

Middleware Support for Distributed Multimedia and Collaborative Computing

KENNETH P. BIRMAN^{1†}, ROY FRIEDMAN^{*2‡}, MARK HAYDEN^{1†} AND INJONG RHEE^{3§}

¹*Department of Computer Science, Cornell University, Ithaca, NY 14853, USA*
(email: {ken,mark}@cs.cornell.edu)

²*Department of Computer Science, The Technion–I.I.T., Haifa 32000, Israel*
(email: roy@cs.technion.ac.il)

³*Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, USA*
(email: rhee@eos.ncsu.edu)

SUMMARY

Maestro is a middleware support tool for distributed multimedia and collaborative computing applications. These applications share a common need for managing multiple subgroups while providing possibly different quality-of-service guarantees for each of these groups. Maestro's functionality maps well into these requirements, and can significantly shorten the development time of such applications. In this paper, we report on Maestro, and demonstrate its utility in implementing several multimedia and collaborative computing applications. In particular, we provide a detailed description of the implementation of *IMUX*, a pseudo X-server (proxy) for collaborative computing applications that is based on Maestro. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: distributed multimedia; collaborative computing; middleware; group communication; quality of service

1. INTRODUCTION

Multimedia and collaborative computing are important application areas that benefit from the increase in computers power and networks speed and bandwidth. The primary tasks in building such applications include designing a good GUI, choosing the right coding and compression techniques [1], supporting various output devices and their corresponding drivers, and so forth. However, when operating in a distributed environment, with more than two participants, another dimension of complexity must be tackled. As reported in [2–5], distributed multimedia and collaborative computing applications often utilize several concurrent data streams, each potentially with a different quality of service and reliability requirements. On top of this, the system must be able to cope with dynamic changes in the environment, typically caused by processes joining, leaving, and crashing, and network partitions and merges.

Typical examples of distributed multimedia and collaborative applications include video/audio conferencing, textual chatting, white and clear boards, and application sharing

*Correspondence to: Roy Friedman, Department of Computer Science, The Technion–I.I.T., Haifa 32000, Israel.

†Supported by DARPA/ONR grant N00014-96-1-1014.

‡Most of the work was done while the author was with Cornell University.

§Supported by the Faculty Research and Professional Development Funds of North Carolina State University.

tools. In a video conference session, there can be a situation in which participants are connected by a WAN, but subsets of the participants may share the same LAN. Different subsets of the participants in a video conference may require different quality video streams, depending on their locale and equipment. We would probably want to use a video coding scheme similar to that used in earlier work [2–4]. Here, one stack is used for low quality frames, which should be delivered to all participants, while another is used for higher quality frames, and delivered only to participants running on nodes that are connected through high bandwidth links to the video sender. In addition, audio data should be delivered in FIFO order, but does not need stronger end-to-end reliability. Indeed, an attempt to overcome infrequent packet loss through a TCP-style flow control and acknowledgement mechanism might introduce undesired latency jitters and hence reduce the *perceived* reliability of the audio channel. On the other hand, text in the chat area should be delivered reliably and all participants should see postings in the same order.

One can also imagine configurations in which subsets of participants would maintain their own private chat groups or side-band conferencing sessions. Such a feature might be useful, for example, in a business negotiation that brings together multiple representatives of one organization in the context of a larger group of participants.

Application-sharing systems also demonstrate the need for multiple data streams with independent membership. For example, application multiplexors in the X Windows environment are typically implemented by interposing a ‘pseudo-server’ between clients and servers. The pseudo-server assumes the role of an X server when interacting with X clients and the role of an X client when interacting with an X server. When input operations are restricted to a single host, an X pseudo-server provides ‘output multiplexing’, allowing the output of a single application to be viewed simultaneously on multiple screens [6]. An alternate approach called ‘input multiplexing’ [7,8] replicates the client and employs a simplified pseudo-server. Output operations are passed to the local X server while input operations are multiplexed and communicated to remote replicas via the pseudo-server. A third approach periodically samples and transmits the client screen image. A collaborative environment can support all three mechanisms simultaneously by providing streams with different service qualities and memberships. In the spirit of the adaptive video tool, participants may subscribe to the service most appropriate for their needs. A single participant may even operate several applications simultaneously, each using a distinct application-sharing technique.

The need for multiple levels of quality of service within a single collaborative application matches naturally with the use of grouping mechanism [2–4,9,10], in which each group employs its own *protocol stack*. (A protocol stack is a collection of one or more communication protocols, stacked on top of each other, e.g. TCP over IP, that together provide a given functionality.) To facilitate the development of distributed multimedia and collaborative applications, we have developed *Maestro*, a middleware support tool for managing multiple process groups, each communicating over its own protocol stack.

In his book *The Mythical Man-Month*, Brooks [11] characterized the complexity involved in designing software as *essential* and *incidental*. The essential complexity is an inherent part of the problem the application is trying to solve, e.g. the algorithms themselves. The incidental complexity, on the other hand, stems from the use of inadequate interfaces, tools, and metaphors, which complicates the realizations of ideas. We believe that management of subgroups and their protocol stacks falls in the category of incidental complexity. We are therefore interested in providing tools to eliminate much of this complexity, enabling designers to devote their time to the more essential part of their work.

The management of process groups involves handling group membership changes, process failures, network partitions, and security. Since they have to be consistently handled with data delivery semantics that applications require, implementing these group management services both correctly and efficiently is a non-trivial task. Maestro provides these group management services as part of middleware with intent to relieve the burden of application designers to implement these services separately for each group-oriented protocol that they implement. Using Maestro, application designers can focus on QoS-related issues concerning their protocols without much concern on providing the related group management services for their protocols. An important property of Maestro is that the group management services are kept out of the critical path of application data communication, and therefore can provide its functionality without interfering with the qualities of service that applications require. In addition, Maestro efficiently supports multiple subgroups by allowing resources and services (e.g. failure detection) to be shared among subgroups.

Maestro is designed for applications supporting small to medium scale collaboration sessions, and does not scale very well for a large group involving more than a thousand participants. We believe that many Internet *synchronous* collaboration sessions are small-scale, and usually involve less than 20 participants. Any larger synchronous collaboration would resemble a fast video arcade game, as Fetterman [12] put it. Of course, when a large portion of participants are observers that do not introduce any input to the on-going session, then the session is certainly manageable. These types of sessions can be naturally supported by forming two separate groups where one group involves only the active members while the other group involve observers. Maestro can be used to support the former group because it can be configured to provide tighter synchrony and consistency guarantees on message delivery, membership changes, and failures. Such guarantees are not normally required for observers.

1.1. Paper outline

To provide the necessary background for this work, we provide a brief discussion about group communication systems in the next section. In that section, we also point to the benefits and shortcoming of group communication systems that motivate why we want to rely on these systems in the first place, but also explains what Maestro's functionality adds to them. Maestro is then described in detail in Section 3. A discussion of how applications can use Maestro appears in Section 4. In particular, we demonstrate in Section 4.2 how Maestro's features are exploited in the implementation of *IMUX* [8], a pseudo X-server (proxy) which transforms X-based applications written for a single-user into shared multi-user applications, without changing their code. The implementation of Maestro using the Ensemble group communication system [13,14] is reported in Section 5, where we also present performance data. The paper concludes with a comparison of our work to prior approaches in Section 6 and a discussion in Section 7.

2. BACKGROUND: GROUP COMMUNICATION SYSTEMS AND VIRTUAL SYNCHRONY

In this section we give a brief overview of group communication in general, and of the Ensemble system in particular. At the end of this section we also explain the limitations of existing group communication systems, including Ensemble, w.r.t. building applications that require multiple communication channels with various QoS guarantees, which is the core

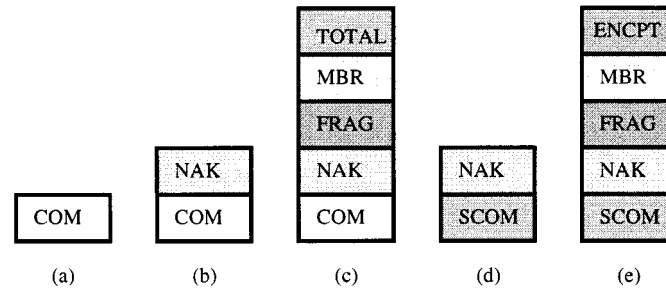


Figure 1. One application might operate over a set of side-by-side protocol stacks, each specialized to a different need. The stacks above provide: (a) unreliable unordered communication, (b) provides FIFO reliable delivery, (c) provides totally ordered virtually synchronous communication, (d) provides authenticated FIFO delivery, and (e) provides authenticated and encrypted virtually synchronous communication. Notice that the ENCPT layer could also be stacked lower in the stack, to encrypt headers created by layers above it.

motivation for building Maestro. This section is given here for completeness, and readers familiar with this area might wish to skip to the next section.

Designing and implementing reliable distributed applications is an inherently complex task, which group communication systems or toolkits are intended to simplify. These systems provide interfaces for creating process groups, for sending messages to members of the group, or multi-casting them to the entire membership, and therefore permit use of the group abstraction as an addressing domain. These systems also allow processes to join and leave groups on-the-fly, and provide mechanisms by which detected failures can trigger membership changes. Examples of group communication systems include, Consul, Delta-4, Horus, ISIS, Phoenix, Relacs, RMP, Transis, TOTEM, and the V system. A general treatment of the area can be found elsewhere [15].

Our own work was based on the Ensemble system, which supports a communication model called *virtual synchrony* [15,16]. The details of the model will not be important in this paper, except insofar as they permit consistent, secure, and fault-tolerant actions at sets of communication endpoints. For example, an Ensemble process group can coordinate the installation of a session key at each of a set of participating processes. Ensemble reports membership information in a consistent manner: processes belonging to a process group all see the same membership information, and see changes to it in the same order. Ensemble groups can be used to replicate data or other forms of information. In the settings of interest to us here, these might include quality of service expectations that would be used to set up communication stacks. For example, one might want all members of an application to add a data encryption layer to a stack, or a special data compression method. By coordinating such actions in a consistent, fault-tolerant manner, Ensemble eliminates the need for the application designer to develop specialized solutions to these problems.

Ensemble, like its predecessor Horus, has a layered architecture. Group communication functionality is decomposed into a number of orthogonal ‘properties’, each supported by a micro-protocol and implemented as a software layer. The layers can be assembled into a stack having the mixture of properties needed for a particular application. This gives flexibility, as protocol layers can be combined in many ways, and only the layers that are needed for a specific functionality are used in a protocol stack. Any given stack enforces roughly the same set of properties to all messages that are communicated using it. Examples of stacks are given in Figure 1.

One consequence of this architecture is that one could configure a virtually synchronous protocol stack for use where coordinated, consistent behavior is desired, while building a second protocol stack focused on some other requirement, such as isochronous transmission of video data.

Although Ensemble is very flexible, there are also protocol stacks external to our architecture that an application might wish to employ. These include, for example, protocol stacks providing special quality of service properties for ATM links (the U-Net architecture [17] can be viewed this way). One is led to a view of application programs that operate over sets of stacks, which they need to configure and control in a consistent manner. This need is precisely where Maestro comes to play! Group communication systems lack the ability to manage multiple communication stacks in a consistent way and Maestro adds this capability.

3. MAESTRO

When using Maestro, a single management group, known as the *core group*, is configured to include all participating processes within an application, or a set of related applications. This core group uses a virtually synchronous protocol stack [16], which we call the *core stack*[¶]. Processes can create subgroups that communicate over their own protocol stacks, called *data stacks*, which can provide quality of service guarantees differing from those of the core stack. In particular, these protocol stacks may be either Ensemble multicast stacks [13], or external stacks, such as TCP/IP [18], Cyclic-UDP/IP [1], user level network interfaces [17], ISIS [16], Horus [19], raw ATM, CCTL [10], and so forth. However, the creation of such subgroups is ‘announced’ by multicasts within the core group (and core stack), as are join requests to subgroups, leave requests, and subsequent membership changes for data stacks. Also, subgroups can choose between virtually synchronous membership and asynchronous membership service, all within the same framework and interfaces.

Maestro can execute in various configurations, including as a library residing in the same process as an application (as illustrated in Figure 2), as a stand-alone server, or both at the same time. This flexibility comes about because the service it provides can be abstracted as an event stream between Maestro and the application. When Maestro is used as a library, interactions between it and the application occur directly through function calls. On the other hand, when Maestro is used as a server, the application(s) must communicate with Maestro through a TCP connection, which also has the advantage that application(s) do not have to reside on the same machine as the Maestro server. Note that using Maestro as a server does not significantly degrade performance because data messages are sent directly between application data-stacks and only messages regarding membership are communicated to Maestro.

In our approach, the membership of a subgroup must be a subset of the core stack membership, and the exclusion of a failed member from the core stack triggers exclusions from its subgroups. In many cases, this frees subgroups from the need to monitor the health of their members, reducing communication overheads. On the other hand, if a subgroup needs to drop a member, it can request a change to its own membership or, by sending an input to the failure detector of the core stack, can cause that member to be excluded from the core stack as well. Maestro thus provides a flexible and efficient failure reporting mechanism, leaving the application to implement the most appropriate failure detection policy for the core stack

[¶]The core group needs secure and reliable communication between its endpoints, and also need to be able to maintain a consistent view of the state of the system. These properties are provided by the virtual synchrony model, as discussed in Section 2.

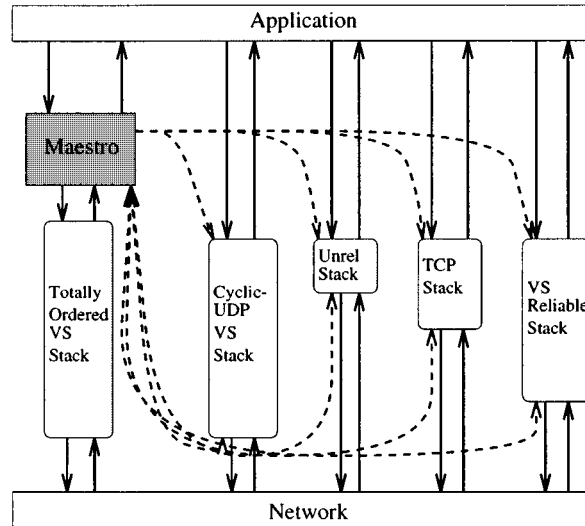


Figure 2. Maestro system configuration.

and data stacks of subgroups. We view the details of how failures are detected as application-specific, and beyond the scope of this paper (but see Vogels [20]).

To reduce redundant communication, data stacks that send periodic messages, e.g. update messages necessary for implementing a NAK protocol, may register such events with the core group. The core group will aggregate all such events and send them in a single message. This feature can substantially reduce network traffic and processing time devoted to sending and receiving messages.

To simplify and automate the architecture, Maestro introduces a notion of core member *properties*. The core group membership includes a list of properties of each member, which are simple ASCII strings, and subgroups can be configured to automatically track subsets of core members that have a desired set of properties. For example, core members might specify a property such as 'system administrator' or 'has an ATM connection'. A subgroup can then automatically be created containing just those members that have ATM connections, or those system administrator processes that also have ATM connections. By automatically adjusting subgroup membership in these common cases, Maestro provides the application developer with an easily exploited facility for creating desired subgroups. For many developers, this eliminates the need to implement special logic for subgroup membership management.

Also, the interface to Maestro provides support for adding new members on-the-fly and merging network partitions by informing the new members about the existing subgroups and their properties. Maestro also provides hooks for the application to do more elaborate *state transfers* if needed.

3.1. Application-defined 'data types'

To interact with Maestro, the application defines several data types that specify the properties of groups and uniquely identify communication endpoints and groups. These data type are represented as uninterpreted sequences of bytes of arbitrary length, though the group

and endpoint identifiers are usually quite small. The only operation Maestro applies to the sequences of bytes are tests for equality among endpoints or groups.

- (a) **group identifier:** these are used to uniquely identify a communication group. As with the endpoint identifiers, these may contain addressing information for the group. For instance, they include the Internet address and port of an IP multicast group.
- (b) **endpoint identifier:** these are used to uniquely identify communication endpoints. The same endpoint may join any number of groups, but may not join the same group twice. Additionally, a single process may have several endpoints.
- (c) **group properties:** properties are used by the application to advertise a group to other application instances, so that they can determine whether or not to join the group. The properties are usually a record with fields describing the group's ASCII name, security information, the type of the group, the expected bandwidth of the group, a list of members expected to join. They can be extended with other application-specific information.

3.2. Application–Maestro interface

The first part of the interface is used for sharing information about groups between application processes using Maestro. Through this interface, an application can create new groups, associate application-specific properties with each group, and announce groups to other application instances. The announcement facility allows applications to advertise new groups and other instances use the information to decide whether to join the groups. At initialization time, the application provides a generic group `new-group` handler function to Maestro. This handler is invoked with group identifiers and group property information when new groups are announced.

In creating a new group, the application performs several steps. First, it allocates a new group identifier and specifies the properties of the group. The application then requests Maestro to announce the new group to other members through the `create-group` downcall with the group identifier and the properties. Other members are notified of the new group through their `new-group` upcall. This upcall asks each member to decide whether or not to join the group.

Group properties include a (possibly empty) list of properties. If the list is non-empty, then members having all of the specified properties will automatically be joined. A second (possibly empty) list of properties identifies members that should be informed about the group. If this list is non-empty, only members that have the specified property, e.g. 'do not have an ATM card', will receive the `new-group` upcall. If the list is empty, all members will be notified. At present, properties are uninterpreted byte-sequences, and Maestro limits itself to equality testing. We are considering adding a more elaborate support for handling properties, if experience with the system indicates that it is needed, but our work so far suggests that the present simple mechanism is sufficient for most anticipated uses.

Maestro maintains a history of announced groups, so that processes joining the system (or rejoining after a network partition) can be informed of the active groups in the system. When a group is no longer needed, the application can call `destroy-group` with the name of the group. When this is done, Maestro will garbage collect its history information and cease to announce the group.

3.2.1. Summary of the Application–Maestro interface

- `create-group`: Called by the application to announce a new group in the system. Other members are notified through their `new-group` callback.
- `new-group`: Called by Maestro to notify the application of a new group.
- `destroy-group`: Called by the application to prevent the future announcement of a group.

3.3. The data Stack–Maestro interface

In many applications that we have considered, Maestro's automatic subgroup membership tools are all that is needed to manage subgroups. However, applications may also decide to join a group as a result of a `new-group` announcement or other events. In such cases, a group is joined by calling `join-group` with an identifier for the group and several other arguments. The additional arguments include the priority of the group (a number between 0 and 255), whether or not the data-stack will support virtual synchrony, and security keys. The `join-group` function returns a new group object on which the application receives updates about the status of the group from Maestro. Note that the application receives one group object per subgroup. However, while multicast and send downcalls by the application are automatically directed to the appropriate data stack, membership operations (such as for joining and leaving groups) are sent to Maestro. As for outgoing messages, incoming messages received by a stack will automatically invoke their corresponding application-level receive functions, without being diverted through Maestro.

Whenever Maestro detects a change in the membership of a subgroup, it reports the change to the stack, since some of Ensemble's internal modules rely on knowledge of the membership. For instance, this knowledge is used by the stack's interface to the network to filter out messages from non-members. Changes to the membership of the group are sent to members via the `group-view` callbacks to the group object.

Members can manually remove other members through the `fail-group` callback. Normally, Maestro automatically handles failure detection and this sidecall is used by the application only to leave the group by 'failing' itself. However, some applications wish to be able to have members remove other members. For instance, application processes may have additional information about failures (some form of external failure detector) which allows them to detect failures more rapidly than Maestro. In this case, the application may decide to fail members and not wait to Maestro to do so. Another use is to remove other members that are not meeting a quality of service requirements for a subgroup or that do not have the required security authentication for a subgroup. Applications need to be careful in failing other members, however, as a group can rapidly disintegrate if many members start to fail each other.

To guarantee virtual synchrony for subgroups that require it, Maestro first synchronizes with all the members before installing a new view. Maestro first sends a `group-sync` event to all group members via the group object. A subgroup that requires virtual synchrony must then wait for all messages sent over its stack to be acknowledged, and then reply with the sidecall `synced-group`. From this point on, subgroups that require virtual synchrony should not send new messages until they receive the `group-view` sidecall (if the applications tries to send such messages, it will be forced to wait until the view change is completed). A more detailed description of how virtual synchrony is guaranteed appears in Section 5. Synchronization increases the cost of view changes but many groups, such as with

multimedia applications, do not require synchronous views, and can consequently ignore this constraint. Thereafter, sending messages over that stack and receiving messages from it are done directly by the application. However, there are still some cases where Maestro needs to communicate with this data stack, as outlined below.

A few protocols need to send periodic messages, although these messages do not have to go out at fixed times. Examples of such protocols are broadcast stability detection and the NAK protocol (NAK protocols must send periodic updates to overcome the 'loss of the last message' problem in pure negative acknowledgement protocols). Since many stacks generate such events, Maestro allows data stacks to register them using the `cast-group` sidecall. Maestro then periodically sends aggregated events to other nodes. Upon receiving an aggregated event from another node, the local instance of Maestro distributes these events to the appropriate data stacks using the `receive-events`. With large groups, Maestro can improve the efficiency of the protocols by distributing the events using hierarchically structured protocols. For instance, in order to detect broadcast stability, the minimum number of messages acknowledged by each member must be determined. Ensemble can structure the dissemination and computation of this information to be much more efficient.

In practice, periodic events on data stacks are generated with the `ENS.LAZY` option. Once such an event reaches the transport layer, instead of sending it over the network, the transport layer invokes the `cast-group` sidecall. Similarly, when a periodic event reaches Maestro, it passes it to the transport layer of the appropriate stack using the `group-cast` sidecall, and from there it is propagated up the stack as if it was received from the network.

3.3.1. Summary of data stack to Maestro sidecalls

- `join-group`: called by the application to join a group. Maestro will call back later with a `group-view` callback.
- `fail-group`: called by the application in order to fail another member in the group (or to leave the group by 'failing' itself).
- `synced-group`: called by the application in response to a `group-sync` sidecall.
- `cast-group`: called by the application to register a message to broadcast to the subgroup. Other members receive the message as a `group-cast` sidecall. The message will be sent lazily so that Maestro can aggregate them.

3.3.2. Summary of Maestro to data stack sidecalls

- `group-view`: called by Maestro to inform the member of a new membership view of the group. All changes in the membership of a subgroup are reported by Maestro to the application using the `group-view` upcall, which includes the list of new members and the rank of the local node in it.
- `group-sync`: called by Maestro in synchronous groups to request a `synced-group` callback from the group member. The new view will be delivered when all live members have responded.
- `group-failure`: called by Maestro in synchronous groups to inform the member of other members that have failed (or left the group).
- `group-cast`: called by Maestro to deliver a message broadcast via `cast-group`.

3.4. Maestro's membership properties

Writing a formal definition of group membership, and in particular for virtual synchrony, is a difficult task, which can be a topic for a paper by itself. (See the frenzy of papers on virtual synchrony in recent years [21–26].) In this paper, we only provide an informal definition of the main membership properties provided by Maestro.

For asynchronous groups, the following properties are guaranteed:^{||}

- **Views consistency:** eventually, all members of a subgroup see the same set of views and in the same order.
- **View uniqueness:** each view has a unique identifier, a unique coordinator, and a unique membership list associated with it.
- **Non-triviality of views:** every endpoint p that sends a `Join` request to a coordinator, eventually becomes a member of the corresponding subgroup unless either the coordinator is eliminated from the core group, or some member declares p faulty.
- **No spontaneous view changes:** a member p is removed from a view only if some member declared p faulty; a member p is added to a view only if the coordinator of the subgroup received a `Join` request from p .

For virtually synchronous groups, Maestro provides the following property as well:

- **Agreement on messages between views:** all messages must be delivered within the view in which they were sent.

For the sake of brevity, we avoid a detailed discussion of each of these properties. However, it is fairly straightforward to see that the above properties are useful for maintaining consistency among members of the same application, and for the application to obtain meaningful information about the configuration of the system from Maestro. (Recall that the configuration of the system is reported to the application by Maestro using view upcalls.) The interested reader may find more elaborate discussions of membership properties elsewhere [16,22,31,32].

4. APPLICATIONS

Maestro's architecture is motivated by the structure of many distributed multimedia and collaborative computing applications. In Section 4.1, we now describe how Maestro helps in managing the communication needs of these applications, and then in Section 4.2, we demonstrate in detail how Maestro's features are exploited in the implementation of *IMUX* [8], a pseudo X-server (proxy) which transforms X-based applications written for a single-user into shared multi-user applications, without changing their code.

4.1. Collaborative multimedia applications

CSCW systems present the user with a rich media mix including text, files, audio and video. A typical collaboration session involves audio and video conferencing, application sharing, textual chatting, and reliable file transfer. CSCW systems support (soft) real-time multimedia transmission as well as reliable collaboration and sharing. Building tools to support various

^{||}In these specifications we only require eventuality of membership changes, rather than imposing explicit timing restrictions. This makes the specification independent of timing assumptions, and is in line with other asynchronous group communication system specifications [24, 27–30].

collaboration activities requires to employ various communication protocols offering different service qualities. For instance, chatting and file transfer tools may be implemented with TCP/IP while video and audio tools with RTP on top of UDP. Since these tools supports simultaneous use by multiple clients, their communication protocols must also incorporate group communication.

As a collaboration session creates needs for multiple service quality protocols, individual tools also exhibit the need for employing multiple subgroups. Below, we examine several collaborative multimedia applications to show the utility of Maestro in building these tools.

Consider an adaptive video conferencing tool [33] as an example of a CSCW application with demanding group communication requirements. In a fair and scalable transmission scheme, image quality can be modulated to match receiver capacity where capacity depends on processor speed, load and network bandwidth. Fairness and scalability can be ensured by splitting a single data stream into multiple components of differing video qualities. Receivers combine components to synthesize an image with a quality appropriate to the available transmission rate [34,35]. Receivers may reconfigure their quality of service by joining or leaving streams in response to changing network load. This scheme requires the underlying communication system to support multiple subgroups with independent membership and service qualities, and subgroup membership operations (joining and leaving) must be very efficient.

Application-sharing systems also demonstrate a need for multiple data streams with independent membership. Application multiplexors in the X Windows environment are typically implemented by interposing a 'pseudo-server' between clients and servers. The pseudo-server assumes the role of an X server when interacting with X clients and the role of an X client when interacting with an X server. When input operations are restricted to a single host, an X pseudo-server provides 'output multiplexing', allowing the output of a single application to be viewed simultaneously on multiple screens [6]. An alternate approach called 'input multiplexing' [7,8] replicates the client and employs a simplified pseudo-server. Output operations are passed to the local X server while input operations are multiplexed and communicated to remote replicas via the pseudo-server. A third approach periodically samples and transmits the client screen image. A CSCW environment can support all three mechanisms simultaneously by providing streams with different service qualities and memberships. In the spirit of the adaptive video tool, participants may subscribe to the service most appropriate for their needs. A single participant may even operate several applications simultaneously, each using a distinct application-sharing technique.

Group communication can be used to implement collaborative distributed systems composed of interacting objects with a resulting unification of naming and object invocation mechanisms. Building distributed systems as a collection of cooperating distributed objects is a well-established paradigm. Group communication can be used to unify naming and object invocation in such an environment [36]. If each distributed object is associated with a unique multicast group, the group name can be used to uniquely reference the object. Method invocation on such objects can be implemented by joining the associated multicast group and requesting the appropriate operation. This scheme can be used to implement shared files systems or data spaces common in collaborative environments [37], but a large number of multicast groups may be required. Once again the common requirement of the communication substrate is the ability to support a large number of multicast groups with efficient, independent group membership operations.

Maestro greatly simplifies the construction of subgroups with different membership and service qualities. The modular design of Maestro removes group management service routines

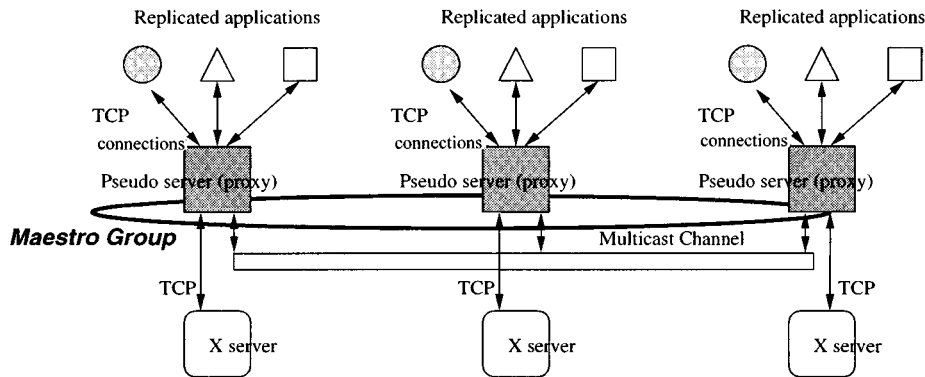


Figure 3. The IMUX application sharing tool. Here clients are replicated at each participant's site, and output operations (e.g. drawing commands) from clients are routed to the local X server while input events (e.g. user inputs) are multiplexed and communicated to remote replicas via the pseudo-servers. Since all replicas see the same set of inputs, they generate the same set of outputs to their corresponding X server, leading to the same view of the shared applications.

from the individual subgroup protocol stacks, and provides hooks for a subgroup protocol stack to use its group management services. Therefore, a protocol can be easily made group-aware using Maestro, allowing application designers to easily construct a protocol stack by focusing only on the data communication quality of the protocol. Maestro defray most of the cost for setting a process group when a process first joins the Maestro core group. Therefore, joining and leaving individual subgroups is very efficient. For example, when the new member first joins the Maestro group, it undergoes a security clearance (if required). Thus joining a subgroup may not require any expensive authorization/authentication operation.

4.2. Case study: The IMUX pseudo X-server

Synchronous collaboration commonly involves a group of participants simultaneously sharing the control and view of same applications through a WYSIWIS ('what you see is what I see') interface. Application-sharing tools aid synchronous collaboration by allowing an application designed for a single user to operate in a multi-user environment without modifying the application's source code. One approach for application-sharing, especially under X windows, is to interpose a *pseudo-server* (or *proxy*) between clients and servers. A pseudo-server assumes the role of an X server when interacting with X clients and the role of an X client when interacting with an X server, as illustrated in Figure 3. We call this approach *input multiplexing* (IMUX) [8].

Maestro greatly simplifies the implementation of IMUX. While clients and X servers communicate with pseudo-servers through TCP connections, the pseudo-servers are usually engaged in a multiway communication among themselves. By forming a Maestro multicast group of pseudo-servers, their multiway communication can be supported through reliable multicast stacks that are managed by Maestro. Maestro's virtually synchronous communication model [16] also simplifies dealing with dynamic changes in the system, such as crashes and joins. In addition, the various message ordering services of Maestro can be used to implement different types of synchronization mechanisms for handling simultaneous input events from multiple users. In the following, we report on how these Maestro group management services are used in implementing IMUX.

Starting an application

As mentioned, all pseudo-servers in the same collaborative session form a Maestro core group. For each shared application, the pseudo-servers create a new subgroup that we refer to as a *channel*. Using the Maestro's automatic subgroup join service and its property specification, a pseudo-server creates a channel for each application before starting the shared application. All pseudo-servers in the core group automatically become members of the channel when it is created, and get a notification from the Maestro server about the new channel creation, which causes the servers to start the application. When the application initiates a TCP connection to its pseudo-server, the pseudo-server also initiates a TCP connection to a local X server, posing as an X client. The pseudo-server then spawns an application thread that listens to these TCP connections and the channel created for the application.

By assigning different channels to different shared applications, the pseudo-server avoids the overhead for multiplexing messages to different applications. Another advantage of independent subgroups is that a user may elect to leave a channel based on its own need, possibly economizing on its resource usage. In this case, its pseudo-server would automatically stop receiving messages pertaining to that application.

Synchronizing inputs

Synchronous collaboration typically involves multiple users introducing input events to the same application simultaneously. Each user is at a desktop containing all the replicated applications, and the desktop is supported by one pseudo-server. Figure 3 shows a three-user session. In this environment, pseudo-servers have to ensure that all the replicas of the application are given the same sequence of input events so that they all have the same state and view. Two approaches are possible: *token passing* and *total-ordering*.

In the token passing approach, a token for each shared application circulates among users. A user that wishes to control the application must grab a token first and only then introduce input events. Only the input events from the token holder are routed to all pseudo-servers which feed the events to their own clients. A user may communicate its wish to use a token to all members, and the current token holder releases the token to the requesting user when it is finished using it. As the approach is simple and does not require a particular ordering among messages from different users, a FIFO reliable multicast service is sufficient to transport the input events and the token. However, in this approach, users have to tolerate the latency between their token requests and receptions.

The total-ordering approach allows all the users to introduce their own events at any time. All the events are totally ordered through total-ordering multicast so that each IMUX will feed the same sequence of inputs to their clients. One difficulty of this approach is that some X events need to be grouped together to be meaningful. For example, a button press event is always followed by a button release event. If two users try to click on a window region at the same time, it is possible that two button press events are ordered before any of their corresponding release events, which might confuse their clients, possibly leading to an erroneous operation. If this happens, the pseudo-servers must look for matching events from the same user and place them together before any other conflicting events. Since all pseudo-servers perform the same operations on the same sequence received from the total-ordering multicast channel, the 'reordered' sequence will be ordered the same everywhere.

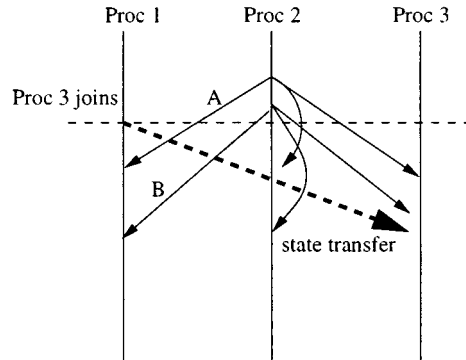


Figure 4. Latecomer in a virtually synchronous environment.

These different types of input control can be supported at the same time as per-application basis. The token passing approach is appropriate for a collaborative session with many users controlling the shared applications while the total-ordering approach is for interactive collaboration with a fewer number of controlling users.

Handling a latecomer

In a synchronous collaborative session, it is common to have latecomers joining an ongoing session. A latecomer must update its state to be the same as other members of the session. We support this by having each pseudo-server store the entire input sequence for each running application. Then, whenever there is a change in the membership, one of the existing pseudo-servers, known as the *leader*, sends the stored sequences to the new pseudo-server, which then plays back those events to its local clients.

The virtually synchronous membership service of Maestro simplifies things here too. Under virtual synchrony, each group member recognizes a new member join at the same logical time and receives the exact same set of messages between any two membership changes. This guarantees that the sequence of events sent to the latecomer are up to date with the state at the time it joins, and that the latecomer will take the exact same actions as the existing ones, and will reach the same state, as illustrated in Figure 4.

We would like to point out that X input events consist of only 48 bytes each. Hence, the buffer requirements needed for this are not too large. However, in a long-running session, the need to play back all these events may not be feasible. There are known techniques to reduce the number of stored events [38], but brevity precludes us from discussing them here.

Handling a failure

Maestro makes it easy to handle the failure through consistent failure detection and notification. When a process fails, Maestro guarantees that all other members will be notified about it through a consistent membership view change. In particular, Maestro ensures that all processes receive the same set of messages from the failed member, and can take consistent decisions on how to recover from the failure, e.g. by revoking a token held by the failed node.

5. IMPLEMENTING MAESTRO USING ENSEMBLE

Maestro is implemented using the Ensemble group communication system [13]. We describe here the overall protocol structure used by Maestro, as well as several issues in the implementation. These issues include performance and fault tolerance.

Recall that Ensemble already provides a group membership mechanism. However, Ensemble only manages the membership of the core group. The goal of Maestro's protocol is to extract the membership of each of the subgroups from the membership of the core group. Ideally, whenever the membership of the core group changes, we would like Maestro to be able to deduce any corresponding changes in subgroups locally. On the other hand, if a process that is included in the core group joins or leaves only a single subgroup, we would like to keep this transparent to other subgroups. Also, in the latter case, since the core group remains intact, and since the core group implements a virtually synchronous stack, we would like Maestro to utilize it, and therefore run a much simpler code than the one required to implement membership over a raw network [16,29]. The implementation we describe below follows these guidelines.

5.1. Informal protocol description

Maestro is an Ensemble application. Both are implemented in the Objective CAML dialect of the ML programming language, although both are accessible from other languages such as C, C++ and Java, and are fully portable to most UNIX variants and to Windows NT. The total size of Maestro is less than 800 lines of code. The protocol structure for managing subgroups is divided between two types of entities: there is a single *coordinator* for the entire subgroup, and one *member* for each endpoint that participates in the subgroup. Naturally, the members are associated with data stacks that have joined the group, and they reside in the Maestro process responsible for that endpoint. The coordinator is associated with the entire group and may reside in any process that is a member of the core group, even if that process is not a member of the subgroup.

The coordinator makes centralized decisions about the subgroup based on information received from the members. It keeps track of the current view of the subgroup as well as a list of currently synchronized members awaiting a view change. When a member joins a subgroup, the coordinator initiates a view change in order to add the member to the group, and similarly it causes view changes when a member leaves the group or fails. Information from the members is sent to the coordinator in point to point messages and the coordinator broadcasts back to the members information about new views.

Members have two responsibilities. First, they forward operations from the coordinator to the data-stack, and *vice versa*. Second, they maintain enough state so that when the coordinator fails, the members together can reconstruct the state of the failed coordinator and start a new coordinator. This state includes information such as the member's unique identifier and whether the member is currently synchronizing for a new view. When the coordinator fails, this information is collected by Maestro and used to generate a new coordinator.

The view change protocol for subgroups is implemented primarily by the coordinator. When the coordinator decides to initiate a new view as a result of a join, leave, or failure, it broadcasts a *Sync* message to all the members. The members forward this to the data-stacks, which eventually reply to the member that they are synchronized**, and this reply is forwarded to the coordinator as a *SyncOK* message. When every member is

**If the data stack is not virtually synchronous, it can reply immediately.

either synchronized or detected as faulty, the coordinator broadcasts a *View* message to the members. If the member is listed in the view, then it forwards it to the data stack. The FIFO virtual synchrony property on the core group guarantees that all members see the same sequence of views [16]. This is a useful property that greatly simplifies the Maestro protocol compared to virtually synchronous membership protocol built directly on top of asynchronous networks [16,23,39].

When there are no failures, the view protocol consists one point to point message for each member of the subgroup and two broadcasts by the coordinator to the entire core group. Although the broadcasts are sent to more processes than are necessary, message packing techniques [29] are used so that the *Sync* and *View* broadcasts for different subgroups can be aggregated and sent in a single message. This reduces the overall load when multiple group changes occur simultaneously, which is often the case when new applications join multiple subgroups.

A new coordinator is chosen for a subgroup *A* when a view change removes the previous coordinator of *A* from the core group. All the members transfer their state to the new coordinator, and it uses this information to reestablish the state of the old coordinator. This usually implies a view change in the subgroup. However, if a core group view change does not add or remove any members within a subgroup, then that subgroup's view does not change.

The coordinator of a subgroup *A* can reside in any process that is a member of the core group, even if no member of *A* is located in that process. In particular, this situation allows to keep track of subgroups that were abandoned by all their members, so that in the future they can rejoin the subgroup. On the other hand, placing the coordinator in a process that includes members of the subgroup, whenever possible, yields better efficiency; by doing so, changes to the core group that do not affect the subgroup are not communicated to the subgroup, while changes in the subgroup's membership that do not affect the core group are only communicated among the relevant processes. Thus, the coordinator is started by default in the same Maestro process as the first member of the group. However, when the subgroup membership changes, Maestro may migrate the coordinator to another process, if there are no additional subgroup members in the process that holds the current coordinator.

5.2. Detailed protocol description

The Maestro group management protocol is described in this section. This includes the portion involved with managing the membership and synchronization of subgroups, but not group announcement or event registration. The implementation of the latter services provided by Maestro are relatively straightforward, so we have elected to focus on the more difficult portions. We also limit the presentation to portions of the protocol necessary to support a single sub-group. It is a simple matter to generalize this protocol to multiple groups. The pseudo-code we give is a slightly transformed version of the source code for the actual implementation.

We divide the protocol description into four parts. The first describes the messages; the second describes the protocol for the normal operation of the coordinator; the third is the normal operation of the members; and the fourth is the combined member-coordinator protocol for core-group view changes.

5.2.1. Messages

Messages are broken into two groups: those going from members to the coordinator (*member messages*) and those going from the coordinator to the members (*coordinator*

messages). Member messages are of three types:

Join: This is a request to join the group. The coordinator replies with a *View* message.

SyncOK: this notifies that the member is synchronized. It is a reply to a *Sync* message from the coordinator, and is replied with a *View* message.

Fail(endpoint): report a member as having failed. The coordinator will remove that member from the group.

Coordinator messages are also of three types:

View(view,ltime): A new view is being installed. The *view* is a list of endpoints. The *ltime* is the logical time of the view. The *ltime* along with the view identifier of the core group uniquely identify the view of the subgroup.

Sync: all members should ‘synchronize’ and then reply with a *SyncOk* message. In virtually synchronous stacks this usually means waiting for broadcasts to stabilize. In other cases, the *SyncOk* message can be sent immediately.

Failed(endpt): a member is being reported as having failed.

5.2.2. Coordinator

The coordinator’s state is represented by a record. Note that the coordinator’s state is kept in only one of the Maestro processes. The state record consists of fields for keeping track of the data group’s view. Based on this information, the helper function *coord_check_view* can determine when a new view should be installed in the group. This is done by incrementing the logical time, resetting the list of synchronized members, and sending out the new group view.

```
(* COORD: Record containing the state of a coordinator.
 * There is one such coordinator per group.
 *)

type coord = {
  syncing : bool ;                               (* am I sync'ing? *)
  ltime   : int ;                               (* view's logical time *)
  alive   : endpt list ;                       (* endpoints in the group *)
  syncnd  : endpt list ;                       (* who is sync'ed *)
}

(* COORD_CHECK_VIEW: Helper function for coord_recv. Check
 * if it is time for the coordinator to install a new view.
 *)

function coord_check_view c =
  if (c.syncing and is_superset c.syncnd c.alive) then (
    c.syncnd := [] ;
    c.ltime := succ c.ltime ;
    c.syncing := false ;
```

```

    c.broadcast (View(c.ltime,c.alive)) ;
}

```

During normal operation (i.e. when there are no core stack view changes), the coordinator only has to react to messages from the members of the data group, which are limited to join requests and fail/synced notifications. This is done in the `coord_recv` function. When a member leaves or fails, the coordinator broadcasts to the members a Sync message. When all members have either synchronized or failed, then it initiates a new view.

```

(* COORD_RECV: Handler for coordinator receiving a
 * membership message from a member.
 *)

```

```

function coord_recv c (endpt,msg) =
  (* Synchronize, if haven't done so already.
   *)
  if (not c.syncing) then {
    c.syncing := true ;
    c.broadcast Sync
  } ;

  (* Process the message.
   *)
  match msg with
  | Join ->
    (* Add the member to the group.
     *)
    c.alive := insert endpt c.alive ;
    c.syncd := insert endpt c.syncd ;
  | Synced ->
    (* Mark the member as being synchronized.
     *)
    c.syncd := insert endpt c.syncd ;

  | Fail(failure) ->
    (* Remove the member from the group.
     *)
    c.alive := except failure c.alive ;
    c.syncd := except failure c.syncd ;
    c.broadcast (Failed failure) ;

  (* Check if the view is now ready.
   *)
  coord_check_view c

```

5.2.3. *Members*

For each member of a data group, there is one member object in the corresponding Maestro process. This object contains state for managing the client data stack. Each member is in one

of three states: normal (no membership changes in progress); synchronizing (preparing for a view change); and synchronized (ready for a view change). In addition, it keeps track of the last view it received from the coordinator and the logical time of the view. The logical time along with the identifier of the core group uniquely identify the view of the subgroup.

```

type state = Normal | Syncing | Syncd

(* MEMBER: Record containing the state of a member. Each
 * client has one such record.
 *)

type member = {
  endpt : endpt ;                               (* my endpoint *)
  state : stage ;                               (* my state *)
  ltime : int ;                                 (* my logical time *)
  view : endpt list ;                          (* view of group *)
}

```

Members communicate both with their client data stack and with the coordinator of the subgroup. The client data stack generates messages when it joins or leaves the group and when it becomes synchronized. The coordinator generates messages to synchronize for a view change, to notify other members that client stacks have failed (or left), and to install new views. `member_recv_client` handles messages from the client, which consist mostly of passing the message on to the server.

```

(* MEMBER_RECV_CLIENT: Handler for a member to receive a
 * message from its client. It just passes the message on
 * to the coordinator and updates the member's stage.
 *)

```

```

function member_recv_client m msg =
  m.send_to_coord msg ;
  match msg with
  | Join ->
    m.state := Syncd
  | Synced ->
    m.state := Syncd
  | Fail(endpt) ->
    m.disable ()

```

Communication with the coordinator is handled in `member_recv_coord` and is somewhat more complicated. Again, the member usually just passes messages from the coordinator on to the client. However, it must also handle some special cases for failures and view changes. If a member receives an indication from the coordinator that it has failed, then this member must disable itself, since it is no longer in the subgroup. Also, when the coordinator sends out a new view, a member must check that it is in the view before accepting it. This verification is required since during periods of frequent membership changes, it is possible that a new member that already sent a `Join` message receives a `View` message that was sent before its join request reached the coordinator. In such cases, the joining member is

logically still not part of the subgroup, and must wait for a later View message that includes it.

```
(* MEMBER_RECV_COORD: Handler for a member to receive a message
 * from coordinator. The member checks the data, and passes
 * it on to its client.
 *)
```

```
function member_recv_coord m msg =
  match msg with
  | Sync ->
    (* If in Normal state, then start synchronizing.
    *)
    if (m.stage = Normal) then {
      m.state := Syncing ;
      m.send_to_client msg
    }

  | Failed(endpt) ->
    (* If the member being failed is in my view,
    * then pass the message to the client.
    *)
    if (is_element_of endpt m.view) then
      m.send_to_client msg ;

    (* If I'm the one being failed, then disable me.
    *)
    if (m.endpt = endpt) then
      m.disable ()

  | View(ltime,view) ->
    (* If I'm listed in the view, then install the
    * information.
    *)
    if (is_element_of m.endpt view) then{
      m.state := Normal ;
      m.ltime := ltime ;
      m.view := view ;
      m.send_to_client msg ;
    }
```

Core-group view changes

Perhaps the most complex part of the protocol occurs when view changes occur in the core stack. This is difficult because processes in the core group may fail, and multiple partitions may merge together. Failed members need to be removed from the subgroup views and partitioned views need to be merged together. In addition, the protocol must determine if

the subgroup is affected by the core stack view change: if not, then the core-group's view change should not affect the subgroup.

When a core stack view change occurs, the old state for the coordinator is thrown out and a new state is constructed based on the members. The states of the members are used to reconstruct the state of the coordinator. The Ensemble protocols automate this state transfer and Maestro only has to provide a function, `coord_reconstruct` that takes all of the states of the members and initializes the new coordinator state. The coordinator's list of live members is the endpoints of the members that transferred state; the logical time is the maximum logical time of all the members; the list of synchronized members are those members that are in the `Syncd` state; and the group is synchronizing if any of these hold (1) if any of the members are synchronizing, (2) any member disagrees about the group's view, or (3) any member disagrees about the group's logical time. If the group is synchronizing, the coordinator then broadcasts a `Sync` message. Next the coordinator broadcasts `Failed` messages for any endpoints that were in the members view but are not in the list of live members. Finally, the coordinator checks if a new view is ready and sends it out if it is.

```
(* RECONSTRUCT: Given lists of the fields of the members,
 * reinitialize a coordinator, and send out messages to
 * update members on the state of the group.
 *)

function coord_reconstruct c (endpts,states,ltimes,views) =
  (* Reconstruct fields as follows:
   *
   * alive: sorted list of endpoints
   * ltime: maximum of ltimes
   * syncd: all members in Syncd state
   * syncing: if any members is not in normal state
   * or any member disagrees about the view.
   *)
  c.alive := sort endpts ;
  c.ltime := list_max ltimes ;
  c.syncd := list_filter2 (state = Syncd) states endpts ;
  c.syncing :=
    exists (state <> Normal) states or
    exists (view <> c.alive) views or
    exists (ltime <> c.ltime) ltimes ;

  (* Send out Sync if necessary.
   *)
  if (c.syncing) then
    c.broadcast Sync ;

  (* Take the union of all views.
   *)
  total := list_union views ;

  (* Any of members not alive are failed.
```

```

    *)
failed := subtract total c.alive ;
foreach endpt in failed {
    c.broadcast (Failed endpt)
} ;

(* Check if view is ready.
*)
coord_check_view c

```

5.3. Maestro performance

There are three cases to consider in analyzing the performance of Maestro. They are listed in decreasing order of how much they affect performance.

1. **Normal case:** Maestro introduces little or no costs in the normal case (when no membership changes occur), because subgroup members normally communicate directly with each other and only use Maestro for membership changes. The only cost is some occasional background communication that is required to detect failures, for instance. In fact, applications that do not use Maestro would have to carry this background communication anyway. When Maestro is used, all of this communication is amortized across all of the subgroups served by Maestro, thus potentially saving communication resources.
2. **Sub-group view changes:** the next situation to consider is the cost of a subgroup view change (when the core-group view does not change). The costs for this are different for synchronized and unsynchronized subgroups. For unsynchronized subgroups, the cost is one broadcast from the coordinator to install the new view. Synchronized subgroups must synchronize first and this adds an additional broadcast and N (where N is the size of the subgroup) point-to-point replies.
3. **Core-group view changes:** the final situation involves a view change in the core-group. These view changes only occur when a process joins or leaves (possibly through failing) the core-group. This is the abnormal case because the core-group membership is typically quite static. In many cases, core-group view changes do not affect communication in the subgroup. For instance, when a new process joins the core-group, this does not affect any subgroups because the new process does not include any subgroup members, nor does it manage any such members. When a core-group process fails, the only subgroups that are affected are those with one or more members that were managed by that process. In these cases, the new subgroup's view is computed after the core-group's view change has been completed, and is basically done by projecting the core-group's view onto the subgroup's view, which can be done locally (fast).

5.3.1. Measured performance

Recall that Maestro stays out of the critical data path for normal message. Thus, the latency and throughput for data stacks using Maestro is the same as if they would have been used without Maestro. Hence, when using Maestro, the interesting performance data is the latency of the protocol that performs view changes, since this is also the latency to join groups, and this is the time interval in which virtually synchronous stacks are prohibited from sending

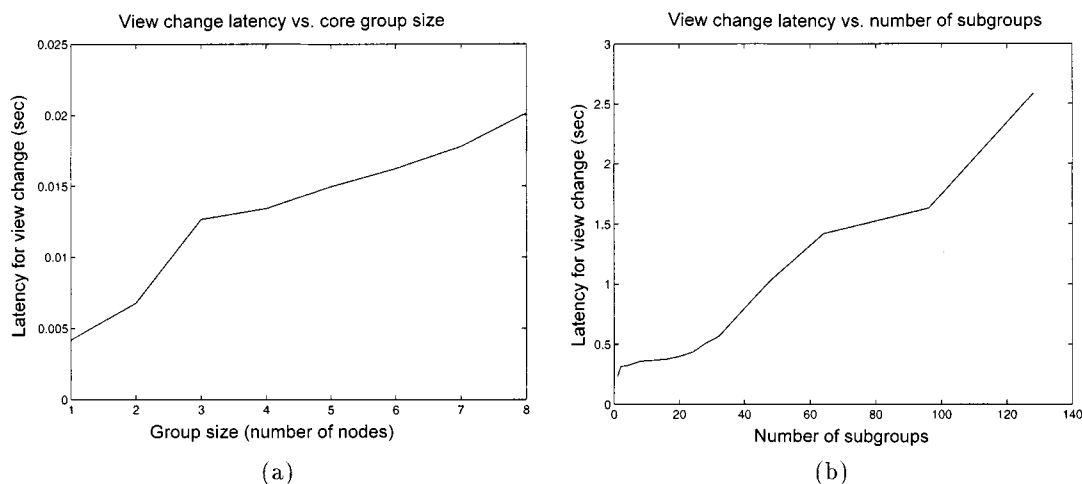


Figure 5. Latency of the Maestro view change protocol. (a) shows the latency vs. the size of the core group (with one subgroup), and (b) shows the latency vs. the number of subgroups running in separate processes. Process switch overhead causes in the latency in (b) to be worse than that of (a) for the 8-member, 1-group case.

messages. It is reasonable to assume that video and audio channels would not use virtually synchronous stacks, and therefore would not suffer from hiccups during view changes.

All performance measurements were taken on an otherwise idle 8-node IBM SP2. Communication between nodes was performed using standard point-to-point UDP communication over an Ethernet segment. That is, we *do not* use the SP2 fast interconnect, and we did not use IP-multicast because the SP2 does not support it, though Ensemble supports both (measurements taken using the SP2 fast interconnect show a moderate improvement over Ethernet, but this medium is not very representative of the typical environment for Maestro).

Performance for one data-stack: the first set of measurements shows the performance for Maestro view changes with one data stack as a function of the number of nodes in the system. We use between 1 and 8 processes, one on each node of an 8-node IBM SP2. Each process has a control stack and one data stack, and we measure how quickly the data stack can perform null view changes (initiated by an empty failure notification from one member), so the control group does not go through view changes. In addition the data stacks respond to synchronization message immediately, so what is being measured is the overhead introduced by Maestro for the view change. The measurements range from 4 ms for one member (where no actual message traffic occurs) to 19 ms for the eight member case, as can be seen in Figure 5(a).

Managing several groups: the second set of measurements shows a 'worst-case' scenario for sub-group view changes in Maestro. We use eight servers and eight client application processes, one per node. Each client connects to the local server via a local TCP connection and joins a number of groups which varies across the tests. Hence this experiment indicates the performance that external communication transports would experience when using Maestro via a TCP connection. When eight members have joined each of the groups, one of the members starts a view change (by sending an empty failure notification), this causes the group to synchronize; all of the members respond immediately to the synchronization. When the new view is ready, a member requests another view change, and the group continues in

this manner. The number of groups each clients joins varies from 1 to 128. We measure the average latency of the view changes for each group. The total number of view changes per second supported by the system for this case can be calculated as $ngroups(1/latency)$, and ranges from around four views per second (for 1 group) to around 50 (for 128 groups). This shows the ability of the system to manage large numbers of subgroups, and the effect of aggregating membership messages in Maestro on scaling the number of groups.

Note that this is a worst case scenario in the sense that all groups (up to 256) are continuously changing views at once, which is an abnormal behavior for many applications; group membership is usually relatively static after initialization. In most cases, the synchronization process for each member is not immediate, and so one can expect less strain on the system in normal operation. Although this case is somewhat pessimistic, the results, as indicated in Figure 5(b), are very promising.

As can be seen from the measurements reported above, Maestro's performance is quite adequate for most applications. Nonetheless, we are continuing to work on improving the performance by experimenting with several alternative protocols. (Although, as we mentioned before, Maestro's performance is outside of the code path for common case communication, so Maestro's performance does not affect normal communication.)

5.4. Correctness

No Spontaneous View Changes is clearly satisfied by the code, since join requests and failure notifications are the only events that can trigger a view change. Similarly, since whenever the coordinator receives a `Join` message from an endpoint p , it adds p to the list of living processes in the next view, and since only a failure notification can eliminate an endpoint from that list, then non triviality of views is satisfied.

Also, since each `View` message contains only one membership list which is sent to all members, and since only the coordinator sends this message, it is clear that there is a unique coordinator and a unique membership list associated with each view. By the code, there can be at most one coordinator of a subgroup within a given view of the core group, and the value of `ltime` is incremented by this coordinator for each view of the subgroup. Since the core group is virtually synchronous, its view identifiers are unique, implying that the view identifier of each subgroup is also unique. Hence, our protocol guarantees view uniqueness.

To prove that the protocol provides views consistency, we must show that all members eventually see the same set of views in the same order. As long as there are no changes in the core group, this is guaranteed by the FIFO reliable delivery property of the core stack, which carries all membership related messages. Whenever there is a change in the core stack that also affects the subgroup, the view change of the subgroup is performed only after the new view of the core group has been established. Due to the fact that the core stack is virtually synchronous, the `View` messages of newer views are received only after all `View` messages that were sent before the change in the core group are delivered.

Recall that by the code of our protocol, whenever a view change occurs in a virtually synchronous subgroup, members of that subgroup do not reply with a `SyncOK` message until all messages sent in the previous view are stable. Similarly, in these subgroups the application is not supposed to send messages between receiving a `Sync` message and receiving the following `View` message. (If the application does send messages, they are buffered until the next view is installed.) Thus, Agreement on Messages Between Views is also satisfied.

6. RELATED WORK

The idea of sending video over several concurrent streams, where each stream corresponds to higher quality, has been used earlier [2–4]. These works use the grouping mechanism of IP-Multicast for managing the subgroups corresponding to the data streams employed by it. Our work can be seen as complementing that approach in two ways: First, we provide a general framework that can be used with or without IP-Multicast. Hence, by using our system, an application can be ported from settings where IP-Multicast is not supported to ones in which it is supported or vice versa, without changing the code. Moreover, it allows applications to run in a mixed environment, in which some of the nodes are connected through multicast routers and some nodes are not. (For the latter nodes our system simulates multicasts.) Secondly, our system adds functionality beyond IP-Multicast groups since we provide stronger semantics for control messages, allowing synchronization where needed, and can utilize Ensemble's security architecture to make the entire application more secure. Note that these semantics are provided in addition to what IP-multicast provides so that programmers can choose appropriate semantics and services that match the applications they develop. On the other hand, IP-Multicast scales better than our approach, which is limited to several hundreds of processes. Hence, our system is more suitable for CSCW and medium scale multimedia applications.

The Real-time Transport Protocol (RTP) [40,41] provides network transport functions suitable for transmission of real-time data, such as audio, video or simulation data, over multicast or unicast networks. RTP encourages use of multiple data streams to support different types of input data, and provides support for handling multiple data streams. A companion protocol called RTCP [40,41] allows monitoring of data delivery, and supplies participant information to the application layer. RTP/RTCP is not designed for reliable data transmission. Note that the purpose of Maestro is not to replace any existing protocols, such as RTP or TCP, but to complement them by providing a convenient platform where a group-oriented data stack can be implemented using such protocols. For instance, Maestro applications can use RTP as one of data stacks while using the core stack of Maestro or another reliable data stack to transmit some data reliably. Since collaborative applications need a variety of protocols, Maestro can be used to provide group management services to these protocols.

The interface to Maestro is an extension of the CCTL interface discussed by Rhee *et al.* [10]. The CCTL interface also supports multiple groups with various stacks. However, it does not support automatic joining to groups, which as we saw earlier in this paper can be useful for many applications. Also, the work of Rhee *et al.* [10] mainly addresses collaborative computing applications (CSCW), while as we have already explained [42], we explicitly support other kinds of applications, including cluster management, prioritized delivery and multilevel security.

Recently, Chockler *et al.* [9] proposed the Multimedia Multicast Transport Service for Groupware (MMTS) architecture for supporting multiple quality of service for multimedia application using group communication. While MMTS is similar to Maestro in providing support for multiple data stacks, it lacks provisions for having different membership in each subgroup. Moreover, in the model proposed by Chockler *et al.* [9] all messages must belong to a particular *bunch*, and must be delivered, regardless of the data stack they were sent on, between the *begin-bunch* and *end-bunch* operations. We believe that enforcing this on all stacks would be unnecessarily restrictive in the applications of interest to us. Indeed, as noted in the earlier discussion of an audio data stream, we believe that there are situations which such synchronization would actually reduce the perceived reliability of a system, not

increase it. In our model, an unreliable stack may synchronize its message delivery with view changes in the core stack, but does not have to do so if the application does not require such strong delivery semantics.

7. DISCUSSION

Maestro is a scalable and efficient multiple-group management tool. It provides membership services for subgroups that subscribe to it, and stays out of the critical data path for the subgroups it manages. Subgroup stacks can be part of the same process in which Maestro runs, or can reside within external processes that communicate with Maestro via a TCP connection (recall that only membership notifications and periodic events pass through such a connection). This architecture allows provision of different quality of service guarantees to different subsets of the same application, or a set of related applications, while guaranteeing consistent membership changes for all subsets. As such, it matches the communication needs of many multimedia and collaborative computing applications, and can be used as a building block, as we have demonstrated using the IMUX pseudo X-server.

Our experience with Maestro was generally good. This is true w.r.t. both writing applications that utilize Maestro, as been discussed before, but also in terms of the decision to build Maestro on top of an existing group communication rather than writing it from scratch. The latter decision greatly simplified our code and algorithms, and fits well with the layered approach of Ensemble, in which complex functionality is gained by adding simple layers on top of existing ones (even though Maestro is not an Ensemble layer *per se*).

One area that is not addressed by Maestro, is real time. That is, neither our specification nor our protocols or Ensemble guarantee timed response to membership events. (On the other hand, Maestro can support external real-time communication stacks in which data messages are sent with real time guarantees, but the membership is not.) Thus, Maestro as is is not suitable for most real time applications. It would be interesting to develop a timed version of Maestro.

Looking into the future, we would like to explore the issue of specifying more elaborate properties, and relations between properties, and incorporate this into Maestro. Also, it would be interesting to look at other application areas, such as multilevel security, and see if they can be answered using a similar framework.

In summary, in this paper we have discussed the concept behind Maestro, showed its applicability with specific examples, provided a detailed description of Maestro's design and implementation, and discussed its performance. Maestro is fully implemented, and can be retrieved from <http://www.cs.cornell.edu/Projects/Ensemble>.

REFERENCES

1. B. C. Smith, 'Implementation techniques for continuous media systems and applications', *PhD thesis*, Department of Computer Science, University of California at Berkeley, 1994.
2. S. Y. Cheung, M. H. Ammar and X. Li, 'On the use of destination set grouping to improve fairness in multicast video distribution', *Proc. of the 15th International Conference on Computer Communication (INFOCOM)*, March 1996, pp. 553–560.
3. L. Mathy and O. Bonaventure, 'QoS negotiation for multicast communications', *International COST 237 Workshop, LNCS 882*, Springer-Verlag, November 1994, pp. 199–218.
4. S. McCanne and V. Jacobson, 'Vic: A flexible framework for packet video', *Proc. of ACM Multimedia*, November 1995, pp. 511–522.
5. T. Rodden, 'A survey of CSCW systems', *Interacting with Computers*, **3**(3), 319–353 (1991).

6. H. Abdel-Wahab and M. Feit, 'A framework for sharing X Window clients in remote synchronous collaboration', *Proc. of the IEEE Conference on Communication Software*, 1991, pp. 159–167.
7. K. Lantz, 'An experiment in integrated multimedia conferencing', *Computer-Supported Cooperative Work: A Book of Readings*, 1988.
8. A. Krantz, I. Rhee, C. Breuker, S. Chodrow and V. Sunderam, 'Supporting input multiplexing in a heterogeneous environment', *Proc. of the Third Joint International Conference on Information Science*, March 1997, pp. 97–100.
9. G. Chockler, N. Huleihel, I. Keidar and D. Dolev, 'Multimedia multicast transport service for groupware', *Proc. of the TINA 96 Conference*, September 1996, pp. 43–54.
10. I. Rhee, S. Cheung, P. Hutto and V. Sunderam, 'Group communication support for distributed multimedia and CSCW systems', *Proc. of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997, pp. 43–50.
11. F. P. Brooks, *The Mythical Man-Month*, Addison Wesley, October 1995. (20th Anniversary Edition.)
12. D. M. Fetterman, 'Videoconferencing on-line: enhancing communication over the internet', <http://cu-seeme.cornell.edu/Fetterman.html>
13. The Ensemble Home Page. <http://www.cs.cornell.edu/Info/Projects/Ensemble/>
14. M. Hayden, 'The ensemble system', *Technical Report TR98-1662*, Department of Computer Science, Cornell University, January 1998.
15. K. P. Birman, *Building Secure and Reliable Network Applications*, Manning Publishing/Prentice Hall, December 1996.
16. K. Birman and T. Joseph, 'Exploiting virtual synchrony in distributed systems', *Proc. of the 11th ACM Symposium on Operating Systems Principles*, December 1987, pp. 123–138.
17. A. Basu, V. Buch, W. Vogels and T. von Eiken, 'U-Net: A user-level network interface for parallel and distributed computing', *Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1996, pp. 40–53.
18. W. R. Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, October 1995.
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).
20. W. Vogels, 'World wide failures', *ACM SIGOPS European Workshop*, 1996.
21. T. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost, 'On the impossibility of group membership', *Proc. of the 15th ACM Symposium of Principles of Distributed Computing*, May 1996, pp. 322–330.
22. F. Cristian and F. Schmuck, 'Agreeing on processor-group membership in asynchronous distributed systems', *Technical Report CSE95-428*, Department of Computer Science, University of California, San Diego, 1995.
23. R. Friedman and R. van Renesse, 'Strong and weak virtual synchrony in Horus', *Proc. of the 15th Symposium on Reliable Distributed Systems*, October 1996, pp. 140–149.
24. L. E. Moser, Y. Amir, P. M. Melliar-Smith and D. A. Agarwal, 'Extended virtual synchrony', *Proc. of the 14th International Conference on Distributed Computing Systems*, June 1994.
25. G. Neiger, 'A new look at membership services', *Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, 1996.
26. A. Ricciardi and K. P. Birman, 'Consistent process membership in asynchronous environments', *Reliable Distributed Computing with the ISIS Toolkit*, IEEE Press, Los Alamitos, 1993.
27. 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.)
28. K. Birman, 'The process group approach to reliable distributed computing', *Communications of the ACM*, **36**(12), 37–53 (December 1993).
29. R. Friedman and R. van Renesse, 'Packing messages as a tool for boosting the performance of total ordering protocols', *Proc. of the Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997.
30. C. Malloth, P. Felber, A. Schiper and U. Wilhelm, 'Phoenix: A toolkit for building fault-tolerant distributed application in large scale', *Technical Report*, Department d'Informatique, Ecole Polytechnique Federale de Lausanne, July 1995.
31. Ö. Babaoğlu, R. Davoli, L. Giachini and P. Sabattini, 'The inherent cost of strong-partial view-synchronous communication', *Technical Report UBLCS-95-11*, Department of Computer Science, University of Bologna, April 1995.
32. 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.

33. S. McCanne and V. Jacobson, 'Vic: A flexible framework for packet video', *Proceedings of ACM Multimedia*, November 1995, pp. 511–522.
34. S. McCanne, M. Verrerli and V. Jacobson, 'Low-complexity video coding for receiver-driven layered multicast', *IEEE Journal on Selected Areas in Communications*, **16**(6), 983–1001 (August 1997).
35. S. Y. Cheung, M. H. Ammar and X. Li, 'On the use of destination set grouping to improve fairness in multicast video distribution', *Proc. of 15th International Conference on Computer Communication (INFOCOM'96)*, March 1996.
36. B. Glade, K. Birman, R. Cooper and R. van Renesse, 'Light-weight process groups in the ISIS system', *Distributed System Engineering*, **1**, 29–36 (1993).
37. V. S. Sunderam, M. Hirsch, S. Cheung, S. Chodrow, M. Grigni, A. Krantz, I. Rhee, P. Gray, S. Oleson, P. Hutto and J. Sult, 'CCF: Collaborative Computing Framework', *Proceedings of SuperComputing*, November 1998.
38. G. Chung and P. Dewan, 'A mechanism for supporting client migration in a shared window system', *Proc. of the Ninth User Interface Software and Technology*, November 1996, pp. 11–20.
39. Y. Amir, D. Dolev, S. Kramer and D. Malki, 'Transis: A communication sub-system for high availability', *Proc. of the 22nd Annual International Symposium on Fault-Tolerant Computing*, July 1992, pp. 76–84.
40. *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889.
41. *RTP Profile for Audio and Video Conferences with Minimal Control*. RFC 1889.
42. K. Birman, R. Friedman and M. Hayden, 'The Maestro group manager: A structuring tool for applications with multiple quality of service requirements', *Technical Report TR96-1619*, Department of Computer Science, Cornell University, March 1996.