n(m)$ to allow for several sequencers. We shall interpret $n(m)$ in terms of the context values used in Section 4.1. We require that each process p maintains a context C_p. Each process also maintains, as before, a delivery context D_p. Initially, $C_p = \varnothing$, $D_p = \varnothing$.

The rules for contexts transmission and update are as follows:

1) When p sends a *DO-MULTICAST* message, it encloses the value of C_p. It may not send a *DO-MULTICAST* request until any earlier requests have been *ACCEPT*-ed.
2) When p receives an *ACCEPT* message with context C_m for a message that it has sent to G, $C_p.G := C_m.G$.
3) When a sequencer receives a *DO-MULTICAST* message with context C_p, it calculates the new sequencer context C as before, but now it sends the context C_m = merge(C|intersect(G), C_p) in the *ACCEPT* message.
4) When p consumes a message m bearing a context C_m, C_p := merge(C_p, C_m).
5) As before, when a message bearing a context C_m is placed on the delivery queue, D_p := merge(D_p, C_m).

The conditions for placing m with destination G on p's delivery queue are:

$$C_m.G = D_p.G + 1 \quad \text{(CT1)}$$

$$C_m|\hat{G} \leq D_p|\hat{G}, \text{ where } \hat{G} = \{G': G' \neq G, p \in G'\} \quad \text{(CT2)}$$

Condition CT1 is the same as condition T1 in Section 4.1. By itself it ensures the total ordering of messages sent to G. Condition CT2, however, is a relaxation of T2 ('≤' replaces '=').

Now we define $n(m) = C_m$. Condition C1 is automatically satisfied since every two values $n(m)$ and $n(m')$ must differ at least in their destination groups' entries. For FIFO ordering (C2), let $G = dest(m)$ and let $C_m.G = N$. We assume that $dest(m') \neq G$, otherwise FIFO ordering follows immediately from rule (1) and by CT1. By rules (2), (1) and (3) applied in the generation of $C_{m'}$, $C_{m'}.G \geq N$. Assume that m has not been delivered at p. Then $D_p.G < N$ (the entry $D_p.G$ may only be incremented as a result of delivering a message destined for G, by rules CT1 and CT2). So $D_p.G < C_{m'}.G$ and m' cannot yet be delivered, by the delivery rule CT2.

To see that condition C3 applies, let C_p be p's context after it has consumed m and before it sends m'. Then $C_m \leq C_p$, by the rule (4) applied at message consumption; and $C_p \leq C_{m'}$ by the rule (3) that the sequencer applies. Therefore $n(m) \leq n(m')$, and so $n(m) < n(m')$ by condition C1.

Condition C4 is satisfied by the delivery rules CT1 and CT2. Suppose $n(m) < n(m')$. If $dest(m) = dest(m')$ then CT1 guarantees that m is delivered before m'. Otherwise, assume that $dest(m) \neq dest(m')$ and that m has not been delivered at p. Then $D_p.dest(m) < n(m).dest(m)$. But $n(m').dest(m) \geq n(m).dest(m)$, so $D_p.dest(m) < n(m').dest(m)$ and m' cannot yet be delivered.

This protocol provides cautal ordering: the messages sent to any given group are totally ordered, however the interleaving of messages delivered to processes in the intersection of two groups is consistent with causal but not total order. With this protocol, any sequencer may sequence any group. By substituting T2 for CT2 in the delivery criteria, and by sequencing strongly-connected groups together, we may obtain a multicast which is causal and strictly total – that is, which is total with respect to overlapping groups.

Increasing concurrency

The cautal and total-and-causal protocols just described restrict sending processes from multicasting a message while a previous multicast has not yet been *ACCEPT*-ed. This reduces the processes' capacity to execute concurrently.

We may remove this limitation by allowing processes to substitute the unique identifier of a message for its sequence number in contexts. We replace rules (1) and (2) above that govern transmitting and updating contexts, by a single rule:

- when p sends a *DO-MULTICAST* message to G, it sets the value of $C_p.G$ to be $SEQ(id(m))$, where $id(m)$ is m's unique identifier, and the $SEQ()$ operation denotes flagging the argument as being a message identifier, requiring transformation into the corresponding sequence number.

The remaining rules (3)-(5) remain the same, except that:

- sequencers and group members' communication modules replace unique identifiers in contexts by sequence numbers, as soon as these are known
- a group member receiving a message whose context contains unique identifiers places the message on the hold-back queue, until it can substitute unique identifiers
- a process merging two contexts cannot numerically evaluate the entry for a group unless both items are sequence numbers. Therefore it records *both* items in the entry for the group. Eventually, some process will substitute sequence numbers for unique identifiers, and choose a single sequence number as the maximum.

Each group member's communication module now retains a table of unique identifiers and sequence numbers of messages that it has already delivered. It may remove an entry from this table when it has received another message from the same client. (Recall that a unique message identifier consists of the client process identifier and a number drawn from a continuously increasing sequence.)

With these amendments, any process may send multicasts to any group without blocking for *ACCEPT* messages.

6 Related work

Florin and Toinard [1992] describe how to construct a causally-ordered multicast from a totally-ordered multicast, but they use a single sequencer. They also describe how to use a single sequencer for sending totally-ordered multicasts to the members of several groups. However, they achieve this by enclosing per-process information in messages. In our design, the size of the context information that is sent in messages is bounded by the number of groups, not by the number of processes – in general this is much less expensive in bandwidth.

In ISIS [Birman et al 1991], causal ordering is maintained by using vector timestamps. Again, this entails enclosing per-process information in messages. ISIS achieves total ordering – with respect to only a single group – by using a token-holding group member as a sequencer. ISIS manages only closed groups, whereas our design can accomodate open groups (at the expense of an extra point-to-point acknowledgement message, in addition to the multicast *ACCEPT* message). Open groups are common, including groups of servers and groups of processes that subscribe to published information.

The Amoeba protocol for totally-ordered broadcast [Kaashoek & Tanenbaum 1991] is designed only for use with a single group of processes. Message delivery is causal also, since senders await acknowledgment of the last broadcast from the sequencer before multicasting again.

The Total protocol [Melliar-Smith et al 1990] builds a total order from a causally-ordered multicast, using messages sent by a majority of members of the group. Delivery is subject to arbitrary delays when only a minority of group members send multicasts. Protocols such as

this are unsuited to groupware, in which users may observe but not update shared data for long periods of time.

7 Conclusions

This paper has described a reconfigurable sequencing service for group (multicast) communication, in which a collection of sequencers orders the messages sent to a set of groups, which may overlap. Our aim is to remove the bottleneck represented by a single sequencer. We have also motivated and defined a new hybrid multicast ordering, cautal ordering. We have defined criteria governing when two sequencers may sequence distinct overlapping groups, and we have described the reconfiguration that takes place when the sequencing activity for a particular group is passed from one sequencer to another. We have described protocols for total-and-causal and cautal ordering.

In these protocols, sequencers establish the total order immediately, regardless of whether all the group members continually send multicast messages. However, there is scope for improving the unmanaged way in which our protocol forwards context information even when it is not all needed, in case of overlap between groups (sequencers do not possess a global picture of the graph whose edges are the intersections between groups).

We are currently implementing the service, using C++ and UNIX, for a collection of Silicon Graphics workstations connected by an Ethernet. The implementation takes advantage of IP multicast.

The performance advantage that we hope to obtain over a single-sequencer approach depends, of course, on the characteristics of the applications. Our implementation is primarily intended for groupware, such as shared virual environments, in which many users browse or navigate through a collection of shared objects, and in which the users' views sometimes overlap and sometimes are disjoint. The sequencers may be processes executing at individual users' workstations, rather than a separate sequencing service used for many applications. In this case, the sequencing activity for a particular set of strongly-intersecting groups can be migrated to the process that currently sends the majority of multicast messages to the groups, if such a distinguished process can be found.

We shall test the protocol's performance over wide area networks, although we suspect that high-latency networks require a different approach, and may in fact not be able to sustain total ordering efficiently. We shall investigate the possibility of adapting the k-resilient version of the Amoeba protocol for improving the robustness of our protocols.

References

[Benford et al 1994] Benford, S., Bowers, J., Fahlén, L. and Greenhalgh, C. (1994). Managing mutual awareness in collaborative virtual environments. *Proc. Virtual Reality Software and Technology*, pp. 223-236.

[Birman & Joseph 1987] Birman, K. and Joseph, T. (1987). Reliable communication in the presence of failures. *ACM Trans. Computer Systems*, vol. 5, no. 1, pp. 47-76.

[Birman et al 1991] Birman, K., Schiper, A. and Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Trans. Computer Systems*, vol. 9, no. 3, pp. 272-314.

[Carlsson and Hagsand 1993] Carlsson, C. and Hagsand, O. (1993). DIVE – A platform for multi-user virtual environments. *Computers and Graphics*, vol. 17, no. 6, pp. 663-669.

[Coulouris et al. 1994] Coulouris, G., Dollimore, J. and Kindberg, T. (1994). Chapter 18 of *Distributed Systems - Concepts & Design* (second edition), Addison-Wesley.

[Deering and Cheriton 1990] Deering, S. and Chériton, D. (1990). Multicast routing in datagram internetworks and extended LANs. *ACM Trans. Computer Systems*, vol. 8, no. 2.

[Ellis et al 1991] Ellis, C., Gibbs, S. and Rein, G. (1991). Groupware: Some issues and experiences. *Communications of the ACM*, vol. 34, no. 1, pp. 38-58.

[Florin and Toinard 1992] Florin, G. and Toinard, C. (1992). A new way to design causally and totally ordered multicast protocols. *Operating Systems Review*, ACM, Oct. 1992.

[Kaashoek & Tanenbaum 1991] Kaashoek, F. and Tanenbaum, A. (1991). Group communication in the Amoeba distributed operating system.

[Kindberg 1992] Kindberg, T (1992). Reconfiguring Client-Server Systems, Tech. report no. 630, Dept. of Computer Science, Queen Mary and Westfield College, Univ. of London.

[Kindberg et al 1994] Kindberg, T., Sahiner, A. and Paker, Y. (1994). Adaptive Parallelism Under Equus. *Proc. Second International Workshop on Configurable Distributed Systems*, IEEE, Carnegie-Mellon University, March 1994, pp.172-182.

[Lamport 1978] Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system, *Communications of the ACM*, vol. 21, no. 7, pp. 558-565.

[Melliar-Smith et al 1990] Melliar-Smith, P., Moser, L. and Agrawala, V. (1990). Broadcast protocols for distributed systems. *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 17-25.