The Maestro Group Manager: A Structuring Tool For Applications With Multiple Quality of Service Requirements*

Ken Birman Roy Friedman Mark Hayden
Dept. of Computer Science
Cornell University
Ithaca, NY 14850, USA
{ken,roy,hayden}@cs.cornell.edu

February 4, 1997

Abstract

Maestro is a tool for managing sets of protocol stacks that satisfy varied quality of service or security requirements. Intended primarily for multimedia groupware settings, it permits a single application to efficiently operate over multiple side-by-side protocol stacks, each specialized to a different communication stream. Maestro can also be used to manage other sorts of external protocol stacks, for example to orchestrate connection setups that require coordinated actions at all endpoints in a multicast group. Our tools are fault-tolerant and secure; they can safely distribute session keys or handle delicate synchronization tasks that would otherwise complicate the managed stacks and potentially interfere with their quality-of-service objectives. Moreover, Maestro can automatically track subgroup membership on the basis of "properties", facilitating its use by developers who prefer not to work directly with multicast communication interfaces.

*This work was supported by ARPA/ONR grant N00014-96-I-1014.

1 Introduction

We are interested in the development of multicast applications that operate over protocol stacks offering quality of service or security guarantees. Such applications often need multiple quality of service stacks matched to the needs of subsystems operating side-by-side within a set of processes. Communication toolkits that support flexible protocol configuration are increasingly common: they include streams, XTP, the COM and DCOM services within NT, and our two multicast systems, Horus and Ensemble [25].

A challenge in applications that use configurable protocol stacks is to set up a set of communication endpoints in a consistent manner (for example, with matching quality of service expectations), and to automatically create connections or tear them down as the multicast group membership changes. Doing so may not be trivial: the endpoint processes need mutually consistent membership information and quality-of-service expectations, and may need access to shared session keys or other data. The control and management tools will often need to tolerate failures and to automatically initiate reconfiguration of the managed stacks when an application joins the session, leaves it, or fails. These are hard problems, and solving them usually require complex protocols.

The issues become even more difficult when a single application uses multiple communication stacks simultaneously. In these cases, there is a need for consistency not just with regard to endpoints in a single multicast group bound to a single stack, but also across the set of groups, each with its own stack and endpoints. This leads to an architecture in which some form of core protocol management group controls a set of managed substacks. For such purposes, we believe that *virtually synchronous group communication tools* [6] find a natural fit. Our objective in this paper is to show how such tools can be used to control and manage a set of protocol stacks with quality of service objectives of their own, in a manner that is minimally intrusive (our users will often prefer not to work directly with multicast communication interfaces), survivable, and most important of all, inexpensive.

The work reported here was done using Cornell's Ensemble groupware system. Prior to this effort, Ensemble lacked any mechanism for guaranteeing consistent membership across sets of groups. That is, a process may be detected as faulty in a group A, and consequently removed from the view of A, but remain a member of other groups whose membership overlaps with that of A. This might mean that one subsystem, using A, reconfigures itself to exclude the process in question, while others continue to attempt to communicate with it for some period of time. In the worst case, the detected failure might be a transient one, in which case those other groups would continue to carry the excluded process indefinitely, despite the fact that it has been dropped from group A. Moreover, each protocol stack is forced to independently perform failure detection, and to implement an independent retransmission protocol (in case of reliable stacks). Jointly, these considerations introduce inefficiency and inconsistency, and can saturate the message receive-queues with redundant messages [24].

Here, we describe *Maestro*, a tool for managing multiple process groups, each communicating over its own stack. These stacks may be other Ensemble multicast stacks, or may be external to Ensemble, as just discussed. When using Maestro, a single management group, known as the *core group*, is configured to include all participating processes within the application.[1] This core group uses a virtually synchronous protocol stack, 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. However, the creation of such subgroups is "announced" by multicasts within the core group (and core stack), as are join requests

[1] Later in this paper we also discuss how several applications can share a single management group.

to subgroups, leave requests, and subsequent membership changes for data stacks.

In our approach, membership of a subgroup must be a subset of the 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 main 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 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 [26]).

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. The automatic joining facility and the aggregated handling of registered events in Maestro extends the interface provided by CCTL [22], as discussed in Section 6.

Finally, 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. In this paper we also discuss how this approach helps in cluster management, providing prioritized delivery for real-time applications, managing external communication stacks such as TCP and user level communication interfaces (e.g., U-Net [4]) in a secure way, and providing multi-level security.

The rest of this paper is organized as follows: Group communication systems and virtual synchrony are briefly described in the next section. A discussion of how applications can use Maestro appears in Section 3. The interface to Maestro is presented in Section 4, while the the implementation of Maestro using the Ensemble group communication system is described in Section 5. The paper concludes with a comparison of our work to prior approaches.

2 Group Communication Systems and Virtual Synchrony

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 multicasting 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

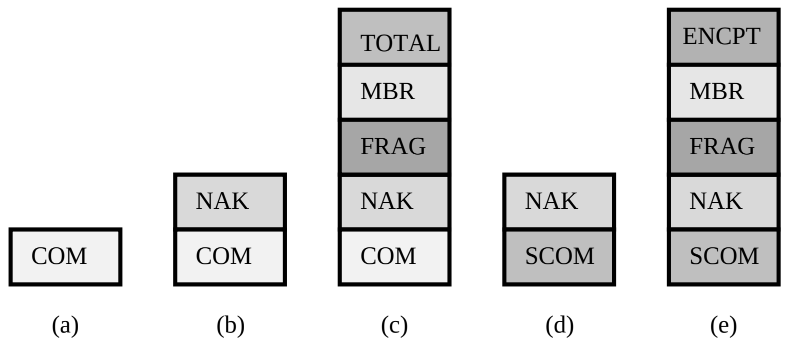


Figure 1: *One application might operate over a set of side-by-side protocol stacks, each specialized to a different need. The stacks above provide: (a) unreliable unordered communication, (b) provides FIFO reliable delivery, (c) provides totally ordered virtually synchronous communication, (d) provides 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.*

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 in [6].

Our own work was based on the Ensemble system, which supports a communication model called *virtual synchrony* [5, 6]. The details of the model will not be important in this paper, except insofor 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. And 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 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.

3 Applications

Maestro's architecture is motivated by the structure of applications from a variety of domains. Here we describe several of these domains and the ways in which Maestro helps in managing their communication needs.

3.1 Collaborative Computing and Multimedia

Many collaborative computing and multimedia applications require support for multiple levels of quality of service. Such a need matches naturally with the use of multiple stacks. Consider, for example, a video conference session, in which there are several participants, connected by a WAN, although there may be subsets of the participants which are located in the same LAN. These participants may want to transmit both audio and video, and perhaps also text from a joint chat area and drawings from a distributed white board.

These types of data require different forms of quality of service. For example, 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 acknowledgment mechanism might introduce undesired latency variations 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. For video, we would probably want to use a coding scheme similar to the one used by *vic* [16]. Here, one stack is used for X-frames, which should be delivered to all participants, while another is used for Y frames, and delivered only to participants running on nodes that support high bandwidth links to the video sender. 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.

It is easy to imagine variations upon this theme. For example, much of our work is concerned with security. By supporting subgroups with varied security "clearances" such a system might support briefings in which different participants have access to different levels of material, as a function of security clearance.

A CSCW system of specific interest to us is CCTL [22], which is structured with multiple subgroups, each having its own stack. This structure simplifies the design of CCTL applications. The ability to automatically join groups can further simplify the development of such applications, since the application does not need to actively join groups like the general audio, low quality video, and the local high quality video subgroups. Using Maestro's automatic membership management features, the developer does not need to understand details of our system to exploit subgroups, and hence should find the tool both natural and also easy to use.

3.2 Cluster Management

Cluster management in implementations of reliable distributed servers is another area where Maestro can be very helpful. An example of a typical architecture for such a cluster was given by Friedman and Birman [11], and is depicted in Figure 2.[2] In this case, there are several (two) *external adaptors*, or EAs for short, to which clients of the system connect and send their requests. The EAs are responsible for forwarding requests for the *compute nodes* inside the cluster, for collecting

[2] Microsoft's Wolfpack and the Stratus Radio are two examples of commercial cluster architectures closely matched to our assumptions; we are now porting Maestro to NT and plan to explore its use in such systems later in 1997.

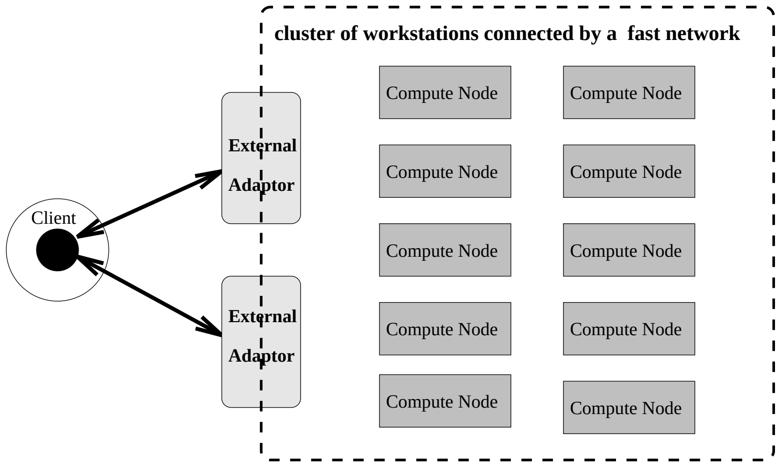


Figure 2: *A cluster based implementation of a reliable server*

the replies from the compute nodes and forward them back to the client, and for reissuing requests inside the cluster in case of a failure. Here, the group communication system helps in managing the cluster, as it provides failure detection and automatic reconfiguration when there is a failure or when a new/recovered node is added to the system. The group communication system also provides hooks for state transfer which is required to bring new nodes up to date with the current state of the system.

In order to provide fault tolerance, the compute node that handles a given client request will typically send its reply to all EAs. Hence, having multiple subgroups, each including one compute node and all EAs can simplify the application and save time, since it allows the replies to be sent to the EAs in one downcall, using hardware multicast capabilities where available. Additionally, the EAs may form a subgroup only for themselves, for use in sending distributed state updates. The automatic membership features of Maestro are handy in this setting, because the subgroups are easily described by property sets. For example, each compute node can create a group and specify that the EAs have to join it. The EAs will automatically be joined to this group, without requiring any explicit application-level code to carry out the action.

One finding in this work is that for maximum scalability and rapid response times, it is necessarily to develop multicast protocols closely matched to the architecture. Maestro, by offering a simple way to manage the corresponding protocol stacks, greatly simplifies the implementation of these sorts of special-purpose protocols.

3.3 Prioritized Delivery and Real-time

Supporting priorities inside a single protocol layer, or a multi-layer protocol stack, is often a hard task, as it involves writing code to permit higher priority messages to bypass lower-priority ones. Many existing group communication systems lack support for such a feature, yet the developers of real-time applications often need it. In developing Horus and Ensemble, many layers were written based on the assumption that messages arrive in FIFO order, and modifying them to support prioritized scheduling would be a complex task. Of course, one could claim that these layers could have been written in a way that supports priorities in the first place. However, writing such layers is often more difficult than writing layers that can assume FIFO message deliveries, so the system would have become more complex and error prone. And, as a result of this added complexity, the formal verification of Ensemble would become much more difficult.

By using several stacks, supporting prioritized delivery becomes straight-forward. Maestro allows members joining a subgroup to specify a *priority*. If all stacks are managed by the same scheduler, as in the implementation in Ensemble described in Section 5, the scheduler can schedule events according to the priority of the stacks in which they arise. Thus, existing layers can be used unchanged, without added complexity, while the system as a whole now provides prioritized delivery.

In Ensemble, the scheduler maintains several FIFO queues, one for each priority. When an event is generated, it is inserted at the end of the queue corresponding to the priority of the stack on which it was generated. Events are then scheduled from the head of the first non-empty queue, i.e., the one with the highest priority that is not empty.

3.4 Management of External Stacks

Maestro has potential applications in settings where more conventional protocol stacks are in use. Consider an application that needs to maintain a stream connection between certain members of a multicast group. By constructing a small stub, Maestro can be used to mediate in the establishment and tearing down of external connections running over standard protocol stacks. In this role, Maestro benefits from the fault-tolerance, consistency and security properties of Ensemble, enabling it (for example) to configure an application for the same quality-of-service expectations at multiple sites in a network, or to replicate a session key in a secure and safe manner for use in secure channel setup. The application itself will then run over the normal protocol stack, benefiting from Maestro's management functions without actually communicating through Maestro itself.

As noted earlier, there may be many reasons why one would want to use Maestro in this manner. Applications that can run with selectable quality of service need a way to coordinate the configuration of communication endpoints, so that the processes at each end have consistent expectations. They need to reconfigure in a consistent manner after failures or when new processes join, or if communication conditions change. And there may be a need for consistent data, such as session keys. Maestro can assist in solving all of these problems.

Also, in recent years, several high-performance user level communication stacks, such as U-Net [4] and Active Messages have been developed. Such interfaces typically incur user-to-user latencies of 30-40 microseconds, and can saturate ATM links with fairly small messages. This performance is achieved by keeping the system kernel out of the critical path of normal messages, and employing highly efficient memory management policies. The main problem with these interfaces is that they require static configuration of the system, and cannot recover from a failure of a single node. This deficiency can be solved by using Maestro as a configuration management tool. That is, normal data messages can be transmitted directly through the fast interface. However, whenever a node fails, or a new node has to be added into the system, Maestro will detect the change, and reconfigure the system for the fast interface in a consistent way. In particular, in most cases adding or dropping a node simply means adding (deleting) some buffer space and/or start (stop) waiting for acknowledgements from a node that joined (failed). Hence, combining these high-performance interfaces with Maestro would make them usable in fault-tolerant high-performance clusters that require their performance, but at the same time must tolerate failures and must allow new nodes to be added on the fly.

3.5 Multi-Level Security

Although we have alluded to the management of session keys several times, we see a more ambitious future use of Maestro, which we are now pursuing. The general problem involves supporting multi-level security using protocol stacks that run with data encription or over a secured network connection. This is the problem of allowing someone into a system, but not allowing him/her full access to all its resources. Using multiple subgroups, we can assign different encryption keys to different subgroups, and then allow processes to access only parts of the system which are accessible through the groups to which they belong. For example, we can associate subsets of a file system with subgroups, and allow only members of a subgroup access to files associated with it. Similarly, during a video conference session, any subset of the participants may form their own subgroup with a different encryption key known only to them, which allows them to discuss some issues without the rest of the participants knowing about it.

We believe that Maestro could automate the creation and management of such multi-level subgroups, but there is additional research that will need to be done to make this a reality. Maestro will need a security model of its own; the current system can be operated in a secure manner, using a protocol layer we call PLM, but this does not extend to securing the property abstraction against malicious members, which might attack the system through knowledge of the protocols it runs. Close integration of Maestro with IPv6 security features also merits study. These topics represent an important area for future research.

4 Maestro

Maestro can execute in various configurations, including as an application library or as a stand-alone server. This flexibility comes about because the service it provides can be abstracted as an event stream between Maestro and the application. Maestro can either reside as a library in the same process as an application (as illustrated in Figure 3) or can run independently as a server, or both at the same time. In the case where Maestro is used as a library, interactions between it and the application occur directly through function calls.

With the current implementation of Maestro, it is possible to have some of the side stacks executing on a different machine, since the side interface to Maestro supports communication over a TCP connection. In this case, we view Maestro as a "server". Moreover, data stacks do not have to be Ensemble stacks. As long as a protocol stack obeys the side interface of the core stack, and lets the core stack maintain their membership, they can be coded in Horus, ISIS, CCTL, Xtp, or even as a simple interface to UDP or Cyclic UDP written in C, C++, JAVA, and so forth. This requires some caution, of course, since some properties such as prioritized delivery and multilevel security depend on features of Ensemble, and on the fact that Ensemble controls the scheduling of all stacks. If a stack is implemented outside Ensemble, such properties, if needed, would have to be implemented by the system that control the external stack. Nonetheless, with the emergence of pre-built subsystems that run over communication stacks with unusual quality of service properties, we see many reasons that this external stack mechanism could be useful.

Use of Maestro as a server does not significantly affect performance because data messages are sent directly between application data-stacks and only messages regarding membership are communicated to Maestro. Normally, the application executes on the same machine as the server, but it is not necessary for it to do so.[3] In the remainder of this section, we discuss the Maestro event interfaces. There are two classes of events: those sent between Maestro and the application, and those sent between Maestro and data stacks. We split our discussion of the Maestro interfaces accordingly. We then discuss the membership properties provided by Maestro.

[3]Note that running the Maestro server on a separate machine from the application process introduces additional failure dependencies: failures of either the server's machine or the TCP connection could cause a running application to be removed from subgroups.

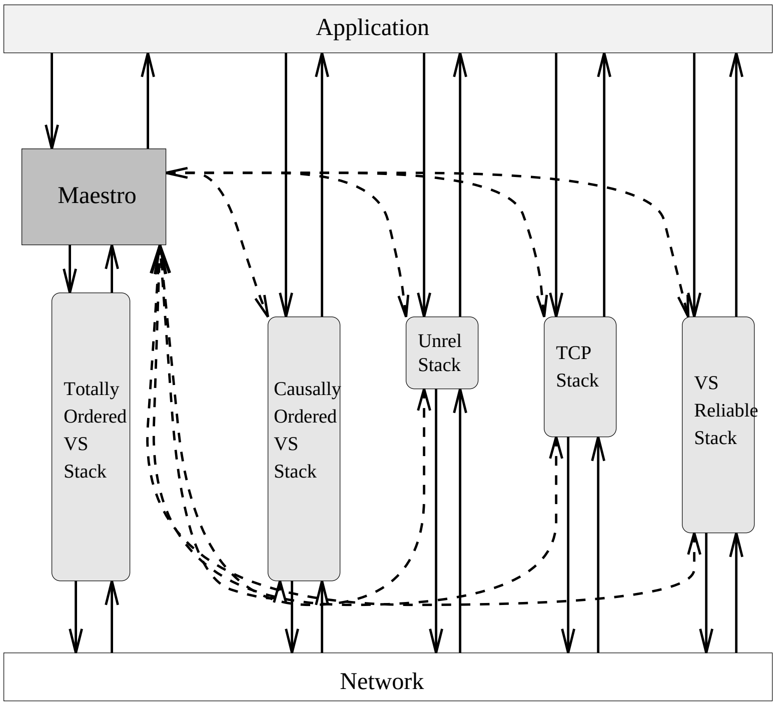


Figure 3: *Maestro system configuration.*

4.1 Application-defined "Data types"

In order 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.

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.

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.

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.

4.2 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.

4.2.1 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.

4.3 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 micro-protocols 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 it to detect failures more rapidly than Maestro. In this case the application may decide to fail members and not wait for 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

ubgroup. Applications need to be careful in failing other members, however, as a group can rapidly disintegrate if many members start to fail each other.

In order 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 acknowledgment 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.

4.3.1 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.

4.3.2 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`.

4.4 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, e.g., [3, 7, 10, 13, 19, 20, 23].) 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.

5 Implementing Maestro using Ensemble

Maestro is implemented using the Ensemble group communication system. 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. Another interesting question concerns the manner in which support for virtual synchrony provided by Ensemble on the Maestro core group facilitates the implementation of virtual synchrony provided by Maestro for the subgroups.

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. 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 participate 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,[4] and this reply is forwarded to the coordinator as a *SyncOK* message. When every member is 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. This is a useful property that greatly simplifies the Maestro protocol compared to virtually synchronous membership protocol built directly on top of asynchronous networks.

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 [12] 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 co-

[4] If the data stack is not virtually synchronous, it can reply immediately.

ordinator 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.

5.1 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.

5.1.1 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 3 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.

5.1.2 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 *)
  syncd : 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.syncd c.alive) then (
    c.syncd := [] ;
    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
```

5.1.3 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 3 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. end remove

```
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 : state ;                      (* 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 state.
 *)

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.state = 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 ;
    }
```

5.1.4 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
```

5.2 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.

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.

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.

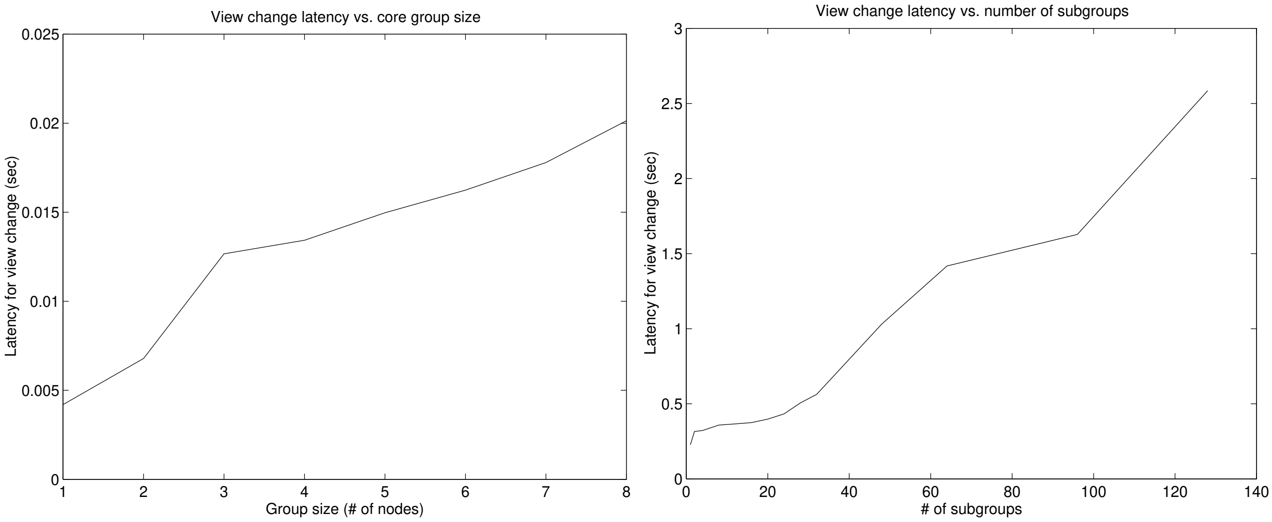
Core-group view changes: The final situation involves a view change in the core group. These view changes only occur when a process joins or leaves (possibly through failing) the core group. This is the abnormal case because the core group membership is typically quite static. In many cases, core-group 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).

5.2.1 Measured Performance

Since Maestro stays out of the data path for normal messages, 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 messages.

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 improvment 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.



(a) (b)

Figure 4: *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.*

The measurements range from $4ms$ for one member (where no actual message traffic occurs) to $19ms$ for the eight member case, as can be seen in Figure 4(a).

Managing several groups: The second set of measurements shows a "worst-case" scenario for sub-group view changes in Maestro. We use 8 servers and 8 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 8 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 4 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 4(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.)

5.3 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.

In order 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 do 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 interface to Maestro is an extension of the CCTL interface discussed in [22]. 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 in [22] mainly addresses collaborative computing applications (CSCW), while as explained is Section 3, 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 [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.

Our work is directly related to *light weight group* architectures [24] (LWG for short). In LWG, several light weight groups share the same heavy weight group, such that all communication is done through the heavy weight group, which filters and multiplexes messages to and from the light weight group. This also implies a single failure detection protocol and a single reliable delivery protocol, which reduces the amount of resources devoted by the system for these tasks and the message overhead caused by them, hence, improving the overall performance of the system. However, in LWG, since light weight group communication is routed through the heavy weight stack, all light weight group messages are forced to use identical stacks. We believe that in many multimedia and real-time applications, this would preclude the use of desired quality of service properties, because some stacks are simply incompatible with others. An LWG approach can only be employed if there exists a possible heavy weight stack with semantics acceptable to all the LWG's that map upon it. Our approach has no such restrictions.

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. Maestro is fully implemented, and can be retrieved from
http://www.cs.cornell.edu/Projects/Ensemble/index.html.

We note that, in the current implementation of Maestro, the failure of the machine on which a core group member resides would cause data stacks that are managed by this member to become disconnected from their groups. At present, such a disconnection would be disruptive, resulting in the restriction noted in an earlier footnote. However, we believe that it will be possible to prevent this problem from occurring using an appropriate fail-over protocol, so that in the event of a failure of a core group member, any of its open TCP connections will be grabbed transparently by other members of the core group. Also, we would like to support load re-distribution of side stacks (connected by TCP) among core group members. For this we need to develop hand-off protocols that can switch side stacks from one core group member to the other. Once these protocols are in place, however, it will be possible to use Maestro in mobile computing applications where such movement of stacks would correspond to movement of the application around a fixed support infrastructure.

Acknowledgements: We would like to thank Injong Rhee for sharing with us his insight regarding the needs of multimedia and CSCW applications.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proc. of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

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

[3] 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.

[4] A. Basu, V. Buch, W. Vogels, and T. von Eiken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, December 1996.

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

[6] K. P. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, December 1996.

[7] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Proc. of the 15th ACM Symposium of Principles of Distributed Computing*, pages 322–330, May 1996.

[8] C. Chang, G. Czajkowski, and T. von Eicken. Design and Performance of Active Messages on the SP-2. Technical Report TR96–1572, Department of Computer Science, Cornell University, February 1996.

[9] G. Chockler, N. Huleihel, I. Keidar, and D. Dolev. Multimedia Multicast Transport Service for Groupware. In *Proc. of the TINA 96 Conference*, pages 43–54, September 1996.

[10] 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.

[11] R. Friedman and K. Birman. Using Group Communication Technology to Develop a Reliable and Scalable Distributed IN Coprocessor. In *Proc. of the TINA 96 Conference*, pages 25–41, September 1996.

[12] R. Friedman and R. van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. Technical Report TR95-1527, Department of Computer Science, Cornell University, July 1995. Submitted for publication.

[13] 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.

[14] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

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

[16] S. McCanne and V. Jacobson. Vic: A Flexible Framework for Packet Video. In *Proc. of ACM Multimedia*, pages 511–522, November 1995.

[17] S. Mishra, L. Peterson, and R. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal*, 1(2):87–103, December 1993.

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

[19] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *Proc. of the 14 International Conference on distributed Computing Systems*, June 1994.

[20] G. Neiger. A New Look at Memebership Services. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, 1996.

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

[22] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam. Group Communication Support for Distributed Multimedia and CSCW Systems. Technical report, Department of Mathematics Computer Science, Emory University, 1996.

[23] A. Ricciardi and K. P. Birman. Consistent Process Membership in Asynchronous Environments. In K. Birman and R. van Renesse, editors, *Reliable Distrinuted Computing With The ISIS Toolkit*, chapter 13. IEEE Computer Society Press, Los Alamitos, 1993.

[24] L. Rodrigques, K. Guo, A. Sargento, R. van Renesse, B. Glade, P Verissimo, and K Birman. Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory. In *Proc. of the 13th Int. Symp. on Reliable Distributed Systems*, pages 34–41, 1994.

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

[26] W. Vogels. World wide failures. In *ACM SIGOPS European Workshop*, 1996.

[27] B. Whetten, T. Montgomery, and S. Kaplan. A High Performance Totally Ordered Multicast Protocol. In *Theory and Practice in Distributed Systems, LCNS 938*. Springer Verlag, 1994.