Multimedia Computing and Networking 1998
Volume 3310

PROCEEDINGS OF SPIE
SPIE—The International Society for Optical Engineering

Multimedia Computing and Networking 1998

Kevin Jeffay
Dilip D. Kandlur
Timothy Roscoe
Chairs/Editors

26–28 January 1998
San Jose, California

Sponsored by
SPIE—The International Society for Optical Engineering
IS&T—The Society for Imaging Science and Technology

Cosponsored by
ACM SIGMultimedia

Published by
SPIE—The International Society for Optical Engineering

Volume 3310

SPIE is an international technical society dedicated to advancing engineering and scientific applications of optical, photonic, imaging, electronic, and optoelectronic technologies.

Please use the following format to cite material from this book:
Author(s), "Title of paper," in *Multimedia Computing and Networking 1998*, Kevin Jeffay, Dilip D. Kandlur, Timothy Roscoe, Editors, Proceedings of SPIE Vol. 3310, page numbers (1997).

ISSN 0277-786X
ISBN 0-8194-2750-0

Published by
SPIE—The International Society for Optical Engineering
P.O. Box 10, Bellingham, Washington 98227-0010 USA
Telephone 360/676-3290 (Pacific Time) • Fax 360/647-1445

Printed in the United States of America.

Conference Committee

Conference Chairs

Kevin Jeffay, University of North Carolina/Chapel Hill
Dilip D. Kandlur, IBM Thomas J. Watson Research Center
Timothy Roscoe, Persimmon IT, Inc.

Program Committee

H. W. Peter Beadle, University of Wollongong (Australia)
Ming-Syan Chen, National Taiwan University
Wu-chi Feng, The Ohio State University
Martin Freeman, Philips Research
Jose J. Garcia-Luna-Aceves, University of California/Santa Cruz
Anoop Gupta, Stanford University
Mark Hayter, Digital Equipment Corporation
Sugih Jamin, University of Michigan
Paul Jardetzky, Sun Microsystems, Inc.
Chuck Kalmanek, AT&T Laboratories
Ian Leslie, University of Cambridge (UK)
Sape J. Mullender, Universiteit Twente (Netherlands)
Klara Nahrstedt, University of Illinois/Urbana-Champaign
Gurudatta M. Parulkar, Washington University
Lawrence A. Rowe, University of California/Berkeley
Debanjan Saha, IBM Thomas J. Watson Research Center
Henning G. Schulzrinne, Columbia University
Doug Shepherd, Lancaster University (UK)
Brian C. Smith, Cornell University
Cormac J. Sreenan, AT&T Laboratories
Ralf Steinmetz, Technical University of Darmstadt (FRG)
Harrick M. Vin, University of Texas at Austin
Jonathan Walpole, Oregon Graduate Institute of Science and Technology
Rajendra Yavatkar, Intel Corporation
Hui Zhang, Carnegie Mellon University

Contents

v *Conference Committee*

SESSION 1 MULTIMEDIA SYSTEM DEVELOPMENT TOOLS

2 **Middleware support for distributed multimedia and collaborative computing** [3310-01]
K. P. Birman, R. Friedman, M. Hayden, Cornell Univ.; I. Rhee, North Carolina State Univ.

14 **Multiplatform simulation of video playout performance** [3310-02]
L. Gharai, R. Gerber, Univ. of Maryland/College Park

28 **Software-only video production switcher for the Internet MBone** [3310-03]
T. H. Wong, K. D. Mayer-Patel, D. Simpson, L. A. Rowe, Univ. of California/Berkeley

SESSION 2 OPERATING SYSTEMS I

42 **Applying statistical process control to the adaptive rate control problem** [3310-04]
N. R. Manohar, M. H. Willebeek-LeMair, IBM Thomas J. Watson Research Ctr.; A. Prakash, Univ. of Michigan

57 **Integrated input/output system for kernel data streaming** [3310-05]
F. W. Miller, Univ. of Maryland/College Park; S. K. Tripathi, Univ. of California/Riverside

69 **Measurement-based resource allocation for multimedia applications** [3310-06]
P. Barham, S. Crosby, T. Granger, N. Stratford, Univ. of Cambridge (UK); M. Huggard, F. Toomey, Dublin Institute for Advanced Studies (UK)

SESSION 3 VIDEO-ON-DEMAND

84 **Supporting interactive scanning operations in VOD systems** [3310-07]
G. Apostolopoulos, Univ. of Maryland/College Park; M. Krunz, Univ. of Arizona; S. K. Tripathi, Univ. of California/Riverside

96 **Movie approximation technique for the implementation of fast bandwidth-smoothing algorithms** [3310-08]
W. Feng, C. C. Lam, M. Liu, The Ohio State Univ.

111 **Implementation of dynamic service aggregation for interactive video delivery** [3310-09]
P. Basu, A. Narayanan, R. Krishnan, T. D. C. Little, Boston Univ.

SESSION 4 OPERATING SYSTEMS II

124 **Symphony: an integrated multimedia file system** [3310-10]
P. J. Shenoy, P. Goyal, S. S. Rao, H. M. Vin, Univ. of Texas at Austin

139 **Adaptive prefetching for device-independent file I/O** [3310-11]
D. Revel, D. McNamee, D. Steere, J. Walpole, Oregon Graduate Institute of Science and Technology

150 **Resource kernels: a resource-centric approach to real-time and multimedia systems** [3310-12]
R. Rajkumar, K. Juvva, A. Molano, S. Oikawa, Carnegie Mellon Univ.

SESSION 5 THE WORLD WIDE WEB

166 **Experiment to characterize videos stored on the Web** [3310-13]
S. Acharya, B. C. Smith, Cornell Univ.

179 **Static caching of Web servers** [3310-14]
Z. Liu, P. Nain, N. Niclausse, INRIA Ctr. Sophia Antipolis (France); D. Towsley, Univ. of Massachusetts/Amherst

191 **Resource-based caching for Web servers** [3310-15]
R. Tewari, H. M. Vin, Univ. of Texas at Austin; A. Dan, D. Sitaram, IBM Thomas J. Watson Research Ctr.

SESSION 6 MULTIMEDIA APPLICATIONS

206 **Integrated audiovisual processing for object localization and tracking** [3310-16]
G. S. Pingali, Lucent Technologies/Bell Labs.

214 **Accelerating M-JPEG compression with temporal information** [3310-17]
H. Bönisch, K. Froitzheim, P. Schulthess, Univ. Ulm (FRG)

226 **Cross-modal retrieval of scripted speech audio** [3310-18]
C. B. Owen, F. Makedon, Dartmouth College

SESSION 7 FLOW AND CONGESTION CONTROL

238 **Adaptive source rate control for wireless video conferencing** [3310-19]
H. Liu, M. El Zarki, Univ. of Pennsylvania

250 **Flow and congestion control for Internet media streaming applications** [3310-20]
S. Cen, Tektronix, Inc.; J. Walpole, C. Pu, Oregon Graduate Institute of Science and Technology

266 *Author Index*

SESSION 1

Multimedia System Development Tools

Middleware Support for Distributed Multimedia and Collaborative Computing

Kenneth P. Birman[a] and Roy Friedman[a] and Mark Hayden[a] and Injong Rhee[b]

[a] Department of Computer Science, Cornell University
Ithaca, NY 14850 USA

[b] Department of Computer Science, North Carolina State University
Raleigh, NC 27695-7534 USA

ABSTRACT

Maestro is a middleware support tool for distributed multimedia and collaborative computing applications. These applicatic share a common need for managing multiple subgroups while providing a possibly different quality-of-service guarantees for ea of these groups. Maestro's functionality maps well into these requirements, and can significantly shorten the the developm time of such applications. In this paper we report on Maestro, and demonstrate its utility in implementing *IMUX*, a pseudo server (proxy) for collaborative computing applications. Examples of other multimedia applications that benefit from Maes appear in the full version of the paper.

1. INTRODUCTION

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

Typical examples of distributed multimedia and collaborative applications include video/audio conferenci textual chatting, white and clear boards, and application sharing tools. In a video conference session, there be a situation in which participants are connected by a WAN, but subsets of the participants may share the sa LAN. Also, text and video may require different guarantees, and different subsets of the participants in a vi conference may require different quality video streams, depending on their locale and equipment. In addition, au data should be delivered in FIFO order, but does not need stronger end-to-end reliability. Indeed, an attemp overcome infrequent packet loss through a TCP-style flow control and acknowledgment mechanism might introd 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 same order. For video, we would probably want to use a coding scheme similar to the one used in.[2–4] Here, one st is used for low quality frames, which should be delivered to all participants, while another is used for higher qua frames, and delivered only to participants running on nodes that are connected through high bandwidth links to video sender. One can also imagine configurations in which subsets of participants would maintain their own pri chat groups or side-band conferencing sessions. Such a feature might be useful, for example, in a business negotia that brings together multiple representatives of one organization in the context of a larger group of participants

Other author information: (Send correspondence to R.F.)
K.P.B: Email: ken@cs.cornell.edu; Supported by DARPA/ONR grant N00014-96-1-1014
R.F.: Email: roy@cs.technion.ac.il; currently affiliated with the Department of Computer Science, The Technion, Haifa 32000, Isra
M.H.: Email: hayden@cs.cornell.edu; Supported by DARPA/ONR grant N00014-96-1-1014
I.R.: Email: rhee@eos.ncsu.edu; Supported by NSF grant ASC-9527186

The need for multiple levels of quality of service matches naturally with the use of grouping mechanism,[2,6,3,4,7] in which each group employs its own "protocol stack". (A protocol stacks 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.) Developing middleware management tool that supports this functionality is the topic of this work.

In his book *The Mythical Man-Month*,[8] Brooks 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 that eliminate much of this complexity, enabling designers to devote their time to the more essential part of this work.

To facilitate the management 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. An interesting property of Maestro is that the membership and group management services are kept out of the critical path of data messages, and therefore can provide their functionality without interfering with the performance of the system.

Maestro is described in detail in the following section. We later demonstrate how to exploit Maestro's features by implementing *IMUX*,[9] a pseudo X-server (proxy) which transforms X-based applications written for a single-user into shared multi-user applications, without changing their code, on top of Maestro (Section 3). The implementation of Maestro using the Ensemble group communication system is reported in Section 4, where we also present performance data. The paper concludes with a comparison of our work to prior approaches.

2. 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,[10] 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,[11] or external stacks, such as TCP/IP,[12] Cyclic-UDP/IP,[1] user level network interfaces,[13] ISIS,[10] Horus,[14] raw ATM, CCTL,[7] 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 an application library, 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. Maestro can either reside as a library in the same process as an application (as illustrated in Figure 1) or can run independently as a server, or both at the same time. 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 do not have to reside on the same machine as the maestro server. Note that using 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.

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[15]).

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

To simplify and automate the architecture, Maestro introduces a notion of core member *properties*. The co group membership includes a list of properties of each member, which are simple ASCII strings, and subgroups ca be configured to automatically track subsets of core members that have a desired set of properties. For example, co members might specify a property such as "system administrator" or "has an ATM connection." A subgroup can the automatically be created containing just those members that have ATM connections, or those system administrat processes that also have ATM connections. By automatically adjusting subgroup membership in these common case Maestro provides the application developer with an easily exploited facility for creating desired subgroups. For mai 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 partitio by informing the new members about the existing subgroups and their properties. Maestro also provides hooks f the application to do more elaborate *state transfers* if needed.

2.1. Application-defined "Data types"

In order to interact with Maestro, the application defines several data types that specify the properties of groups ai uniquely identify communication endpoints and groups. These data type are represented as uninterpreted sequenc of bytes of arbitrary length, though the group and endpoint identifiers are usually quite small. The only operati 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 identifie these may contain addressing information for the group. For instance, they include the Internet address ai port of an IP multicast group.

endpoint identifier: These are used to uniquely identify communication endpoints. The same endpoint may jo any number of groups, but may not join the same group twice. Additionally, a single process may have sever endpoints.

group properties: Properties are used by the application to advertise a group to other application instances, that they can determine whether or not to join the group. The properties are usually a record with fiel describing the group's ASCII name, security information, the type of the group, the expected bandwidth of t group, a list of members expected to join. They can be extended with other application-specific informatio

2.2. Application-Maestro interface

The first part of the interface is used for sharing information about groups between application processes usi Maestro. Through this interface, an application can create new groups, associate application-specific properties wi each group, and announce groups to other application instances. The announcement facility allows applications advertise new groups and other instances use the information to decide whether to join the groups. At initializati time, the application provides a generic group `new-group` handler function to Maestro. This handler is invoked wi 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 a specifies the properties of the group. The application then requests Maestro to announce the new group to oth members through the `create-group` downcall with the group identifier and the properties. Other members a notified of the new group through their `new-group` upcall. This upcall asks each member to decide whether or r to join the group.

Group properties include a (possibly empty) list of properties. If the list is non-empty, then members havi all of the specified properties will automatically be joined. A second (possibly empty) list of properties identif members that should

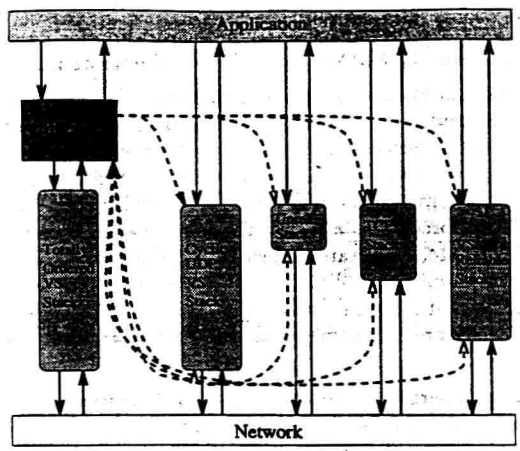

Figure 1. *Maestro system configuration.*

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.

2.3. 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 subgroup. Applications need to be care 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 t members before installing a new view. Maestro first sends a **group-sync** event to all group members via t group object. A subgroup that requires virtual synchrony must then wait for all messages sent over its stack to acknowledged, and then reply with the sidecall **synced-group**. From this point on, subgroups that require virtu synchrony should not send new messages until they receive the **group-view** sidecall (if the applications tries send such messages, it will be forced to wait until the view change is completed). A more detailed description how virtual synchrony is guaranteed appears in Section 4. Synchronization increases the cost of view changes l many groups, such as with multimedia applications, do not require synchronous views, and can consequently ign this constraint. Thereafter, sending messages over that stack and receiving messages from it are done directly the application. However, there are still some cases where Maestro needs to communicate with this data stack, outlined below.

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

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

2.4. Maestro's Membership Properties

Writing a formal definition of group membership, and in particular for virtual synchrony, is a difficult task, wh can be a topic for a paper by itself. (See the frenzy of papers on virtual synchrony in recent years, e.g.,.[16-21]) In t 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 associa with it.

Non Triviality of Views: Every endpoint *p* that sends a **Join** request to a coordinator, eventually become member of the corresponding subgroup unless either the coordinator is eliminated from the core group, or sc member declares *p* faulty.

No Spontaneous View Changes: A member *p* is removed from a view only if some member declared *p* faulty 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 w sent.

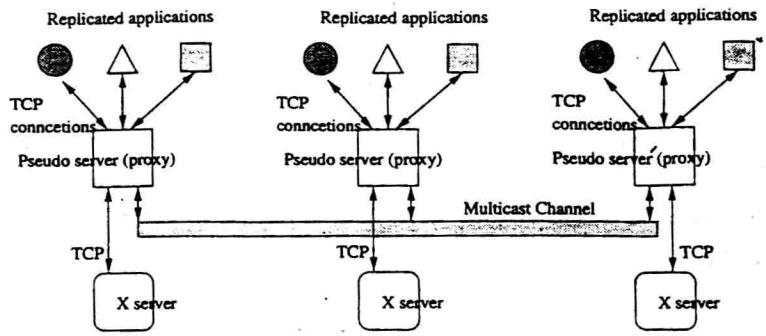

Figure 2. *The IMUX application sharing tool. Here clients are replicated at each participant's site, and output operations (e.g., drawing commands) from clients are routed to the local X server while input events (e.g., user inputs) are multiplexed and communicated to remote replicas via the pseudo-servers. 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.*

3. AN EXAMPLE: 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 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 2. We call this approach *input multiplexing* (IMUX).[9]

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[10] 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 the same application simultaneously. In this environment, pseudo-servers have to ensure that all the replicas the application are given the same sequence of input events so that they all have the same state and view. Tv 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 control the application must grab a token first and only then introduce input events. Only the input events from t token holder are routed to all pseudo-servers which feed the events to their own clients. A user may communica its wish to use a token to all members, and the current token holder releases the token to the requesting user wh it is finished using it. As the approach is simple and does not require a particular ordering among messages fro different users, a FIFO reliable multicast service is sufficient to transport the input events and the token. Howev 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 a totally ordered through total-ordering multicast so that each IMUX will feed the same sequence of inputs to th clients. One difficulty of this approach is that some X events need to be grouped together to be meaningful. F example, a button press event is always followed by a button release event. If two users try to click on a windo region at the same time, it is possible that two button press events are ordered before any of their correspondi release events, which might confuse their clients, possibly leading to an erroneous operation. If this happens, t pseudo-servers must look for matching events from the same user and place them together before any other conflicti events. Since all pseudo-servers perform the same operations on the same sequence received from the total-orderi multicast channel, the "reordered" sequence will be ordered the same everywhere.

These different types of input control can be supported at the same time as per-application basis. The tok passing approach is appropriate for a collaborative session with many users controlling the shared applications wh 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 ongoi session. A latecomer must update its state to be the same as other members of the session. We support this having each pseudo-server store the entire input sequence for each running application. Then, whenever there is change in the membership, one of the existing pseudo-servers, known as the *leader*, sends the stored sequences to t 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 synchror each group member recognizes a new member join at the same logical time and receives the exact same set of messag between any two membership changes. This guarantees that the sequence of events sent to the latecomer are up date with the state at the time it joins, and that the latecomer will take the exact same actions as the existing on and will reach the same state, as illustrated in Figure 3.

We would like to point out that X input events consist of only 48 bytes each. Hence, the buffer requiremer needed for this are not too large. However, in a long-running session, the need to play back all these events m not be feasible. There are known techniques to reduce the number of stored events,[22] but brevity precludes us fr discussing them here.

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

4. IMPLEMENTING MAESTRO USING ENSEMBLE

Maestro is implemented using the Ensemble group communication system.[11] We describe here the overall proto structure used by Maestro as well as several issues in the implementation. These issues include performance a fault tolerance.

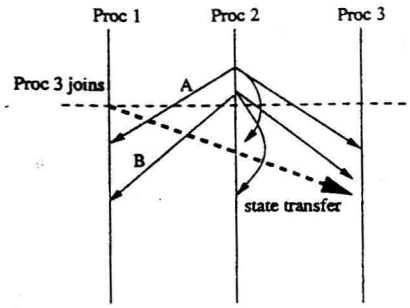

Figure 3. *Latecomer in a virtually synchronous environment.*

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 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,* 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.[10] This is a useful property that greatly simplifies the Maestro protocol compared to virtually synchronous membership protocol built directly on top of asynchronous networks, e.g.,.[23,10,18]

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[24] 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

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

the state of the old coordinator. This usually implies a view change in the subgroup. However, if a core group vi 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 mem of A is located in that process. In particular, this situation allows to keep track of subgroups that were abandon by all their members, so that in the future they can rejoin the subgroup. On the other hand, placing the coordina in a process that includes members of the subgroup, whenever possible, yields better efficiency; by doing so, chan to the core group that do not affect the subgroup are not communicated to the subgroup, while changes in t subgroup's membership that do not affect the core group are only communicated among the relevant process Thus, the coordinator is started by default in the same Maestro process as the first member of the group. Howev when the subgroup membership changes, Maestro may migrate the coordinator to another process, if there are additional subgroup members in the process that holds the current coordinator.

The full version of this paper[25] includes a detailed description of the implementation. These details were omitt from this version due to lack of space.

4.1. Maestro Performance

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

Normal case: Maestro introduces little or no costs in the normal case (when no membership changes occur), beca subgroup members normally communicate directly with each other and only use Maestro for members changes. The only cost is some occasional background communication that is required to detect failures, instance. In fact, applications that do not use Maestro would have to carry this background communicat anyway. When Maestro is used, all of this communication is amortized across all of the subgroups served 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 co group view does not change). The costs for this are different for synchronized and unsynchronized subgrou For unsynchronized subgroups, the cost is one broadcast from the coordinator to install the new view. S chronized subgroups must synchronize first and this adds an additional broadcast and N (where N is the s of the subgroup) point-to-point replies.

Core-group view changes: The final situation involves a view change in the core group. These view chan only occur when a process joins or leaves (possibly through failing) the core group. This is the abnormal c because the core group membership is typically quite static. In many cases, core-group view changes do affect communication in the subgroup. For instance, when a new process joins the core-group, this does affect any subgroups because the new process does not include any subgroup members, nor does it manage such members. When a core-group process fails, the only subgroups that are affected are those with one more members that were managed by that process. In these cases, the new subgroup's view is computed a the core-group's view change has been completed, and is basically done by projecting the core group's v onto the subgroup's view, which can be done locally (fast).

4.1.1. Measured Performance

Recall that Maestro stays out of the critical data path for normal message. Thus, the latency and throughput data stacks using Maestro is the same as if they would have been used without Maestro. Hence, when using Maes the interesting performance data is the latency of the protocol that performs view changes, since this is also latency to join groups, and this is the time interval in which virtually synchronous stacks are prohibited from send messages. It is reasonable to assume that video and audio channels would not use virtually synchronous stacks, therefore would not suffer from hickups during view changes.

All performance measurements were taken on an otherwise idle 8-node IBM SP2. Communication between no was performed using standard point-to-point UDP communication over an Ethernet segment. That is, we *do not* the SP2 fast interconnect, and we did not use IP-multicast because the SP2 does not support it, though Ense supports both (measurements taken using the SP2 fast interconnect show a moderate improvement over Ethe but this medium is not very representative of the typical environment for Maestro).

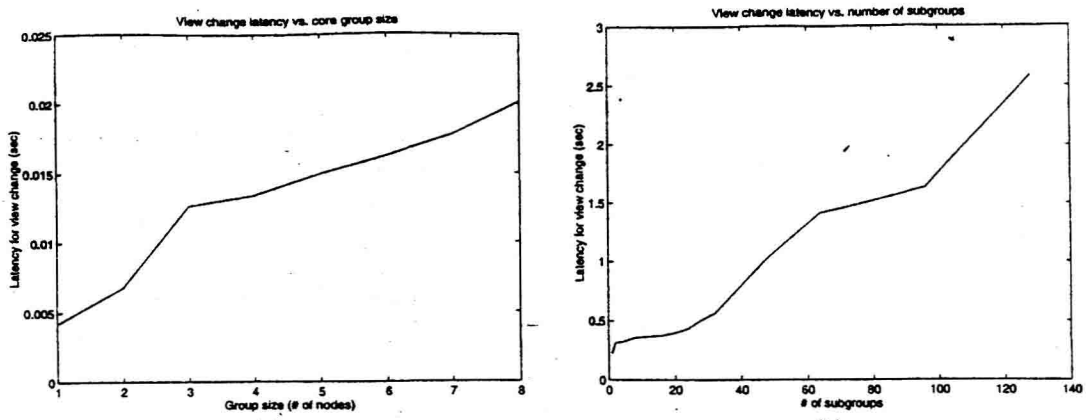

Figure 4. *Latency of the Maestro view change protocol. (a) shows the latency vs. the size of the core group (with one subgroup), and (b) shows the latency vs. the number of subgroups running in separate processes. Process switch overhead causes in the latency in (b) to be worse than that of (a) for the 8-member, 1-group case.*

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 $4ms$ 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.)