Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-mmusic-agree-00.ps

MMusic WG
Shenker/Weinrib/Schooler
Xerox/Intel/Caltech

July 6, 1995
Expires: 12/6/95

# Managing Shared Ephemeral Teleconferencing State: Policy and Mechanism

## Status of this Memo



**Abstract**

In recent years there has been dramatic progress on the enabling technologies for workstation-based multimedia teleconferencing applications. We expect that such applications will soon become an important component of many future social and business interactions. Teleconferencing applications have aspects of their state, such as membership, types of media beings used, and encryption, that are under joint control. Much of this state is *ephemeral*, in that it is of importance only for the duration of a session, and does not have importance outside of the session itself. In this paper we focus on the specification and realization of policies for managing this shared ephemeral teleconferencing state. We first define a broad family of policies which has three dimensions: initiation, voting and consistency. We then consider three different communication models, and for each model present a mechanism which implements this family of policies.

This specification is a product of the Multiparty Multimedia Session Control working group within the Internet Engineering Task Force. Comments are solicited and should be addressed to the working group's mailing list at `conctrl@isi.edu` and/or the authors.

# 1 Introduction

There has been much work in recent years on workstation-based multimedia teleconferencing applications. Significant progress has been made on enabling technologies such as packet transport for audio and video, resource management, connection establishment, scalability and privacy. These applications are now highly usable and, after long gestating as research prototypes, are now vigorously entering the mainstream commercial market. With the increasing availability of audio and video equipment on workstations, and with much faster networks being installed, we expect teleconferencing applications to become an important component of many future social and business interactions. However, while there has been considerable work on underlying technologies for workstation-based teleconferencing, there has been less attention given to higher level aspects of teleconferencing. In this paper we focus on one of these higher-level components: the specification and realization of policies for managing shared teleconferencing state.

Multimedia teleconferencing applications have aspects of their state that are, in general, under joint control. The membership of the conference, the types of media being used, the detailed specification of the media streams (for example, the video encoding), which member has the floor, and how the bill is to be shared are all aspects of this shared *session* state about which the participants must often decide jointly. Much of this state is *ephemeral*, in that it is of importance only for the duration of a session, and does not have importance outside of the session itself; thus, the required level of correctness and consistency can be more relaxed than that required for many applications built using standard database technology. Joint control of ephemeral state is a largely unexplored issue, since the database literature typically ignores questions of policy and focuses instead on satisfying stringent correctness conditions. In this paper we define a family of policies for managing this shared ephemeral teleconferencing state, and then discuss mechanisms for implementing these policies. While our primary motivation comes from teleconferencing, our results are relevant to other arenas that have joint ephemeral state, such as key management for privacy and temporary access control for security.

Even though a teleconference is mediated by technology, it is important to remember that teleconferencing is fundamentally a human endeavor. As with all human endeavors, the exercise of joint control is primarily a social issue, not a technical one; that is, social conventions rather than objective standards dictate how control should be exercised. Given the incredible diversity of such social conventions, we expect teleconferences to utilize a wide variety of decision making criteria, or *policies*, to control this ephemeral shared state; some common examples of such policies are unilateral control, majority rule, and anarchy, but a quick inspection of our election laws, country club membership bylaws, and academic tenure decision processes reveals a much richer set of possibilities. To avoid separate implementations for each social convention in this diverse collection, the mechanism by which such shared state is controlled should support a wide variety of these policies.

Current teleconferencing applications do not meet this standard. To take the most ubiquitous example, the phone system has one particular model for such joint control of membership[1]; there is very tight control on membership since the two ends of the phone call are fixed. On the other hand, the conferencing currently enabled by unencrypted multicasting in the Internet[2] allows a very fluid membership, with participants free

[1] We are ignoring, for the moment, operator assisted conference calls and "gab-line" services in which individuals can call into a bridge within the network.

[2] The most common example of this is conferencing using the *sd* session directory tool and the *nv* and *vat* video and audio tools.

to come and go as they please. These policies are embedded in the mechanism itself; one cannot have phone calls with loose membership styles and one cannot have tight membership control in unencrypted multicasted conferences[3]. In the teleconferencing literature, there has been little discussion about the spectrum of such policies of joint control; instead, there are merely instantiations of particular policies implicitly defined by the various teleconferencing mechanisms. However, we think it crucial that such policy choices be explicitly expressed, rather than implicitly defined, and that the teleconferencing infrastructure support a wide variety of policy choices.

In this paper we carefully separate policy from mechanism. In Section 2, we introduce a framework for expressing a broad family of policies for joint control of ephemeral state. These policies describe who can propose changes to state, and the degree of consensus needed to enact these changes. The policies also describe to what extent the views of state must be consistent when voting and when all changes to state have been executed. This family of policies is not completely general, but we expect that its scope is sufficient to adequately serve most applications. In particular, it expresses a spectrum of policies which span the traditional "tight" control of phone calls to the "looser" control exhibited in today's Internet multicast teleconferences.

In Section 3 we describe several communication models upon which mechanism can be built; these models differ in reliability and in how the transport is addressed. We propose a mechanism that implements our policy family for each communication model; the mechanisms specify how members[4] of the conference send control messages among themselves using the available communication service. These mechanisms combine a two-phase commit algorithm with a timestamp ordering algorithm. Every proposed state change is initiated by a member, the *initiator* of the change. Depending on the policy, the initiator may need to contact some or all members to collect votes as to whether the change should take effect and/or to set locks that exclude other changes for guaranteeing various consistency policies. Finally, the initiator announces the change to all the members of the session. These changes are applied by the members; the order in which these changes are applied can depend on the timestamps. At this point, the change has become *executed*. Care has been taken to ensure that these mechanisms scale reasonably well in the limit of many members and imposes relatively little overhead on the teleconference.

Implementing the family of policies is straightforward when the session state is centralized. However, the teleconferencing applications we consider are inherently decentralized, with each member of the teleconference having its own version of the session state. The distributed nature of teleconferences lends particular significance to the policies regarding state consistency. The correctness of the mechanisms discussed in Section 3 depend upon certain assumptions about the computing and communication context in which they will operate. These assumptions are not universally valid. However, our foremost goal is to provide precise expressions of the various policies; we are willing to concede that the actual implementing mechanisms, when operating in a distributed world with unreliable and unpredictable communications and with finite buffers, may sometimes fail to faithfully implement those policies. In Section 3.2.4 we discuss how the possibility of failure can be accommodated to some extent in the reliable communication model by mod-

[3]Here, we are considering open unencrypted multicast. Using multicast to transport the media streams that make up the conference does not preclude other forms of joint control of membership. In particular, using encryption and key management provided by a session layer allows one to separate distribution from membership, and thus implement any desired membership control.

[4]We use the term *participant* to refer to a person who is taking part in a teleconference, and the term *member* to refer to the computer program that is executing the agreement algorithm on behalf of the participant.

ifying some of the implementing mechanisms and some of the policy requirements to enable sessions to be aborted easily. Since the teleconferencing state is ephemeral, we believe that optimistically relying on the validity of our assumptions, and then facilitating termination of sessions when there is a problem, is an acceptable risk. The alternative would be to use heavyweight fault-tolerant protocols which make very few assumptions about the computing and communication context in which they operate, but whose overhead could well be prohibitive. In short, we choose to approximately implement explicitly and precisely stated policies; we do not merely implicitly define policies by an implementation.

We turn to related work in Section 4. We are not aware of any other formal discussions of the policy issues inherent in controlling shared and ephemeral distributed state. However, there are many existing workstation-based multimedia teleconferencing applications and each of these implicitly define policies. Furthermore, there have been informal discussions of the various styles to teleconferences: open and closed, public and private, and tight and loose.

We conclude, in Section 5 with a brief summary of our work.

# 2 Expressing Policy

This section first introduces our notation. We then describe how policy is specified, identifying three aspects to policy: initiation, voting and consistency.

## 2.1 Notation

Let $S$ denote the shared state that is under joint control of the members of the session. The state $S$ is decomposed into state variables $\{S_i\}$. These variables describe such aspects as the set of members of the session $M$, a description of the session policies, and other state variables that define domain-specific parameters such as media encodings, encryption keys, and the like. We let $\mathcal{T}_i$ be the range of possible values for $S_i$ and define the session state space $\mathcal{T} = \mathcal{T}_1 \times \mathcal{T}_2 \times \ldots$. Each member maintains its own local values for the state $S$, providing to each participant its own view of this state.

During the course of a teleconference, members will occasionally choose to change this shared state. We denote a general change operation by $op$. Changes to $S$ are specified as a composition of separate elemental operations to the state variables $S_i$ that comprise the state. We define the set of operations that change a state variable $S_i$ to be $\mathcal{O}_i$, and denote a particular operation in that set by $O_i$. Each elemental operation $O_i$ changes, and depends on, only the state variable $S_i$, so $O_i : \mathcal{T}_i \mapsto \mathcal{T}_i$. Although not the most general formulation of a change operation,[5] it appears adequate for a significant range of interesting cases. Thus, in our formulation, a general change operation $op$ can be uniquely expressed as $op = O_i \circ \ldots \circ O_j$ where no subscript is repeated. Furthermore, note that for all $O_i$, $O_j$, with $i \neq j$, the operations will commute

[5] In particular, one could imagine conditional change operations that depend on other variables. For instance, the following change command is not expressible in our formalism: "if you are using video codec A then decrease your contrast but if you are using video codec B then leave your contrast unchanged".

($O_i \circ O_j = O_j \circ O_i$) because they act on different state variables. We use the notation that $STATE(op) \subseteq S$ refers to the set of all possible state variables acted upon by $op$.

## 2.2 Policies

Policies are specified along three dimensions: initiator policies, voting policies, and consistency policies. In this subsection we describe these in turn.

### 2.2.1 Initiator

The first dimension of policy is which members may initiate certain change operations. For each elemental operation $O_i$ we define the initiating set of members $I(O_i)$; only those members in $I(O_i)$ can initiate the operation. For a general change operation $op = O_i \ldots \circ O_j$ we then define $I(op) = I(O_i) \cap \ldots \cap I(O_j)$; a member can initiate a general operation if and only if it can initiate each of the elemental operations. For instance, one such policy is that only a member can initiate deleting itself. Our mechanisms, discussed in Section 3, treats in a special way those variables whose changes can be initiated only by a single specified member. For each member $m \in M$ there is a unique largest (but possibly empty) set $U_m$ of state variables such that $STATE(op) \subseteq U_m \Rightarrow I(op) = m$. The sets do not intersect: $U_m \cap U_n = \phi$ when $m \neq n$. While in this case there is a single member who may initiate the change, the change is not necessarily unilateral—the voting policy (discussed below) may well still require that a vote be taken. We define the union of all such sets to be $U = \cup_m U_m$.

### 2.2.2 Voting

The second dimension of policy is voting. For every elemental change operation $O_i$, there is an associated voting rule $V(O_i)$ which takes a vector of votes, with each element taking a value from {YES, NO, ABSTAIN, NO-REPLY} and returns 0 if the vote fails and returns 1 if the vote carries. Voting rules can span the gamut from requiring unanimous consent to requiring no consent at all. The first three responses (YES, NO, ABSTAIN) are explicit responses from other members; the NO-REPLY value indicates that no explicit response was received. In general, as a function of the vector of responses, a voting rule may depend on how specific members respond, not just on the total number of votes of each type. A NO-REPLY response could be caused by the failure of a member process, by communication failure or delay, or because the member wasn't asked. For instance, if a voting rule only requires two YES votes to carry, then the initiator might choose to ask for votes from only two, or a few, members. A voting rule may explicitly take into account NO-REPLY responses (for instance in requiring a quorum), may pessimistically map them to NO replies, or may simply ignore them.

To be more precise, votes are collected for a fixed amount of time. Once the voting is over, NO-REPLY is assigned as the votes for any members not heard from and the voting rule is applied to determine whether the vote carries or fails. Thus, the result of the vote does not depend on the order in which votes are received. The outcome of the vote may be decided before the voting period expires: if at any point during the time votes are being collected the vector of {YES, NO, ABSTAIN} votes received combined with all possible values[6] for the votes not yet received (assuming that the initiator knows a priori who might vote) yields a passing vote, then the vote passes. An operation has a *null* voting rule if no other members need to cast votes in order for the measure to pass (i.e., requires no approval from anyone other than perhaps from the initiator[7], and so $V$ returns 1 no matter what votes were cast).

For a general change operation $op = O_i \circ \ldots \circ O_j$ the resulting voting rule is written as $V(op) = V(O_i) \circ \ldots \circ V(O_j)$; the composed voting rule is satisfied only if the responses satisfy *all* of the subsidiary voting rules in turn. Since any $op$ has a unique decomposition into $op = O_i \circ \ldots \circ O_j$, this voting rule is well defined. An operation involves no voting if the voting rule for each elemental change operation is null.

Members are polled (if required by the voting policy); if the set of responses satisfy the voting criterion then the change is executed. A voting rule is specified for a particular operation on a particular state variable. For instance, the voting policy for adding a member might require that a majority of the current members vote YES, while the voting policy for deleting a member might require that either the member agrees or all but one member vote YES. Moreover, a voting rule may be asymmetric; for instance, the rule $V(O_i)$ could return 1 when either a particular member $m$ votes YES or when a majority of members vote YES. However, we have made the simplifying decision that the voting rules do not depend on which member initiates the change.

### 2.2.3 Consistency

The third dimension of policy is consistency; this dimension is of particular importance when, as assumed in this paper, the teleconference is to be controlled in a distributed manner. Recall that each member has its own view of the session state $S$. There is no other definition of the session state besides the state belonging to each member; in particular, there is no "truth" against which to compare these individual versions. We can only compare these individual versions to each other and require various degrees of consistency.

In general, even if messages are reliably delivered, these individual views will become different when messages related to two or more changes are received in different orders by the members. One policy issue is the extent to which participants want their views to be *eventually* consistent; that is, consistent after all change operations have taken effect. We expect that there will be, in general, a subset of members for which this condition is to hold; we will call these *eventual* members and denote the set of them by $M_e$. In addition, there will be a subset of state for which this condition should hold; we call this subset $E$. The final issue is how to constrain the state that all members in $M_e$ eventually agree upon. To this end, we define *causal* ordering: an ordering of change operations for a member is causal if and only if for each change operation that the member sends out (i.e., executes on the session state), say at time $t$, all executed change operations that have arrived at the member before time $t$ are ordered before that change operation. Equivalently, if a member has seen the message executing change operation $op^\alpha$ before it executes the change operation $op^\beta$, then a causal ordering of the change operations will have $op^\alpha$ before $op^\beta$.

[6] One would expect that well-formed voting rules will, in general, be monotonic in their arguments {YES$\geq$ABSTAIN,NO-REPLY$\geq$NO}, thus allowing the test to be simply assigning NO to the votes not yet received, rather than all possible values.

[7] Some voting rules are null if the initiator approves but are not null if the initiator cast a NO vote; for example, consider the case where there must be at least one YES vote. In any case, the initiator, knowing its own vote, can decide if the voting rule is null.

We define two kinds of eventual consistency, assuming that no further change operations are initiated after time $t_0$:

**Strong Eventual Consistency (SEC).** There is some time $T$ such that all eventual members agree on all state variables in $E$ at all times $t > T + t_0$. Furthermore, this state is the result of a causal ordering of the executed change operations.

**Weak Eventual Consistency (WEC).** There is some power $a > 0$ such that the probability that the eventual members have different views of any state variables in $E$ asymptotically decreases at least as fast as $(t - t_0)^{-a}$. The agreed upon state is the result of a causal ordering of a subset of the executed change operations.

These definitions of eventual consistency differ in two important ways. The strong version requires that after some given time $T$, all eventual members are in agreement on the state in $E$; in contrast, the weak version requires only that the probability of eventual members continuing to disagree converges to zero as some power law in $t$. Furthermore, the strong version requires that the state upon which the eventual members all agree is given by a causal ordering of some sequence of the change operations $op^a \circ op^b \circ \ldots$ applied to the initial state, and that this sequence includes all of the change operations there were executed.[8] The weak version requires only that the state on which the eventual members agree be given by a sequence of change operations that may be only a subset of all executed change operations; that is, some executed change operations may be ignored, as long as all members ignore them.[9].

Note that while all eventual members will eventually agree on the value of the state variables in $E$, it is possible that while casting votes on the various change proposals the members have different views of $E$. Thus, another form of consistency involves the views participants have when votes are cast. More specifically, we identify two subsets of state variables, $C$ and $D$, and a subset of members $M_c$. $C$ is the *critical state* (the state variables that are viewed consistently), $D$ is the *delta state* (the state variables that are acted upon), and $M_c$ lists the *critical members*. The consistent voting condition is then:

**Consistent Voting (CV).** If $STATE(op) \cap D \neq \phi$ then all votes regarding operation $op$ cast by critical members (i.e., members in $M_c$) are cast with an identical view of the state variables in set $C$.

Informally, the critical state $C$ is the state upon which decisions are based, and the delta state $D$ is the state being changed. The CV condition requires that whenever a change is made to $D$ the members in $M_c$ agree on $C$. For this condition to make sense, when critical members are voting they must share a common view of the critical membership, $M_c$. Thus, $M_c$ must be part of the critical state $C$. In addition, we introduce the restriction that only critical members may initiate changes to the state in $C$ or $D$, i.e., $STATE(op) \cap (C \cup D) \neq \phi \Rightarrow I(op) \subseteq M_c$. This restriction is not unreasonable, since critical members in a sense "control" the critical state, and allows for considerable simplification of the mechanism introduced

[8] As we shall see below, the mechanism we define in Section 3.2 actually produces an ordering that only reorders non-causally related change operations that are executed at times closer than the clock skew between the processors.

[9] We shall see that our algorithm that satisfies the WEC condition will rarely remove operations that are far apart in time. In particular, if there are two change operations such that $a$ caused $b$, then if $b$ appears $a$ will also appear (before $b$).

in Section 3. Finally, notice that we have defined the consistency conditions independent of the operation being performed; at the cost of additional complexity, one could define a more general framework in which the consistency conditions are applied depending on the operation. Naturally, the added complexity would be a function of the number of operations allowed on the session state, along with the overhead of tracking potentially different subsets of state variables associated with each operation, e.g., $C$, $D$, $M_c$.

Our mechanisms do not require further restrictions on the various sets that define the consistency policies. However, useful policies may well further restrict the space. For instance, a common choice may be to have $M_c \subseteq M_e$, $C \subseteq E$, and $D \subseteq E$ because it seems likely that critical members would also care about eventual consistency and that the carefully controlled sets $C$ and $D$ should appear consistent to the eventual members in addition to the critical members.

In summary, the consistency policies define three interesting subsets of state: $E$, the eventually consistent state, $C$, the critical state, and $D$, the delta state that requires consistent critical state when voting about changes. Similarly, we have two interesting subsets of members: $M_e$, the eventually consistent members, and $M_c$ the critical members. The policy statements are that (1) all critical members must have a consistent view of $C$ when voting on changes that affect $D$, and (2) all eventually consistent members eventually agree on $E$. Of course, there can be other gradations of membership (e.g., listening only members, nonvoting members, etc.); these are specified by the particular policies adopted by a teleconference. The membership distinctions we are making here are special in that they refer to the degree of consistency of the views of the state, not the content of that state.

#### 2.2.4 Example

Before turning to how to implement this family of policies, we briefly discuss an example in which a core of critical members controls the membership of a multimedia teleconference. The shared state $S$ consists of seven variables: critical members, eventual members, all members, media stream descriptor (describing the media streams that comprise the conference), encryption key (the key currently being used to encrypt/decrypt media streams), common-names of members (the names used by the applications to identify the various members), and next-to-speak (a state variable identifying the member who gets the floor following the current speaker). All members maintain an eventually consistent view of the eventually-consistent state, so $M = M_e$ and we shall not distinguish between them from now on.

Critical membership $M_c$ is in $C$ and $D$, and $M_e$ is in $D$. Thus, only critical members may initiate changes to either type of membership. Adding a member requires a unanimous vote of the critical members; the vote is taken with a consistent view of who is voting (the critical members). Deleting a member is initiated by a critical member, but can occur if and only if the member assents. The media stream descriptor and encryption key is in $E - (C \cup D)$ (i.e., needs to be eventually consistent, but critical members do not need a consistent view of these state variables to vote on changes to $D$ and critical members do not need a consistent view of $C$ to vote on changes to these state variables). The voting rules on the encryption key are null (no vote need be taken) and any member can initiate a change. The common name of member $m$ is in $E - (C \cup D)$ and also in $U_m$ with a null voting rule (member $m$, and only member $m$, may unilaterally make a change to its common name). The next-to-speak state variable is outside of $E$ (since this changes so rapidly, the eventual consistency condition is not considered relevant), and requires a majority rule vote that can be initiated by any member.

This example illustrates how, in a single teleconference, many different aspects of policies can be used, thus motivating the development of a mechanism that can implement the entire family of policies we have defined. We now turn to a discussion of such a mechanism.

# 3 Implementing Policy

This section describes a mechanism for implementing the family of policies discussed above. We are assuming *cooperative* and functioning members; we have made no attempt to design a mechanism that is invulnerable to attack from one of the members, focusing on *implementation* of policy not *enforcement* of policy. While we do not assume cooperation from nonmembers, we do assume either the absence of or the protection from malicious attacks from nonmembers—a session will be safe only if there is underlying protection against malicious attacks such as nonmembers spoofing as members or flooding the members with messages[10]. We provide a logical description of the mechanism, leaving implementation specific details for the future. For instance, one can achieve the logical equivalent of a limited distribution by broadcasting encrypted versions of the messages and then carefully controlling the key distribution.

The appropriate mechanism necessarily depends on the context in which it is implemented. We first describe our assumptions about the computing and communication infrastructure. In particular, we identify three different communication models to consider. We then present a separate implementing mechanism for each model. All of the mechanisms have the same basic structure but differ in important aspects.

## 3.1 Communication and Computing Infrastructure

Associated with each member of a teleconference is a computing infrastructure that has a *local clock* and an *operation buffer*. The local clock is used to timestamp certain messages, and should be thought of as being roughly the member's local notion of time-of-day; however, in order to preserve causality, we will insist that a member's clock is never less than the greatest timestamp it has received (this is accomplished by always inserting into the local clock the greater of current time and an arriving timestamp). We further assume that the clock skew between the local clocks of members is bounded by a constant time $\tau_{skew}$; that is, if at any particular instant the clock readings are denoted by $\{t_i\}$, then $\|t_i - t_j\| \leq \tau_{skew}$[11]. The operation buffer will be used to store the sequence of operations applied to the shared state, and in some cases the timestamps will be used to order the operations. We assume that the operator buffer is unlimited[12]. We assume further that all initiated changes are either cancelled or executed (to be mechanistically described below) within a time $\tau_{decide}$ of being initiated.

[10] We do not explicitly discuss ways to provide security and privacy in this paper.

[11] This bound is preserved by the modification to the local clocks described above.

[12] While formally unbounded, the operator buffer will be unlikely to grow large given reasonable values for the various time constants $\tau$ defined in this section.

There is also an underlying transport service[13] that is used by the teleconferencing application to send a variety of control messages. Here, we are concerned only with the transport *abstractions* made available to the teleconference application, not with the details of how these abstractions are realized on the underlying network. Communication is asynchronous, but we assume all messages, if they ever arrive, arrive within time $\tau_{deliver}$ of being sent.

We consider two different models of the reliability of communication: *reliable* and *unreliable*. In the reliable communication model, all messages arrive at their intended destination on time (within $\tau_{deliver}$), and all messages from a particular sender to a particular receiver arrive in the order in which they are sent; we do not require that messages from different senders arrive in the same order as sent. In the unreliable model, the probability that $k$ consecutive messages from a sender destined for a particular receiver all fail to arrive is $O(k^{-\gamma})$ for some constant $\gamma > 0$. This failure model is quite conservative and when the consecutive messages are separated by enough time we think that this is a reasonable assumption for intermittent failures[14]; it is certainly satisfied if failures are independent and random—in fact, in that case the probability decays exponentially (i.e., $O(\omega^{-k})$ for some $\omega > 1$), not algebraically. In this model, messages from the same sender may arrive out of order.

We consider two different models for how the underlying transport is addressed. In the *Explicitly Named List* (ENL) model, messages are delivered only to those members that are known by the sender to be members. One way to implement this would be as a sequence of unicast messages; another would be using a sender-defined multicast tree. In the *Shared Bus* (SB) model, messages are delivered to all those who choose to receive the messages. This could be implemented using receiver-initiated multicasting, where the sender just sends to a multicast group identifier without knowing explicitly who is a member of the group; it is up to the receivers to join the multicast group if they want to receive messages. Joining a multicast group is entirely different from joining the session—there is no policy on joining a multicast group, one merely informs the routing protocol that one wishes to receive traffic addressed to that multicast group. Note that the key difference is that in the ENL model the sender controls the delivery, whereas in the SB model the receivers control the delivery. We assume that joining a group takes no more than time $\tau_{join}$ to take effect; all messages sent more than time $\tau_{join}$ after a member has indicated a desire to join the group will be directed to the member (but in the unreliable model they may not be delivered).

Combining our reliability and addressing options gives four system models to consider: reliable ENL, unreliable ENL, reliable SB, and unreliable SB. In the following subsections we first discuss the reliable ENL case; making use of the reliable transport, the mechanism we develop supports the full range of policies described in Section 2. The same mechanism works for the reliable SB model. We then turn to the unreliable SB case, which is essentially the model assumed by many of the conferencing tools in use on the Internet today; we introduce two mechanisms for this model that ensure eventual consistency and that scale well but do not realize all of the policies defined in Section 2. We do not focus on the unreliable ENL case since it is of limited interest—there are standard techniques to make unreliable point-to-point connections very reliable (e.g., TCP over IP). We assume that with the unreliable SB model (and the reliable SB model) a sender still has access to a pairwise reliable communication mechanism; distribution to all members uses

[13] Note that this paper focuses solely on the control of a multimedia conference; thus, this discussion concerns only assumptions about the transport for control messages associated with our mechanisms, *not* the transport of the media streams that make up the conference.

[14] The case of a permanent link failure which renders some members as unreachable is rather different; we do not treat that case here.

the unreliable SB model (or reliable SB model) but certain one-to-one control messages can be sent reliably point-to-point.

We are making certain assumptions about the context in which our mechanism operates. If any of these conditions are violated, then the mechanism may fail to properly implement the policy. Such failures might make continuation of the session impossible or undesirable. We state now, but will discuss in more detail later in Section 3.2.4, that to recover from such a situation, members are always able to unilaterally leave a session, thus enabling recovery to a safe state.

### 3.2 Reliable Communication

In this section, we focus on mechanisms that depend on underlying reliable communications. We first consider the reliable ENL model, and describe a mechanism that supports the full range of policies defined in Section 2. We then show that the same mechanism also works for the reliable SB case.

There are three dimensions to policy: initiation, voting and consistency. The implementation of the initiation rules is essentially a local matter—we assume that each member knows the policy and abides by it (we are discussing implementation of policy, not enforcement of policy). The implementation of the voting rules is similarly straightforward; the initiator solicits votes on proposed change operation from the other members, and then based on the responses decides whether or not to execute the operation as dictated by the voting rule. However, implementation of the consistency conditions is not so straightforward. The traditional way of ensuring that distributed databases retain consistent views is the two-phase commit paradigm. We will use a variant of that approach to meet our consistent voting condition. The eventual consistency condition is less standard, but timestamp approaches can be employed (see [1]); we adopt that approach here to meet our eventual consistency condition.

As a preview, we provide a brief overview of the mechanism. Each change in state is initiated by a single member, and this change is assigned a unique identifier for tying together the various messages associated with a particular operation. If the change requires a vote, then the initiator is responsible for soliciting and then counting the votes. If, for reasons of consistency, locking must occur during the change process, then the initiator is responsible for installing and then releasing[15] the lock. Change operations are associated with a timestamp, and members keep a buffer of the change operations so that out-of-order receipts of change operations can be detected and corrected by reapplying the ordered list of operations to the state.

Changes in membership require a somewhat more complex mechanism than other state changes. For clarity, we first describe the basic mechanism, where membership is constant, and then discuss changes in membership. We then discuss how this mechanism applies to the reliable SB model. This section ends with a discussion of when the reliable model fails and how the consistency condition can be modified slightly to allow unilateral departures of critical members.

[15] Because the initiator may fail while holding a lock, a complete protocol will include specification of a time out for locks. We defer this level of detail to future work.

### 3.2.1 Basic Mechanism: Static Membership

The mechanism utilizes several basic message types:

`Poll(Id: id, Operation: op)`. Asks for a vote on the proposed operation; `id` is the identifier that ties together messages associated with a particular operation. The response is {YES, NO, ABSTAIN}.

`Lock(Id: id)`. Asks for exclusive lock from all critical members. Only critical members respond, and the response is {OK, BUSY}.

`Poll_Lock(Id: id, Operation: op)`. Asks for exclusive lock from all critical members and for a vote on the proposed operation from all members. The response from critical members is {YES, NO, ABSTAIN, BUSY}. The response from noncritical members is {YES, NO, ABSTAIN}.

`Announce(Id: id, Operation: op, Time: timestamp)`. The operation is to be applied to the local state of each member. No response. The timestamp refers to the time the message is sent (according to the clock of the initiator), and will be used to order operations. Upon receiving such a message, each receiver in $M_e$ must set its local time to be the maximum of its current value and the timestamp of the message[16].

`Commit(Id: id, Operation: op, Time: timestamp)`. The operation is to be applied to the local state of each member, and the associated lock, if any, is to be released. No response. The timestamp refers to the time the message is sent, and will be used to order operations. Upon receiving such a message, each receiver in $M_e$ must set its local time to be the maximum of its current value and the timestamp of the message.

`Release(Id: id, Operation: op)`. The associated lock, if any, is to be released. No response.

`Response(Id: id, Response: response)`. Response to `Poll`, `Lock`, or `Poll_Lock` message.

A member starts with some initial state $S(t^0)$ and then receives a sequence of change operations $op^\alpha = O_1^\alpha \circ O_2^\alpha \circ \ldots$ with associated timestamps $t^\alpha$. Each member in $M_e$ has an operation buffer that stores the various operations that are to be applied to its state. This buffer keeps track of each state variable in $E$ separately (members in $M_c$ need only keep a buffer for state variables in $E-(C \cup D)$); we denote the buffer for state variable $S_i$ as $B_i$, and the sequence of stored operations is represented by $< O_i^\alpha, O_i^\beta, \ldots >$, where the operations are sorted in order of increasing $t^\alpha$. The current state of $S_i$ can be computed by composing the sequence of operations and applying them to $S_i(t^0)$. When an `Announce` or `Commit` message is received containing an operation $op^\alpha = O_1^\alpha \circ O_2^\alpha \circ \ldots$, the corresponding $O_i^\alpha$ is placed in order in the sequence stored in $B_i$. The states are then recomputed. Because message delivery times are bounded, and clock skews are bounded, these buffers need not store change operations for more than time[17] $\tau_{buffer} = \tau_{deliver} + \tau_{skew}$.

[16] This is necessary for causality and preserves the bound on clock skew.

[17] The time that change operations must be stored is measured in real time, not the timer being used to compute time stamps which may jump by up to $\tau_{skew}$ upon receipt of a message.

The buffer stores all operations with timestamps more recent than $t - \tau_{buffer}$ and the state resulting from all operations with timestamps older than $t - \tau_{buffer}$ is computed to form $S_i(t - \tau_{buffer})$, at which point these operations can be discarded.

There are some special cases where such a buffer is not required because change operations arriving in `Announce` or `Commit` messages can be immediately applied to the current state and then discarded. Changes to state variables outside of $E$ need not be buffered. Furthermore, if a state variable is in $U$, then no buffer is needed because all messages changing that variable will arrive in order (since they are sent by a single member and we assume that messages from the same sender arrive in order). Furthermore, if for a particular state variable all operations commute, then buffering of changes to that state variable is unnecessary.

Each critical member is either in the *busy* (i.e., locked) state or the *nonbusy* (i.e., unlocked) state. Upon receiving a lock request (in either a `Poll_Lock` or `Lock` message), a nonbusy critical member enters the busy state. Upon receiving a `Release` or `Commit` message for the associated lock, the critical member enters the nonbusy state. Here, we have chosen to define a single lock for the entire critical state, including both $C$ (the "readset") and $D$ (the "writeset"). Other options exist, such as having separate locks for these two sets or even separate locks for the different elements of the sets. These options would allow for possible additional concurrency, at the cost of significantly more complexity. The advantage to be gained from this additional concurrency is dependent on the frequency of operations that compete for elements of sets $C$ and $D$.

Any proposed operation *op* can be categorized according to (1) does it require a vote (do any of the elemental operations require a vote), and (2) does it change any state variables in $C \cup D$? Consider the case where the operation *op* changes some state variable in $C \cup D$, and requires voting. Then, the initiator (which must be a critical member in $M_c$) sends out a `Poll_Lock` message to all members. The members respond with one of {YES, NO, ABSTAIN, BUSY} in their `Response` message (noncritical members may not respond BUSY because they do not maintain locks on the critical state). If any critical member responds with BUSY, or if the vote fails, then the initiator sends out a `Release` message and the proposed operation does not take place. If the vote passes, and no critical BUSY responses are received, then the initiator sends out a `Commit` message and all members execute the operation *op* on their view of the state.

If the operation changes some state variable in $C \cup D$, but does not require voting, then the change is initiated with a `Lock` message, and the responses from the critical members are either OK or BUSY. If all these responses are OK, then the initiator sends a `Commit` message; otherwise it sends a `Release` message. Note that for each of these two cases, the `Commit` messages can only be sent after all responses are collected, but the `Release` messages can be sent after the first BUSY response is received.

If the operation does not change any state variable in $C \cup D$, but does require voting, then the initiator sends a `Poll` message and, after collecting enough votes to know if the vote passes, the initiator sends an `Announce` message. If the vote fails, no message is sent.

If the operation does not change any state variable in $C \cup D$, and does not require a vote, then the initiator merely sends out an `Announce` message.

| | VOTE | NO_VOTE |
|---|---|---|
| $STATE(op) \cap (C \cup D) \neq \phi$ | PL,Rs,{C,Rl} | L,Rs,C,Rl} |
| $STATE(op) \cap (C \cup D) = \phi$ | P,Rs,A | A |

Table 1: **Message Exchanges:** `Poll` *messages are indicated by P,* `Poll_Lock` *messages are indictated by PL,* `Lock` *messages by L,* `Announce` *messages by A,* `Commit` *messages by C,* `Release` *messages by Rl, and* `Response` *messages by Rs. Brackets are used to indicate alternative messages, depending on whether the operation is successful or not.*

These message exchanges are summarized in Table 1, where we use the notation that $STATE(op)$ refers to all possible state variables changed by $op$.

In describing this algorithm, we implied that all members receive the same message. It would be algorithmically equivalent to send `Poll_Lock` messages only to critical members and send corresponding `Poll` messages to noncritical members. Similarly, `Lock` messages need only be sent to critical members, and noncritical members need only be sent the `Announce` message if the lock request succeeds. This is a question of implementation.

The mechanism described above satisfies the Strong Eventual Consistency condition and the Consistent Voting condition. The SEC condition holds for the following reason. Because messages are reliably delivered, all final `Commit` or `Announce` messages will be received by time $T = \tau_{deliver} + \tau_{decide}$ after the last message is sent. Whenever there are no messages outstanding, all members of $M_e$ have the same view of $E$ because their buffers have the same contents and the operations are applied in the same order. The causality condition holds on the ordering because if a member has seen the `Announce` or `Commit` message for one change operation before sending an `Announce` or `Commit` message of a change for which it is initiator, then the timestamp on its message must be later since the local clock is made equal to the timestamp on an arriving message if that timestamp is later than the local clock.

The CV condition holds because `Lock` messages are used to ensure that the critical state $C$ is not being changed while critical members are voting on changes to $D$. In addition, the lock ensures that there is at most one proposal to change $D$ or $C$ in progress at any time. Therefore, all critical members process all operations which change $C \cup D$ in exactly the same order, so eventual consistency is achieved for the critical members for state in $C \cup D$ without any further work (i.e., without requiring that critical members maintain buffers for operations on state in $C \cup D$). If some of the state in $C$ or $D$ is also in the eventually-consistent set $E$, then eventual members who are not in $M_c$ (i.e., members in $E - M_c$) must buffer changes to that state and order them by timestamps just as for any other state in $E$.

Notice that the sets $C$ and $D$ appear symmetrically in the algorithm we have defined because we use a single lock to provide mutual exclusion on both of the sets. As discussed above, we could define a more complicated algorithm that makes use of additional types of locks or that makes use of knowledge that certain operations commute; such optimizations would have the potential to allow for more concurrency, but at the cost of significant additional complexity, especially if performance bottlenecks cannot easily be generalized.

This concludes our description of the mechanism when membership is constant. We now describe the additional mechanism needed to accommodate membership changes.

### 3.2.2 Membership Changes

There are three kinds of membership changes: members leaving an existing teleconference, members joining an existing teleconference, and the establishment of a new teleconference.

No additional mechanism is required for a member to leave a session. However, we need to specify that no member initiates a change to membership that is not meaningful; in particular, no member initiates the leaving of someone that the initiator does not consider to be a member, and no member initiates the joining of someone it already considers a member. This prevents inconsistencies if a member leaves and then immediately rejoins the group using two different initiators[18].

The process of joining a group is somewhat more involved. The prospective member (called a candidate) is assumed to be in contact with the initiator of its membership. No such initiation will occur unless the candidate wants to join (we assume that it makes no sense for a member to be forced to join). When the `Commit` message announcing the addition of a new member is sent, the initiator also sends out a state status message to the candidate supplying it with the initiator's view of the current state (contents of operation buffers and initial states as well as the current values of all state variables $S_i$). The initiator will forward to the new member all `Announce` or `Commit` messages that arrive after it sends the state status message for a time period $\tau_{forward}$.[19] This time constant must be at least $2\tau_{deliver}$, thus ensuring that the new member will receive all `Announce` or `Commit` messages sent to the group that arrive at the initiator after the new member has joined. Forwarding of `Poll` messages is optional, but would allow the new participant to participate in voting more quickly. Forwarding of `Lock` messages is not necessary since they are relevant only to critical members; since the critical membership $M_c$ is included in $C$, the joining of a critical member has its own `Lock` message so either the locks will conflict, or the later lock messages will be sent directly to the new member.

Initiation of a session may be accomplished by sending `Poll` messages to all of the proposed members for an operation that builds the proposed session state. This state includes a voting rule that determines whether the session will be established that is a function of how the proposed members respond (whether YES, NO, ABSTAIN, or NO-REPLY). For a successful proposal, only those proposed members that respond with YES are sent `Announce` messages and become members of the session. Alternatively, a session can be initiated by starting with a session containing a single member and adding members one-by-one.

[18] Without this rule, the rejoining might have an earlier timestamp than the leaving (because of clock skew), resulting in an eventual state with the member in question thinking it was a member but the rest of the group thinking it was not.

[19] This algorithm requires that the initiator of a new member remain a member for the time period $\tau_{forward}$. This requirement does not appear too onerous given the expected small amount of time involved.

### 3.2.3 Reliable SB

In the mechanism described above, there was no explicit mention of the addressing scheme except when various time constants were invoked to motivate why the mechanism indeed satisfied the SEC and CV conditions. We claim that the same mechanism can be applied to the reliable SB case; of course, we continue to use the reliable point-to-point communication for the forwarding of messages and for membership joining exchanges. The only difference in the time constants is now that the time for which messages must be forwarded is $\tau_{forward} = \tau_{join} + \tau_{deliver}$ (rather than $2\tau_{deliver}$); this ensures that all messages not forwarded were sent after the joining has taken effect in terms of routing. The time constant $T$ in the SEC condition (the time after which all members agree on state) is still given by $T = \tau_{deliver} + \tau_{decide}$.

### 3.2.4 Failures

Our arguments concerning the correct operation of the algorithm described above are based on the assumption that the transport of messages is reliable with a bounded delay. Transport algorithms exist that provide reliability, but at the cost of potentially unbounded message delay; on the other hand, a fixed upper bound on message delay can only be provided in real networks by dropping late messages. Thus, our reliable communications assumption may occasionally fail in any real implementation. We have made the design choice that it is better to tolerate occasional violations in the consistency conditions when communication is unreliable than to use a more complicated algorithm that maintains correctness in the face of general network or host failures; this latter choice would require that applications tolerate potentially unbounded delays, which is not reasonable. Shared ephemeral state exists solely for the purposes of the session, and so the worst that can happen is that the session becomes useless due to some inconsistency. While it may be tolerable for the algorithm to occasionally enter a useless state when communication becomes unreliable, it is important, however, to allow participants to extricate themselves from such a wedged state.

In the reliable communications algorithm as described up to now, critical membership is in the critical state ($M_c \in C$), and membership in general may be in the critical state as well. Thus, if some critical member responds with a BUSY to a `Lock` request associated with the operation to remove a critical member $m$, or the critical member does not respond at all, then $m$ can not leave. If this is merely a temporary condition, and a subsequent attempt to leave succeeds, then the slight delay to leaving is probably tolerable. However, due to some software error or communication or host failure, the critical member may continue to respond with BUSY (or not respond at all) for an indefinite period, resulting in $m$ being trapped in the session. To rectify this, we modify the policy associated with membership changes and change the description of voting correctness. A member may leave unilaterally if need be (although it may still be advantageous for a member to leave using the standard `Lock` protocol) by sending out an `Announce` message to inform the rest of the session members of its unilateral departure from the session. If such a message is received while an initiator is collecting votes, the departing member's vote is recorded as an NO-REPLY (unless a vote for the proposal was received earlier) and the departure is processed after the lock is released.

Thus, the consistent voting condition is changed: all critical members cast their votes with identical views of $C$, except for critical (and possibly other) membership, and all votes are *plausible*. A vote is *plausible* if it is cast with some view of membership that may or may not include those members that have unilaterally departed. While each vote is cast with some individual view of membership that lies between the full one (the one before all the unilateral departures occurred) and the reduced one (the one after all the unilateral departures occurred), there is not necessarily a single global view of the membership that would explain all the votes.

### 3.3 Unreliable Communications

The previous subsection presented a mechanism that, in the presence of reliable transport, supported the full family of policies described in Section 2. In this subsection we explicitly allow for a nonzero probability that a message is lost. We weaken the consistency conditions to those achievable in the presence of communications failures. In Section 3.2 we implemented a stronger form of consistency, assuming reliable transport, and provided the ability to abort a session (by unilaterally leaving) if transport or host failures rendered the session state unusable. The approach that is more appropriate in practice depends on the likelihood of transport failures and on the need for stronger forms of consistency.

We assert that in the context of unreliable communications there is no adequate way to provide consistent voting[20]. We base this assertion (and it is a design judgement, not a rigorous result) on the ubiquitous use of two-phase commit ideas in such consistency algorithms, and on the reliance upon transport reliability in the two-phase paradigm[21]. In this unreliable context we only support the eventual consistency property. In addition, it is impossible to implement the SEC condition since there is a finite probability that all messages will fail during any time period $T$. Thus, for this unreliable model we realize the WEC condition.

In this unreliable case, we focus on the SB model of addressing; this is very similar to the model presented by receiver-initiated multicast, and so we will use, for convenience, the language of multicast when describing our mechanism. The session state includes the multicast group identifier that is being used to multicast control information to members of the conference[22]. As we stated earlier, we assume that within time $\tau_{join}$ of a member joining the multicast group[23] multicast routing will begin directing all messages sent to the multicast group to this member.

The unreliability of the transport does not greatly affect implementing the initiation and voting policies; we will use essentially the same `Poll` and `Response` messages as in the mechanism described above (we no longer need the locking messages since we are not supporting the CV condition). The key difference between the reliable and unreliable communication models arises in how one executes change operations. Because transport is unreliable, sending a single `Announce` message is insufficient (a single failure can leave the session state permanently inconsistent). Instead, in the unreliable case we will use repeated transmission

[20] Obviously, one can do this by essentially providing a reliable transport layer on top of the unreliable layer, but then this is equivalent to the reliable cases discussed above. Our statement should be read as asserting not that one cannot provide consistent voting in unreliable contexts, but rather that doing so is essentially equivalent to providing a reliable transport abstraction. We have already considered such mechanisms in Section 3.2.

[21] If the transport abstraction is not reliable then typically two-phase commit algorithms use acknowledgements and retransmissions to provide the equivalent functionality.

[22] A member must be given this information (so that it can inform the routing protocol that it wishes to receive traffic sent to this multicast group) using some other mechanism before it can start to receive the control messages.

[23] Recall that joining a multicast group is different than becoming a member of the session; there is no policy restriction on joining a multicast group.

of a `State` message that announces the current value of a particular (or a set of) state variables. We choose to announce the resulting variable, rather than the operation, since (as will become clearer below) that relieves the need for all members to receive exactly the same set of change operations to eventually agree. Thus, the two differences between the mechanisms for the unreliable and the reliable cases are the repeated transmission of a `State` message versus the single transmission of an `Announce` message, and the announcing of the resulting state variable rather than the announcing of the change operation itself. Our mechanisms in the unreliable cases rely on the set of messages:

`Poll(Id: id, Operation: op, Variable: i, Value: value-i, Variable: j, Value: value-j, ...)`. Asks for a vote on the proposed operation, with result as shown. The response is {YES, NO, ABSTAIN}.

`Response(Id: id, Response: response)`. Response to `Poll` message.

`State(Variable: i, Value: value-i, Variable: j, Value: value-j, ... Time: timestamp)`. This announces the content of state variable $i$ (several state variables can be included in the same message).

Note that here we include both the change operation and the resulting state variable in the `Poll` message. This is needed since a member's decision on how to vote may depend not just on the result but on the operation that produced it. Also, the `State` message contains the resulting value of the state variable but not the change operation as in the mechanisms for the reliable case. If the `State` message contained only the change operation then, in order to achieve eventual consistency, every member would have to receive all of the change operations. To ensure that all members received all of them would require continually resending the `State` message associated with each change; by sending the value of the state variable we need only resend the latest version of the state variable.

If the proposed operation requires a vote, then the message exchange is `Poll`, `Response`, and then a sequence of `State` messages. If no vote is required, then there is merely the sequence of `State` messages. Exactly how this sequence of `State` messages is sent out will determine the overhead of the algorithm and its correctness. We consider two possible approaches to sending this sequence.

### 3.3.1 Mechanism 1

The most straightforward sequence of `State` messages would be to have each executed change rebroadcast indefinitely at a fixed frequency $f_0$;[24] more specifically, the cumulative number of `State` messages sent by a member at any time $t$ must be no less than $(t - t_0)f_0$ and no more than $2 + (t - t_0)f_0$ where $t_0$ is the time the member joined the group. This straightforward approach is appropriate when the session state is separable; that is, $\mathcal{T} = U$, and the state in $U_m$ refers only to member $m$ so when member $m$ leaves the teleconference it takes the state in $U_m$ with it—its state is no longer part of the conference state. The

[24] By fixed frequency we mean having no explicit time dependence. However, the frequency could depend on the number of members so as to bound the total rebroadcasting traffic. This is currently done in the *vat* audio tool.

`State` message is sent out with a constant frequency by the owner of the variable, and all variables in $U_m$ are reported (not just those that have changed). There is no state status handoff when new member joins.

This mechanism satisfies the WEC condition. Assume the last change operation was executed at time $t_1$; then the cumulative number of `State` messages sent reflecting the current state of any state variable is bounded below by $(t-t_1)f_0$. The probability that none of these were delivered is of order $O(f_0^{-\gamma}(t-t_1)^{-\gamma})$ for some $\gamma > 0$. Thus, the probability of disagreement vanishes algrebraicly in time as required by the WEC condition; the WEC condition is satisfied for any power $0 < a < \gamma$. Furthermore, the causality is assured since any time there is a causal relationship between two change operations it is reflected in the value of the state variable in the `State` message. However, some change operations may be "lost" (superseded by noncausally related changes).

This is similar to the algorithm used in the Internet teleconferencing tools *vat* and *nv* to maintain lists of members of the conference. Each new member is apprised of the current state by the incoming `State` messages. The lack of an initial state status exchange allows this mechanism to efficiently support an open membership policy (anybody who wants to can join), since then membership is merely announced by beginning to send `State` messages; the new member need not contact an old member to be initiated into the group. If the frequency of updates is fixed, then the overhead of this approach scales linearly with the number of members and with the size of the state in each $U_m$. On the other hand, if the overhead is kept constant by reducing the update frequency as membership increases, then the delay in learning the new state also increases with the number of members.

### 3.3.2 Mechanism 2

Mechanism 1 functions well when the per-member state is small and when $\mathcal{T} = U$ so every piece of state has a natural owner. However, it is less applicable to a more general situation where the size of the state is larger and where state changes can be initiated by multiple members. We now present Mechanism 2, which is designed for this more general situation.

The `State` message[25] is sent out by the initiator of the change operation with a frequency $f_0(t+s)^{-\nu}$ where $t$ is the time since the original `State` message and $f_0$, $s$, and $\nu$ are positive constants with $0 < \nu < 1$; more specifically, the cumulative number of `State` messages sent must be no less than $f_0\left(\frac{t+s}{s}\right)^{1-\nu}$ and no more than $2 + f_0\left(\frac{t+s}{s}\right)^{1-\nu}$. Clearly these `State` messages can be rendered obsolete by subsequent changes. We introduce the suppression policy that once someone hears a `State` message with a more recent timestamp that pertains to some state variable $i$, it deletes any mention of state $i$ from its `State` message (and stops sending altogether if there is nothing left to declare). When a new member joins, the initiator sends the new member a state status message reliably when the joining is announced. This state status message requires the reliable pairwise communication. When a member leaves, it must handoff responsibility for resending its `State` messages to a current member.

This mechanism satisfies the WEC condition. To see this, define $s_0 = t_0 + \tau_{decide} + \tau_{deliver} + \tau_{join}$.

[25] In Mechanism 1 `State` messages were associated with members and each member broadcast their entire state; here we associate `State` messages with particular change operations. Thus every `State` message in a sequence has the same value for the state variables.

If the last change operation is initiated at time $t_0$, then the cumulative number of `State` messages sent for current change operations (i.e., those not rendered obsolete by subsequent changes) at any future time $t > s_0$ is greater than $f_0 \left(\frac{t-s_0+s}{s}\right)^{1-\nu}$. Thus, the probability that all of these messages fail to arrive is of order $O(f_0^{-\gamma}\left(\frac{t-s_0+s}{s}\right)^{-\gamma(1-\nu)})$. Thus, this mechanism satisfies the WEC condition for any power $0 < a < \gamma(1-\nu)$ (since we required $\gamma > 0$ and $0 < \nu < 1$ we know that $\gamma(1-\nu) > 0$). As before, the causality is preserved by the timestamping mechanism.

This mechanism is designed to scale well in the number of updates to state. Consider the case where a particular state variable is changed every $\kappa$ seconds. Then, the traffic due to the `State` messages for that variable is, in terms of messages per unit time, $\approx \frac{f_0\left(\frac{\kappa+s}{s}\right)^{1-\nu}}{\kappa}$. If the `State` messages were sent out at a constant frequency $f_0$, then this overhead would be $\approx f_0$. However, in return for this reduced overhead the mechanism requires the initiation of new members and a state status handoff.

# 4 Related Work

There are several multimedia teleconferencing applications that are relevant to our discussion here. The Touring Machine™[26] system project at Bellcore [2] [3] was one of the first projects to introduce the concept of a session abstraction with specification of associated session policies. The session is implemented in a centralized fashion; thus, policies related to consistency of the shared state were not required.

The Rapport system [4], the Etherphone system [5], the BERKOM system [6], and similar multimedia conferencing system have not focused on issues related to specification of session policy. They tend to use a fixed "call model" to realize the communications abstraction they offer to a participant.

MMCC [7] and IVS [8] are both designed to run over the Internet. MMCC provides a mechanism for an initiator to rendezvous with other potential members to set up a conference using the packet video and audio conferencing tools *nv* and *vat*. IVS also supports rendezvous, but uses its own H.261 compliant coding for packet video and audio transport. Both of these tools have relatively static session policies.

Finally, the IP-multicast based packet conferencing tools *nevot*, *nv*, and *vat* (and probably others that are appearing as we write), fit into our shared bus model, but with unreliable transport. They lend themselves to relatively open conferences, and do not have ways to support the range of policies introduced in this paper. Each member uses unreliable multicast to send out its state—primarily to inform others that it is a member and what is the participant's real name. The session directory tool *sd* is used to inform interested participants of the existence of ongoing conferences, and automates starting up the required tools.

It is important to distinguish mechanisms that coordinate the actions of participants in such conferencing tools from the policy issues we have discussed here. We addressed the issue of how to jointly reach a decision about the shared state, *not* how to turn that state into appropriate actions by the members. For instance, we consider how to decide who can speak in a teleconference but we do not discuss the mechanisms

[26] Touring Machine is a trademark of Bellcore

by which those members who are not allowed to speak are prevented from doing so.

We are not aware of other work that formally addresses the spectrum of policies for managing shared ephemeral state. However, there has been much informal discussion of these policy considerations. These informal discussions refer to several dimensions of policy, such as loose/tight, open/closed, and public/private. Loose control versus tight control refers to both the degree of consensus needed (the voting policies) and how consistent the views of the state needs to be. Open versus closed control typically refers to the membership policies; at the extremes, closed sessions do not allow new members, and open sessions allow anyone to join. Public versus private, to the extent that it is different than the open versus closed distinction, refers to who can listen to the session rather than who can join. In this paper we have defined a family of policies which addresses all of these dimensions, and allow us to vary the policies along these dimensions independently.

Often the policy attributes of being loose, open, and public are associated with large groups, whereas the policy attributes of being tight, closed, and private are associated with small groups. In principle, there is no inherent connection between the size of the group and the policies being used; in fact, none of our policy definitions refer to the size of the group. However, practical issues do force at least some connection between size and policy. The locking technique used to achieve consistent voting is likely to incur significant problems when applied to very large groups. However, the number of (controlling) critical members that must execute the locking protocol may be small in certain situations even when the total membership grows large, thus removing this problem. In addition, the voting mechanism will result in an implosion of `Response` messages at the initiator if the group is large and the voting rules require that all members vote. Even the assumption of reliability will become more problematical as the size of the group increases (since the chance that a message reaches all participants will decrease). We have yet to explore the extent to which size and other practical concerns influence the appropriate choice of policy.

# 5 Summary

Our purpose, in this paper, was twofold: to both *specify* and *realize* policies for managing shared ephemeral teleconferencing state. To that end, we first defined a relatively general family of policies within which application policy requirements could be expressed. These policies addressed the issues of who could initiate changes to the shared state, what the voting rules were on these changes, and the degree of consistency required in the distributed view of the state. The adequacy of this family of policies can only be evaluated relative to the social conventions that arise in future teleconferencing applications.

We then, for each of our three communication models, described mechanisms that implemented these policies. These mechanisms allow applications to combine many different policy requirements while still using only a single implementing mechanism. We will not know whether these mechanisms are sufficiently robust and efficient without implementation experience and testing in real application settings. This is the subject of our current work.

We are also studying mechanisms for models with unreliable communications, with particular focus on those applicable to IP multicast on the Internet. While the full range of policies introduced in this paper are not naturally realized in these models, we are discovering mechanisms that support more limited policies and scale well to the large conferences that are enabled by IP multicast over the *MBone* [9].

## 6 Addresses of Authors

Scott Shenker
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
United States
electronic mail: `shenker@parc.xerox.com`

Abel Weinrib
Intel Corporation
211 N.E. 25th Avenue
Hillsboro, OR 97124
United States
electronic mail: `aweinrib@ibeam.intel.com`

Eve Schooler
California Institute of Technology
Computer Science Department, 256-80
Pasadena, CA 91125
United States
electronic mail: `schooler@cs.caltech.edu`

## 7 Acknowledgments

An earlier version of this paper appeared in the Proceedings of the International Workshop on Multimedia Transport and Teleservices, COST237, Viennea, Austria, Nov 1994.

We would like to thank many members of the IETF Multiparty Multimedia Session Control Working Group and of the Touring Machine project at Bellcore for valuable discussions and advice.

## References

[1] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, H. Sturgis, S. Shenker, D. Swinehart, and D. Terry, “Epidemic Algorithms for Replicated Database Maintenance,” *Proc. of 6th ACM Symp. on Principles of Dist. Comp.*, pp. 1-12, 1987.

[2] Arango et al., “The Touring Machine System,” *Comm. of the ACM*, Vol. 36, pp. 68-77, Jan. 1993.

[3] V. Mak, “Session Management for Distributed Multimedia Applications,” *Proc. 5th IEEE COMSOC Int’l Wkshp. on Multimedia ’94*, Kyoto, Japan, May 1994.

[4] S.R. Ahuja and J.R. Ensor, “Coordination and Control of Multimedia Conferencing,” *IEEE Comm.*, Vol. 20, No. 5, pp. 33-43, May 1992.

[5] H.M. Vin, D.C. Swinehart, P.T. Zellweger, P.V. Rangan, “Multimedia Conferencing in the Etherphone Environment,” *IEEE Computer*, Vol. 24, No. 10, pp. 69-79, Oct. 1991.

[6] M. Altenhofen et al., “The BERKOM Multimedia Collaboration Service,” *Proc. ACM Multimedia 93*, pp. 457-463, Aug. 1993.

[7] E.M. Schooler, “Case Study: Multimedia Conference Control in a Packet-switched Teleconferencing System,” *J. of Internetworking: Research and Experience*, Vol. 4, No. 2, pp. 99-120, June 1993.

[8] T. Turletti, “H.261 Software Codec for Videoconferencing over the Internet”, Research Report 1834, Institut National de Recherche en Informatique et en Automatique, Sophia-Antipolis, France, Jan. 1993.

[9] S. Casner, S. Deering, “First IETF Internet Audiocast”, *ACM Sigcomm Comp. Comm. Review*, Vol. 22, No. 3, pp. 92-97, July 1992.