Teleconferencing Session Management Engine

by

Theodore Jerry Ko

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF SCIENCE

and

MASTERS OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

(c) Theodore Jerry Ko, 1995

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.
Teleconferencing Session Management Engine

by Theodore Jerry Ko

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of Master of Science and Bachelor of Science

Abstract

Modern technology allows people to communicate in ways more sophisticated than the basic telephone call or conference call. Computer teleconferencing systems allow people to create complicated communication sessions involving multiple media, large numbers of participants and complex functionality.

Current conferencing systems have incorporated some notion of policies in their sessions. A policy is a set of rules that governs how a session is controlled. Traditional systems give a participant a small set of pre-defined policy options. Such systems limit the participants to very specific conferencing styles. To date, no system has implemented methods for providing participants with a wide range of styles.

This thesis explores the concept of flexible session policies. Flexibility is attained through providing participants with choices for the session rules. These choices allow a person to specify unique policies on a session by session basis.

The main result of this work is an abstract framework which can be used to describe sessions and their associated policies. Two different conferencing models were coded to identify the implementation issues involved in using the framework. The coding demonstrated that a reasonable implementation of flexible policies is achievable. Further research, including comparisons to other available systems, provided some insight into whether flexible policies is an idea that people would find useful.

The framework also addresses the issue of information consistency requirements across distributed locations. If there is no central location maintaining a session, participants can have conflicting views of the session information. A session’s ‘consistency policy’ determines how consistent these views are required to be. The framework provides the mechanisms for flexibility in this type of policy also, and this thesis further investigates how consistency policy relates to the scalability of session in terms of people and area.

Academic Thesis Supervisor : John Wroclawski
Title : Research Scientist, MIT Lab for Computer Science

Company Thesis Supervisor : Abel Weinrib
Title : Senior Communications Architect, Intel Corporation
I Introduction

A conferencing session is a transient association of people who can communicate with each other. In the world of electronic communication, the most common example is the basic telephone call. A telephone call is a type of session which associates telephone users who communicate with the audio medium. From a person's point of view, the 'protocol' for initiating and receiving a telephone call is simple and widely understood.

With the growth of communications technology, more complicated types of conferences become possible. People can communicate with different types of media such as video or computer application data. Sophisticated computer control of these conferences opens up a wide range of new functions available to the conference participants. The word 'teleconference' is the term used in this thesis to denote conferences that involve multiple media and functions beyond the basic telephone protocol. This work is concerned with the software which creates and controls teleconferencing sessions. In particular, the focus here is on teleconferences which follow the model in Diagram 1. A conference is made up of people who participate through their computers. Control of a conference is maintained by one or more software objects called Session Management Objects (SMO's). Audio and video data is transmitted and received through external devices. The SMO(s) communicates with the participants' computers to control the session and communicates either directly with the hardware which transports the flows of data or with the media software which controls the hardware.

```
Diagram 1
```

```
In this thesis, a teleconferencing session consists of people who are communicating with each other and the software information which determines how users communicate. The term ‘user’ denotes a person running an application which is available for conferencing. A participant is a user who is involved in a session. The software in the SMO(s) which controls a teleconferencing session is responsible for maintaining the information which is jointly used by the participants. This information can be modelled as a collection of variables each representing a distinct number, word, or other bit of information. For example, variables may represent media device status, membership lists, encryption keys, or billing information. The ‘state’ of a session is then defined as the set of values that the variables have at a particular time. Since the state is shared among the participants, the software needs to have policies which determine how the state is to be controlled. Such policies can be separated into three major areas: session initiation, state change initiation and agreement, and state consistency. Session initiation policies determine how a session can be created. Change initiation and agreement policies determine who can propose changes to the state and how much agreement is necessary from the participants before a change can occur. State consistency policies are concerned with information which is distributed among the participants. If the software does not maintain the ‘true’ state of a session in one location, then participants may have conflicting views of the state. Consistency policies specify the degree to which the participants need to have the same view of the state. The particular choices of policies in each area defines the “style” of a session.

This thesis addresses the problem of flexibility in conferencing style. To date, a few multimedia teleconferencing systems have been created, but most provide a limited set of conferencing styles. Their policies often consist of a small set of pre-defined choices. To provide flexibility, instead of presenting a user with pre-defined policy options, a system could support a method of specifying policies which would allow a user to define policies for each particular session. Although some systems have explored the concept of flexible policies, no system has implemented such a method.

In the paper titled “Managing Shared Ephemeral Teleconferencing State: Policy and Mechanism” [1], a framework is given which provides a method for describing a set of session policies. This framework includes a model for collecting state change agreement, a method for describing policy at the level of individual state variables, and mechanisms for achieving desired consistency
conditions. Out of this framework, an 'agreement algorithm' is developed which would allow session management software to implement flexible session policies. A description of this algorithm is given in Section III below. Although the framework outlined the essential ideas in flexible policies, the details of how the policies could be specified were not developed.

The primary goal of this thesis is to develop the ideas in [1] to demonstrate how flexible policies could be realized. So, a 'specification framework' was created to provide a detailed model of how sessions and their policies could be described. Essentially, the research involved expanding and detailing the framework in [1], creating necessary control procedures, and finding the implementation issues that arise in providing policy flexibility.

During this research process, some insight was gained with respect to some larger questions. First, issues in potential uses of a flexible policy framework were considered. Also, the usefulness of flexible policies as compared to specific application policies was examined. Finally, advantages of flexible consistency policies in relation to cost tradeoffs in large conferences were explored.

The format of this thesis report is as follows: Section II defines the type of policy flexibility used in this thesis and makes comparisons to currently available teleconferencing systems. Section III gives a brief description of the algorithm in [1] and outlines the stages of the thesis work. Section IV defines the specification framework. Sections V and VI detail the different conferencing models used to test the framework, including issues which arose during implementation. Section VII presents a summary of the framework produced and various conclusions reached during the development. Section VIII gives the areas in which future work could be done on the framework and the idea of policy flexibility.
II Background

The focus of this thesis is the idea of policy flexibility. Policies are separated into two main types. Control policy consists of the rules governing initiation of a session and changes to the session state. Consistency policy is concerned with the consistency of state information distributed among the conference participants. The flexibility in these policies is achieved in the method of policy definition.

The following sections will first define the type of policy flexibility this thesis is focused on. Then, the motivation for flexible consistency policies will be given. Finally, related work will be described in terms of the way each system handles policies.

II.1 Policy Flexibility

Policy flexibility is based on providing a person with a method for specifying his own policies for each session that he creates. Instead of choosing from a set of policies, the person is given methods for describing the particular policy that he desires. The amount of flexibility given to the person is dependent on the range of policies that these methods allow him to describe.

II.1.1 Control Policies

This thesis bases control policy description on individual state variables. A policy description consists of specific choices for how each variable will be controlled. These choices are made on three parts of a state change: initiation, polling, and result determination. So, a policy for a variable will consist of choices on who can initiate changes to the variable, who will vote on changes to the variable, and what kind of votes are necessary to have a change succeed. A set of choices is expressed as a rule assigned to the variable.

A great deal of flexibility in conferencing policy is attained by allowing the user to specify these rules. Since users can decide how changes will be executed on individual variables, a broad range of different policies becomes available for a session.

II.1.2 Consistency Policies

Distributed conferencing systems maintain the state of a session as information local to each session participant. Hence, each participant has their own particular view of the current state. The degree to which these views agree during a session is set by the consistency requirements of the system.
In systems which follow the teleconferencing model used here, the two available consistency styles are referred to as loosely-controlled and tightly-controlled. In a loosely-controlled scheme, there is little or no effort to maintain consistency between participants' views of the session state. The session state often consists of information that does not need to be shared and the state information which is shared is not always guaranteed to be current. The mechanisms for state information updating often take the form of periodic messages which may not be completely reliable. A tightly-controlled protocol, on the other hand, keeps the participant's views as precisely consistent as is possible. The procedures for session initiation and state changes take extra steps to ensure that operations on the state happen in the same order for all participants and that every participant receives every change.

A session may require consistency conditions which fall in between these two extremes. Ideally, one would want the tightly controlled system for most conferences because users' views of the session would always be consistent. However, the motivation for implementing loose control is that tight control is difficult to maintain over large distances and large numbers of participants. In a large scale session, tight control has a significant overhead in communication costs and extra message processing time. Also, tight requirements are less tolerant of network failures which are more likely to occur in a large session.

The idea behind flexible consistency policy is to allow a conference to specify consistency requirements somewhere between these extremes. In essence, the current loose and tight systems all base their consistency on a single style for the entire session state. Flexibility can be attained by giving the user more freedom to specify which parts of the state need to be tight or loose and which participants need to have a consistent view. This thesis work examines a system of describing consistency requirements which provides finer options on individual state variables so that a session's consistency conditions can be tuned more closely to the needs of the users and the size of the conference.

II.2 Related Work

In the following sections, current teleconferencing systems are described in terms of their initiation and change polices and their consistency mechanisms. These systems are a representative sample of the latest work which is available for public review. For change and initiation policy these systems are examined relative to the range of choices available to the participants. The areas where
choices can be made are state change initiation, state change notification, and state change and session creation agreement style. For consistency policy, systems are compared based on what part of the session state is maintained and whether a policy choice is made available to the users.

II.2.1 Touring Machine

The Touring Machine™ project at Bellcore is a teleconferencing system which provides an abstraction of conferencing functionality [2]. Basically, this system provides an application programming interface (API) which is designed to hide some of the details of multimedia communication. Developers design their applications to work on top of the API without needing to handle resource management, directory services, session management, etc.

Touring Machine implements some simple policies. For state changes the only available policy is a single parameter called “Permission”. This parameter determines who is allowed to make changes to the session and has three possible values: private, protected, or public. ‘Private’ refers to the session initiator, ‘protected’ to all the members, and ‘public’ to anyone. This policy is limited in two ways. First, the possible initiators can only be set to three choices. Secondly, the system has pre-determined the set of users who are notified of changes and the type of agreement they must come to. For any change, Touring Machine notifies only those directly involved with the part of the state being changed. The definition of “directly involved” is not modifiable. Then, the informed participants must agree unanimously to the state change for it to occur. In addition to this one policy, other attributes of the session may be specified but the system itself has no support for user defined policies.

Session state in Touring Machine is maintained at a central software object. Thus, consistency policies are not necessary because all users rely on that one view of the state.

II.2.2 IBM Distributed Collaborative Environment (DiCE)

The Distributed Collaborative Environment (DiCE) developed at the IBM T.J. Watson Research Center [3] has implemented its own framework for describing teleconferences. Various types of conferences are built by combining the applications users run to handle different media types. Essentially, DiCE is focused on bringing end system applications together under a common collaboration paradigm.
The session state is modeled as a collection of participants, applications and media connections. The only variables in the state are the status of the participant's media connections and the overall list of participants. State change policy is limited to "access rights". For each type of media a participant may have transmit or receive permissions. Changes to the state are done unilaterally by a participant if it involves his own media status within his permissions. Otherwise, any changes go through a negotiation process. However, this process is not specifiable as policy.

For state consistency, the DiCE negotiation process is implicitly tightly controlled.

11.2.3 Connection Control Protocol (CCP)

The Connection Control Protocol [4] is a protocol designed to support distributed multimedia conferencing. It provides a call model and a set of basic services for initiating and maintaining a session. The main focus of CCP is to provide a base for heterogeneous end systems. Thus, CCP has defined a set of conferencing fundamentals which end systems make use of.

The limitations of CCP's policies result directly from its model of session state. The state of a session consists of the existence status of participants and the 'process state' each participant is in. Process state is the stage of the protocol that CCP executes to make changes to a session. To determine policy, the participant is allowed to set a small number of policy flags which are mainly concerned with session initiation. CCP provides an inter-participant messaging service for end systems to run their own protocols but has no other support for policies.

Since CCP is mostly concerned with participant's process states, state consistency mechanisms take the form of a synchronization protocol. CCP takes great pains to synchronize participants during an operation. Otherwise, consistency of any other session state is outside the scope of CCP.

11.2.4 Multicast Backbone (MBone) Conferencing Tools

The Multicast Backbone (MBone) enables one-to-many (multicast) data packet transmission over the Internet using specially reserved IP multicast addresses. Lawrence Berkeley Laboratory has produced four software tools intended to enable multimedia conferencing over the MBone. These tools are Network Video (nv), Visual Audio Tool (vat), Whiteboard (wb), and Session Directory (sd). People who are connected to the MBone can create conferences
using sd. A user creates a conference by having his sd announce the parameters of a session including the type of media being used and the multicast addresses reserved for each type of media. Other people who run sd receive the announcement of these sessions and can then join in. Each participant in a conference runs nv, vat, or wb depending on whether the conference includes video, audio or whiteboard information. The video, audio, and whiteboard data is digitized and also sent over the MBone.

The policies of a session are determined by the sd program which creates the session (initiator). Shared state information is limited to the list of session participants and the designation of who is "speaking". This information is maintained by the media tools at each location. Policy is limited to two things: controlling the privacy of a session through data encryption and regulating who is transmitting information, i.e. floor control. The initiator can determine session membership, i.e. privacy, through distribution of encryption keys. Floor control mechanisms also depend on the initiator, but aren't currently well developed. Since the designers wanted 'lightweight' sessions, the initiator does not gather any kind of agreement on its actions.

Part of the motivation for such a minimal shared state comes from the fact that sd and the media tools are used for very large conferences. Also, session control is directly tied to the limitations of IP multicast. Multicast messages are not guaranteed to be reliable and the data can be intercepted by anyone listening to the correct address. These factors motivate a loose consistency style.

II.2.5 International Standards

The International Telecommunication Union (ITU, formerly CCITT) has produced a number of recommended international standards relating to teleconferencing. These standards define a specific model for teleconferencing and then specify the necessary hardware configurations, communication characteristics, and message formats. Overall, the model is heavily based on the telephone call paradigm.

ITU-T Recommendation H.320 [5] outlines the protocol for establishing a connection between two terminals. A 'terminal' consists of the all the hardware a person uses to communicate with video or audio. Signalling between terminals is used to match the hardware capabilities into a common mode for a connection. Note that the terminology in the Recommendation centers around "visual telephone systems". The model of a visual telephone inherently suggests a
simple telephone style call which includes video as well as audio. Terminal hardware is not assumed to have a wide range of extra functions beyond setting up the necessary media connections. H.320 forms the basis for systems which conform to ITU standards, so the visual telephone idea limits the higher level session control functions.

CCITT Recommendation F.730 [6] defines the ITU's basic videoconferencing service. In keeping with the telephone model, the videoconferencing standard is focused on technical specifications for the videoconference connection. Quality of video and audio and connection configurations are emphasized. A person's options are determined by the type of service given to him by the service provider who controls the communication lines. There is little notion of session policy.

Essentially, the F.730 model is intended to be used in a specialized setting. A person first sets up a conference through the service provider. This 'booking' of the communication lines is done outside of the conferencing system itself. Once a call is established, a person controls the media that he sees or transmits. The only notion of 'conferencing style' is in the two 'modes' of conference management. The unconducted mode assigns no priority to any person in the conference in terms of when a person transmits signals. The conducted mode designates one person as the conference chairman and allows him to institute some form of floor control. F.730 does not describe how this floor control is accomplished.

With regards to state consistency, the ITU standards teleconferencing model has no real shared state which conference participants can control. A participant's control functions are all local to the his terminal so there is no shared information that he can change. So, in this model, state consistency questions do not apply.
III Thesis Overview

The premise of this thesis originates in work done in the Multimedia Multiparty Session Control Working Group (MMUSIC) of the Internet Engineering Task Force (IETF). To address issues in flexible session control, a draft paper [1] was written as a component of a larger session control protocol. In the paper, the authors propose a framework for expressing flexible control policies, a method of describing consistency conditions, and protocols for implementing the policies for different messaging capabilities. However, the details of how the framework could be instantiated were not fleshed out. The thesis premise is that the framework and its accompanying protocol could be implemented in a software tool and this tool would be useful in providing flexible session policies. In the following sections, the framework, hereafter known as the "base framework", and algorithm are summarized and then an outline is given of how these were realized.

III.1 The Base Framework

A session is described as a collection of session participants (called members) and state variables. State variables are specific pieces of state information which participants may want to control. Examples include the status of a video source, the media encoding type, or the designated chairman of the conference. Each state variable has a value associated with it and a change to that value is accomplished with an operation on the variable. Note that although members could be considered part of the state and, as such, also a kind of state variable, special handling of members required the distinction between the members and regular state variables.

Operations on variables are governed by three types of policies: initiator policies, voting policies, and consistency policies. Policies are specified per variable rather than on the state as a whole. This allows for a very finely grained control of the state. Initiator policies determine which members are allowed to initiate operations on variables. Voting policies determine the type of agreement necessary among the session members for an operation to be executed. A voting policy is expressed as a voting rule assigned to operations on each variable. Each rule specifies whether a certain set of [YES, NO, ABSTAIN, NO-REPLY] votes from the members passes.

1. NO-REPLY indicates a lack of response from a member rather than an explicit vote
Maintaining session state consistency involves a more complicated set of policies. To allow for different degrees of consistency, variables are individually classified according to the type of consistency desired. A variable can be classified with the terms eventual, critical, or delta. The variables classified as eventual are collectively termed the eventual state (E). Similarly, critical and delta variables are in C and D respectively. Session members are also given classifications eventual and critical. These classifications specify policy in the context of the following two rules:

1) Eventual Consistency - “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, the state is the result of a causal ordering of the executed change operations.” [1] In this definition, $T$ is some point that is a bounded time after the execution of the last change operation.

Causal ordering is defined as “an ordering of change operations for a member is causal if and only if for each change operation that the member sends out, say at time $t$, all executed change operations that have arrived at the member before time $t$ are ordered before that change operation.” [1] If all eventual members follow this ordering, an overall ordering of the operations will emerge. As long as each member accounts for message delay of change notification, each member can make the changes to his view of the state in the correct order.

So, the Eventual Consistency requires all eventual members to execute the same set of operations on the eventual state in the same order. Assuming that all of the members began with the same initial state, this condition will guarantee that they all have the exact same view of the eventual state within a bounded time.

2) Consistent Voting - If a variable modified by an operation is in the delta state “then all votes...cast by critical members are cast with an identical view of the state variables in set $C$” [1]

Essentially, critical members look at the variables in $C$ in order to decide how to vote on changes to $D$.

Again, here one sees that the policies are defined at a fine grained level of the session state. The consistency policy for a session consists of the various classifications assigned to the variables and members in the session.

A key aspect of this framework is the way in which abstraction is used to
hide the actual meaning the state variables have in the context of a teleconference. Policy is specified and enforced in terms of the abstract description of the session. In other words, the software which implements the state changes and maintains consistency does so without concern for the fact that a particular operation may, for instance, turn on a microphone or change the video stream from analog to digital. This abstraction is the source of the policy flexibility because any policy which can be specified in the abstract framework, can be enforced by the software.

III.2 The Protocol

With this base framework, the paper proposes a protocol for implementing the policies. This protocol outlines the messages which pass between members and the sequence of events involved in controlling the session state.

Executing the initiator and voting policies is fairly straightforward. The execution is done in software local to the state change initiator. The local software can determine whether an initiator has the correct permissions for an operation. For voting, execution is simply a matter of determining who needs to be polled, soliciting votes, and collecting the replies. So, the basic protocol consists of the change request, voting replies, and change announce procedures.

To maintain the consistency policies, the algorithm adds locking and message buffering to the basic protocol. To maintain the Consistent Voting condition, a lock is set on the critical state to guarantee that changes are not made to the critical state during operations on the delta state. As noted in [1], this condition only makes sense if the entire critical state is contained in the delta state. Then, message buffering is used to maintain ordering for the Eventual Consistency condition. The proposed message protocol is summarized below.

When an operation on the session state is initiated, the initiating member first determines which members need to be polled. If no voting is necessary, it then checks to see if the operation is on the delta state. If not, the initiator sends out an Announce message announcing the state change. If the delta state is being changed then a Lock message is sent to all critical members. Every critical member must respond “ok” for the lock to succeed. A successful lock results in a Commit message to execute the change and release the lock. An unsuccessful lock results in a Release message to release the lock and the change is not executed.

If the initiator needs to collect votes on an operation, he again starts by
checking whether the change affects the delta state. If not, all relevant members are sent Poll messages requesting a vote on the change. When the votes are collected, if the requirement is satisfied, an Announce message is sent, otherwise nothing is sent. If the delta state is being changed, then critical members are sent Poll-Lock messages which request a vote and a lock. Critical members respond with a vote or “busy” to indicate that the lock is already being held for some other operation. If the lock and the vote succeed, a Commit message is sent, otherwise a Release message is sent to the critical members. So, by locking the critical state seen by the critical members, the Consistent Voting condition is maintained.

For Eventual Consistency, Announce messages are timestamped by each member’s local clock before being sent. These messages are then buffered by eventual members if the changes being made for an operation affect the eventual state. This buffering is done for a certain length of time on each message as determined by the maximum message delay and the maximum skew between local clocks. By buffering the messages, each eventual member can execute the changes in the correct order based on the timestamps.

III.3 The Project

An implementation of the base framework would require a detailed method of describing sessions and policies. Hence, the specification framework was designed. The specification framework allows a session to be abstractly modelled in a way consistent with the agreement algorithm. Policy flexibility is achieved by providing a wide range of access rights, voting rights, voting styles and consistency conditions. A method for specifying operations on variables is also included.

For this thesis, the specification framework was instantiated in software as a set of C++ classes. Then a software object was created to run the agreement algorithm based on the C++ description of a session. This object is called the ‘Engine’ and is intended to be used as a tool by a session management object. The abstraction provided by the C++ session description defines the interface between the SMO and the Engine.

With this abstraction barrier, the Engine need only concerned with initiation, state change and consistency aspects of the session. A SMO will be responsible for managing all other aspects of the conference operations. For example, if a participant initiates a change which turns on a video source it sends the request to the SMO. The SMO translates the request into an abstract request
based on the specification framework and passes the new message to the Engine. The Engine only executes procedures and sends messages concerned with the policies for changing the video source variable and keeping the value of that variable consistent. Then, the actual procedures necessary for turning the physical device on are carried out by the SMO. In short, the Engine handles only the abstract description while the SMO deals with the conferencing details. However, because the Engine handles the consistency of the variables, the SMO regards the abstract representation as the 'true' state of the session.

The work on this project was organized in stages. First, an initial specification framework was created. Development of the first draft involved decisions regarding the range and sufficiency of policies describable by the framework. By translating the session state of the Touring Machine system, additional framework extensions were discovered. Then, the Engine was coded in a centralized control model in order to test the expressiveness of the framework and expose basic implementation issues. The Engine was next implemented in a distributed control model with the addition of consistency policies. In the distributed case, new implementation issues were exposed and some further research questions were identified.
IV Specification Framework

The specification framework is an abstract set of objects which can be used to model a session state and its associated policies. In general, session management software uses a software instantiation of the framework to communicate with procedures which implement the agreement algorithm. For this work, the framework is a guide for how the SMO interfaces with the Engine.

As a starting point, the framework was structured around a model of a session containing participants and variables. Then, design choices were made on what fundamental policies needed to be expressible. The beginning stages of implementation and comparison to other systems then led to additions to the framework. Throughout the design of the framework, many decisions were motivated by the desire to keep the framework simple and basic. Hence, extra features only became part of the framework if a strong need was perceived.

The following sections detail the framework and present the various decisions made during the design. The framework consists of a set of objects which represent the major pieces of a session. Each piece has its own specifiable parameters. To model a session, the parameters are set for each piece and the pieces are combined into a complete description. In general, the framework provides the parameters for each object so that a sufficient range of policies can be expressed.

Overall, a conferencing session is represented by a Session object. Contained within a Session are Member objects and Variable objects. Voting rules and operations on variables are also represented by objects. See Diagram 2. A general description of each object is given below. For a formal description of the framework see Appendix A.

IV.1 Variable

The variable is the basic building block of a session. The value of each variable is either a single piece of data or a list of other variables. The possible variable types are character string, integer, floating point number, or variable list. Complicated data structures were not found to be necessary since the variable list type acts as a container for building structures of variables. Each variable also contains a list of rules which govern how the variable value may be changed.

A variable is designated by a variable name and its owner. Variables can be owned by a session participant (member) or by the session as a whole. Note that ownership is simply a designation used in identifying variables and
Session Object Structure

Session

"MemberList"

"SVars"

Member

"MemberVars"

Variable

Rule

Rule

Rule

Variable

Variable

Initiator

Initiator

Voter

Voter

Diagram 2
specifying voting options. Variables can be identified through their owner and voting rules can reference an owner through one of his variables. The ownership designation does not pre-determine the policies of a variable but does imply an association between the member and the variable. A variable exists in the session state only if its owner exists as a session participant.

Examples of variables include a “video source” owned by member “John” with a value of “on”, or a “video bridge” owned by the session which contains the “video source” and “video sink” variables of each of the members.

IV.2 Operation and Operation Set

Operations on variables are represented by an operation object. The parameters for this object include: variable name, variable owner, operation type, value type, and new value or variable. Each operation is taken on one variable where the operation type can be ‘add’, ‘delete’, or ‘change’. The value type specifies ‘string’, ‘integer’ or ‘floating’ if the variable being acted upon has a single data value. If the operation is of type ‘add’ the object contains the new variable or new member to be added. For ‘delete’ the object has a variable reference1 to the variable to be deleted. Example: to delete John’s “video source” from the “video bridge”, the operation would look like {“video bridge”, “Session”, “delete”, no value type, “John;video source”}

All operations are part of an operation set which allows members to group operation requests. The operation set specifies the name of the session (for use when multiple sessions exist simultaneously) and the name of the operation set initiator. During the execution of an operation set, the entire set is treated atomically. In other words, the whole set is considered one action, and the framework assumes that all operations in the set are intended to occur simultaneously. Thus, members cast one vote for the entire set, and members are notified about changes to the state on a set by set basis.

This atomicity places an implicit limitation on the use of the framework. Operation sets should not be specified such that operations within the same set are dependent on the order changes are made. For example, an operation set should not contain both an operation to add variable “chairman” and an operation to change the value of “chairman”. The change operation would not make sense since the change is set to occur at the same time the variable is added.

1. Variable references are defined in IV.4
Also, an operation to add a member should not be combined with any other operation. If an add-member operation were combined with some kind of change operation, for instance, the new member would not receive notification of the change.

IV.3 Rule

A rule specifies the policy to be enforced for a particular operation on a variable. The type of rule is determined by the type of operation and the specific variable value, if any, it applies to. The three parts of a rule are the initiator list, the voter list, and the voting requirement.

Rules currently support three operations: add, delete, and change. Add and delete are relevant only for variable lists. It was decided that the framework should allow the specification of different rule types for different values of the variable. This decision was required for the cases where different rules are needed for each variable value. So, rule types can be specified relative to the new value of a variable. For instance, one rule may apply when a member requests that a “video source” be set to “off” while another applies when the member requests the same variable be set to “on”. The option to specify rules at this level of resolution does not require the description to be so precise in all cases. Rules can be specified as default for all changes to a variable. I decided that there was no need to allow rules to be set for specific variables being added and deleted from a list, hence the value part of a rule type is only relevant for a change operation.

Example: A user may specify a policy which requires a unanimous vote of all members to set variable “audio source” to “on”. However, he may not specify a rule for the exact case where John’s “audio source” is being added to the “Sources” variable list. He may specify a rule that applies for all additions to “Sources”, but not for the addition of a specific variable.

The initiator list contains the designations of those members who can initiate an operation on the variable. An initiator specification can be a single member name, a group of members, “owner”, “anyone”, or a variable designation. The variable designation is a reference to another state variable which contains the name of a member. For instance, a rule may specify that only the session chairman may initiate an operation where the name of the chairman is stored in the “chairman” variable.

The voter list contains the designations of those members who need to vote
The voter specification can be a single member name, a group of members, "owner", a variable designation, "target", "target owner", "list owners" or "target list owners". The target (T) of an operation is defined as the variable which is being added or deleted to a variable list (VL). Hence, if voter is specified as type "target" then T is a member and VL is a member list. "Target owner" refers to the member who owns T. "List owners" refers to the owners of each of the variables in VL. "Target list owners" is valid when T is itself a variable list and so the voters are the owners of each of the variables in T. Examples: an operation which deletes a member from the session may require a vote of the member being deleted (target); the removal of the session’s "video bridge" may require a vote of each of the members who own a "video sink" or "video source" in the variable’s list (target list owners). Note that in a hierarchy of variable lists the owners of a list may not be easy to determine. Currently, the framework only supports this idea for a list of single-valued variables, i.e. determination of 'list owners' is only one level deep in a hierarchy.

The voting requirement specifies the number of 'yes' votes needed from the voters in order for the operation to be executed. The requirement can be "agree", "majority", "unanimous", "any", or "fraction". The fraction option is used to require a fraction of the total votes from the members polled. "Any" requires only one agreement vote from the set. The other options are simply shorthand forms of the fraction option.

For simplicity, it was decided that the requirement need only be concerned with number of yes votes. Reasonable examples of more complex voting requirements could not be found in the context of a teleconference.

IV.4 Member

A member object represents a session participant. Each object contains the name of the participant and a special variable named "MemberVars". This variable is a list of the variables which the member owns. Rules can be specified for adding and deleting the member’s variables.

Initial implementation exposed the problem of a Variable object instantiated in different places within a session. For example, John’s "video source" is a variable in his "MemberVars" but is also contained in the session owned variable "video bridge". This situation could potentially cause confusion over which instantiation of the variable represents the variable’s existence in a session and which is merely a reference. Thus, the variable objects of type
'variable list' were extended to allow references to variables. The condition was then set that the Variable object is instantiated within the Member object of its owner. Variable lists outside the owner only contain a reference to the variable. As such, to add or delete a variable from the session an operation must be done on the owner's "MemberVars" if the owner is a Member or "SessionVars" if the owner is the session.

So, to use the above example, to create John's "video source" an operation adds the "video source" Variable object to John's "MemberVars". To add John's "video source" to the session's "video bridge" an operation adds the reference "John;video source" to the "video bridge" variable. To remove the "video source" variable completely, an operation deletes the object from John's "MemberVars".

IV.5 Session

A session consists of a list of members and a list of session owned variables. Each session is referred to by the name given to it by its initiator. A session is created with one parameter: creation type. The creation type specifies the kind of initiation agreement among session members that is required for the session to be created. The types are the same as those used for voting requirements, i.e. a session with creation type 'unanimous' requires that all members agree to participate in the session before it can be created, while a session with creation type 'any' only needs a single member to agree to the session for it to exist. In all cases where the creation is successful, only the members who answered 'yes' to the creation request are incorporated in the session.

The session's member list is called "MemberList" and is a special variable. Operations taken on the MemberList are all in terms of adding and deleting members, not variables. However, the MemberList also retains the properties of a normal variable with regards to rules for operations. The list of session variables is called "SessionVars" and is analogous to the "MemberVars" variable in the member objects.

IV.6 Consistency Designations

The policies for consistency conditions in a session are specified by consistency designations based on the system outlined in [1]. Overall, the variables can be classified with four different types of state: critical (C), delta (D), eventual (E), and none. Members have classifications critical, eventual and none. The use of these classifications is defined by the two conditions described in
Section III.

Given these two conditions, the creator of the session decides on the
designations for the variables and members. The specification framework allows
the designations \{C, D, E, N, W, X, Y, Z\} for variables where \(W = C \& D \& E\), \(X = C \&
E\), \(Y = D \& E\), \(Z = C \& D\). Members are allowed \{C, E, N, X\}.

IV.7 Translation example

To test the expressive power of the framework, the Touring Machine
protocol was translated into a framework specification. This translation showed
the utility of parts of the framework, but also exposed some needed extensions.
The translation is described in Appendix B.

IV.8 Use of the Framework

The overall model of how the framework is "used" consists of procedures
in the SMO and the Engine. Remember that the Engine is the software tool which
enforces session policy for the SMO. For each object in the specification
framework the software defines a C++ class, i.e. a Session object is represented by
an instance of the Session class. The parameters for each framework object are
then mapped to class functions of the appropriate class.

Example: The parameters of a Session object consist of the name, the
creation type, the members, and the session owned variables. To create a Session
in C++, the class constructor is called with the name and creation type
parameters. Then, a separate class function is called to add Member class
instances to the session. Likewise, Variable class instances are added with a
function. In this manner, C++ class instances are built up to represent the
description of a Session.

This system of representing framework objects as C++ class instances
allows the SMO and Engine to maintain an abstraction barrier. The SMO receives
a session creation request and translates the requested session into a description
based on the C++ classes. A Session class is built from the ground up. Rule
classes are instantiated with the desired type, voting requirement, and voter and
initiator lists. Variable classes are instantiated with the desired type and value.
Variable classes are further built by adding in the appropriate Rule instances and
other Variable instances. Member classes are instantiated with their names and
then built up with the variables they own. Finally, the Session class is instantiated
with its name and creation type, and all of the Member and Variable instances are
added to it. When the final Session instance is complete, the SMO passes the
entire class instance to the Engine as an argument to the creation request function.

The SMO executes participants’ requests for session operations in the same manner. It receives the request from the participant and translates the request into an instance of the Operation class. All of the operation parameters are set in the instance and, for the operations which add variables or members, the SMO instantiates the necessary classes and adds them to the Operation instance. Note that a single participant request might require multiple operations involving multiple variables. So, the SMO groups the necessary Operation instances into an instance of an Operation Set. The entire Set is then given as an argument to the Engine’s operation request function.

Each of the C++ classes also contains the functions and information which the Engine needs to run the policies. For a creation request, the Engine receives the proposed Session instance. The Engine then calls the CreatePoll procedure defined for the Session class. This procedure determines who needs to be polled based on the creation type. As replies come in from the participants, the Engine inputs the answers to the Session instance, which then determines whether or not the vote passes. Operations are handled in the same way. The Engine uses procedures within the Variable class to determine if an operation is allowable and what rule(s) should apply to an operation. Procedures within the Rule class are then called to determine who needs to be polled for an operation and whether a vote passes.

When the Engine determines the results of a creation or state change operation, it returns these results to the SMO also in the form of C++ class instances. The result of a session creation request is the approved session itself, so the Engine calls a procedure in the SMO to acknowledge session creation and uses the Session instance as an argument. Similarly, completed operations are returned to the SMO with Operation instances as arguments to procedures.

In summary, the abstraction barrier is maintained because the Engine only handles policy based on the C++ class instances given to it. The Engine is not required to ‘understand’ anything about the session beyond the description given in these instances. The SMO is responsible for understanding how the abstract description relates to reality and how the requests from the users are translated before being given to the Engine.
V Centralized Control Model

The first conferencing model used to test the specification framework is based on a centralized control configuration. A centralized setup contains one SMO and one Engine to handle all of the session management duties. All session control operations occur at the one location. See Diagram 3. Since the session state is stored in only one location, the consistency policies are unnecessary. This model was implemented to test the fundamental operations of the framework and expose basic implementation issues.

V.1 Engine API

In the centralized model context, the Engine functions as a session initiation and state change policy tool only. As shown in the figure, the Engine is used as a module rather than a stand-alone object. The set of procedure calls which are defined between the SMO and the Engine makes up the application programming interface (API). As a whole, the API provides a separation of duties. The Engine runs only those procedures which are necessary to perform the duties which are part of the agreement algorithm. Other conference duties such as resource management are performed by the SMO.

Also, communication with the participants is left to the SMO. The only characteristics of the communication that the Engine requires are that the messages are sent reliably and in order. So, the last part of the API barrier is the
communication abstraction it provides. On one side, the communication details can be hidden from the Engine. On the other side, since the users do not communicate directly with the Engine, the users do not need to understand the specification framework.

V.2 Flow of Control

Conference operations are defined as one of two types: session initiation or state change (note that session deletion is accomplished through state changes). The series of procedures which are required to execute these types are nearly the same. The following describes the flow of control during a typical operation. See Diagram 4. For a formal specification of the protocol and API calls, see Appendix C.

Session Initiation -
1) Request: User requests a new session. Application sends initiation message to SMO
2) Request processing: SMO converts message to abstract description and calls Engine to check policy
3) Policy check: Engine determines whether vote is needed. If so, tells SMO to send polling message to all members.
   i) Initiation poll: SMO sends initiation poll message to user application
   ii) User query: Application queries user and sends user's reply to SMO
   iii) Vote collection: SMO calls Engine procedure which tallies votes
       If the vote is 'no', the member is deleted from the session.
4) Result check: Engine waits until all members have replied or timeout occurs
5) For each member still in the session, Engine tells SMO to sends out initiation vote result (commit or abort)
   i) Result notification: SMO sends result to the application. Application informs user.

Session change -
1) Request: User requests a change. Application sends change message to SMO
2) Request Processing: SMO converts message to abstract description and calls Engine to check policy
3) Policy check: Engine checks for initiator validity, determines which
Centralized Model Control Flow

User  Application  SMO  Engine
---  ---  ---  ---
Request  

Req Msg

Req Converted

Policy Check

Poll

Poll Msg

Query

Reply

Reply Msg

Enter Vote

Result Check

Result

Result Msg

Notify

Diagram 4
members need to vote on operation and tells SMO to send out change poll messages.

i) Change poll: SMO sends change poll messages to user application

ii) User query: Application queries user and sends user's reply to SMO

iii) Vote collection: SMO calls Engine procedure which tallies votes

4) Result check: As soon as policy rules pass or fail, Engine tells SMO to send out result (commit or abort)

i) Result notification: SMO sends result to user application. Application informs user.

V.3 Hardcoded Policies

Since the focus of this work is on policy flexibility, an important issue in implementation is hardcoded policy. Hardcoded policies are policies which are coded into the operation of the Engine and are not modifiable on a session by session basis. In general, these policies were hardcoded because there was no obvious need to make them flexible.

V.3.1 Termination policy

The initiation of a session is governed by the policy specified by the initiator. Termination of a session, however, is set to occur when all members have exited. In other words, the Engine will treat a session as active until all of the members have left.

V.3.2 Operation logic

For an operation set to successfully be executed two conditions need to be met. First, for each operation in the set, a logical OR is taken of the voting rules. This means that of all the rules which apply to a particular operation, only one needs to pass in order for the operation to pass. Then, a logical AND is taken of the operations in the set. This means that for an operation set to be successful, every operation in the set must pass.

For example, suppose an operation set consists of two operations: adding John’s "audio source" to the session and changing the video encryption key. Additions to the "Media Sources" list have two applicable voting rules. One rule specifies that any member may initiate the addition and the operation needs a unanimous vote of all the members. The second rule specifies that the conference chairman can initiate the addition without any need for a vote. For the key change, a rule is specified that allows any member to initiate the change and the
change needs a majority vote of all the members.

Suppose then that the operation set is initiated by the conference chairman. Then both voting rules for the first operation apply. If either of the conditions are met, the member addition passes. However, the second operation requires a majority vote. If the majority do not respond yes then the change operation does not pass. If the change operation does not pass, then the entire operation set is aborted.

V.3.3 Rules dynamicity

There is currently no mechanism in the operation object to allow a member to change a voting rule while the session is in progress. Such an action is analogous to changing permission settings in other systems. This capability would not be extremely difficult to implement, but it is unclear whether the additional functionality is useful. Given the transient nature of a conferencing session, it was considered sufficient that the participants can depart the session and re-create another one with the desired rules.

V.4 Implementation issues

Implementation of the Engine brought up several issues, some of which resulted in modifications to the original framework. None of these modifications were of major importance but were deemed necessary for the correct operation of the protocol.

V.4.1 Unchangeable variables

A session may have certain unchangeable variables whose values are fixed at the time of session initiation. Although the framework could express this condition by having a rule with an empty initiator list, this was considered inelegant. Also, the implementation needed a defined action for the situation where a variable has been specified without any rules. At first, an operation on a variable with no rules was handled as an automatic pass. However, the unchangeable variables logically fit this condition better, so the Engine was implemented to disallow operations on a variable with no rules. To handle the automatic pass case, the automatic voting option was added to the rules.

V.4.2 Special variables

As described above, a session contains special variables with reserved names: “SessionVars”, “MemberList”, “MemberVars". Although each acts as a
variable, the initiator does not explicitly specify their creation when specifying a session. "SessionVars" and "MemberList" are created automatically when a session is created. "MemberVars" is created for each member. The initiator does, however, need to give rules and consistency designations for the special variables. So, extra procedures were added for that purpose.

V.4.3 Variable pathnames and references

In the first draft of the framework, variables were referred to by their owner and their individual variable name ("John;video source" or "Session;chairman"). However, the recursive nature of 'variable list' type variables led to difficulties in referencing during implementation. A variable designated by, "Session;chairman" could sometimes be difficult to find if it was nested in a hierarchy of variable lists. Also, one could imagine two different variables with the same individual name, e.g. "video source", which can only be distinguished by their relative positions in a variable list hierarchy. So, to reference a nested variable, variable pathnames were added. All references to variables then take the form of the owner name and a series of variable names ending with the individual name of the desired variable. Examples: "Session;Video bridge; Site1; Sources", "John;Video media; Video sources; camera".

V.4.4 Member groups

The agreement algorithm and the specification framework handle voting in terms of member groups. Rules specify what group of members need to be polled for an operation. For now, there exists only three pre-defined member groupings within the framework: M - the set of all members, Mc - the set of critical members, and Me - the set of eventual members. Other than these three, member groupings exist on a rule by rule basis only. It is possible that a session initiator would need to specify other groups of members separate from a particular voting rule. For example, a national conference may need to associate all of the participants in California. Decisions and voting may depend on the membership in the 'California group'.

Adding a general member grouping feature would require significant additions to the framework. First, a new object would be created to represent a member group. Then, special operations and voting rule options would be necessary. Finally, extra procedures would need to be added to maintain membership information and handle the departures and arrivals of members.
Based on comparisons to other systems, the usefulness of the member grouping feature was not judged to be worth this additional complexity.

V.5 Software Implementation

Overall, the design of the specification framework in terms of objects made translation to an object-oriented programming language easy. Each framework object’s parameters became data fields and an object which contained other objects was implemented with simple list structures. The next part of the implementation consisted of defining how the framework would be used in a conferencing model. The API between the Engine and the SMO defines the separation of duties and the functionality of the framework. The SMO calls Engine procedures to enforce policy. The Engine calls SMO procedures to communicate with the participants. For a listing of the API procedures see Appendix D.

V.5.1 Engine and SMO

In keeping with the model of the Engine as a tool, the code for the Engine was contained in a C++ class called ‘Engine’. So, the SMO took this class definition and instantiated an Engine at runtime. The Engine itself contained only the basic procedures required by the protocol. Most of the policy logic is contained in the Variable and Rule classes. Thus, the Engine procedures essentially matched requests from the SMO to the appropriate class functions. Similarly, the SMO functioned mostly as a dispatcher in this model. No real conference sessions were created by the SMO, so there wasn’t a need for media hardware control procedures. The SMO’s main responsibility then was just to enable the communication between the users and the Engine and demonstrate the API.

V.5.2 Policy procedures

As mentioned before, the bulk of the policy logic was coded directly into the classes which represented the framework objects. The object-oriented nature of the design naturally lent itself to emphasis on class functions and also allowed a blurring of the hierarchical structure of the framework. The Variable class contains the function which actually checks the policy rules for an operation and the Rule class determines who can vote on an operation. Even though the Variables and Rules were ‘contained’ inside a Session, the Session instance could be passed as arguments to the Variable and Rule functions.
Case in point: To execute an operation, the Engine calls a Session class function in order to find the appropriate variable. This variable is passed back to the Engine. The Engine then passes the Session instance to the Variable instance in order to enforce the policy. The Variable class function calls Session class functions to check on session information. Also, the Variable class passes the Session instance to the appropriate Rule instance. It is actually the Rule function which then calls the Session functions to send out messages to users.

V.5.3 Communication Mechanism

The communication between the SMO and the users was accomplished over TCP sockets. The SMO became aware of a user when the user opened a TCP connection to the SMO's server port. Because messages could be large and complex, a message formatting system had to be employed. ASN.1 was used to format the messages for several reasons. First, it provided a structured method for describing messages in a hierarchical manner with was closely analogous to C structures. Second, a simple description of the messages could be automatically translated into C structures. Finally, routines for encoding and decoding the bit fields to be sent over the socket could be automatically generated. So, to translate the C++ classes into a message, each class includes a "Parse" and "UnParse" procedure for translating its information into a C structure. Since each class had its own function, a Session could be recursively translated and reconstructed.

V.5.4 Timing

The SMO runs on an internal master loop. Within the loop, each socket is checked for messages from the users. If a message is present the SMO dispatches to the appropriate Engine procedure and expects an up call. For most of the procedures in the Engine, execution is event driven. The Engine executes a procedure in the event that a request comes from the SMO, rather than at specific time intervals. The only time dependent function of the Engine is the timeout of session operations. When any kind of poll message is sent from the Engine, the relevant operation is assigned a timeout. However, the Engine doesn't keep internal track of time. So, within the SMO's loop, the SMO calls the "Time_Ping" Engine procedure which tells the Engine to check the system time and handle all necessary timeouts.

V.6 Evaluation

The centralized implementation was intended to first test that the
framework could be translated into software. The coding of the SMO and the Engine verified that the API was realizable. To simulate the control of a conference, client application software was written which included a simple Tcl/Tk based GUI. This software was tested for conferences up to three users. Simple test cases with the three users and a small number of session variables were designed to demonstrate the operation of each of the voting choices. Verification was based on observation that the proper messages were received by each user.

Essentially, the centralized implementation successfully instantiated the specification framework. It showed that the object representation of the framework and the abstraction barrier could be implemented and brought up some issues which required framework modifications.
VI Distributed Control Model

The distributed control setup consists of a SMO and an Engine for each user. Control of a session is initiated at any of the user locations. See Diagram 5. Since the session participants now execute control based on their own view of the state, the consistency policies become relevant. It is important to note that even though this model implements a set of procedures different than the centralized model, the separation of duties remains essentially the same. This distributed model was intended to test the consistency policy aspects of the framework and determine distributed implementation issues.

VI.1 Engine duties and API boundaries

The duties of the Engine in the distributed model involve all of the state control policies and the associated inter-user communication. As shown in the diagram, there are now two boundaries on the Engine: the SMO, and the Communication Mechanism (CMech). Here, the boundary between SMO and application is not important to the Engine.
The SMO to Engine boundary is, as in the centralized model, a two-way procedure call interface. It remains the abstraction barrier where the specification framework is used, but the protocol is different due to the new position of the Engine. In one direction, the SMO calls procedures for the requests of only one user. So, operation requests and answers to queries go through the Engine. In the other direction, the Engine sends queries to the user application through the procedures the SMO provides. From the viewpoint of the SMO, the Engine handles all of the policies. Since the state consistency mechanisms generally need not be visible to the user, the Engines handle the consistency policies invisibly. The degree to which the SMO is aware of state consistency is only in the guarantees the policies provide, not in the actual workings of maintaining the conditions.

In this model, the communication mechanism is abstracted behind another two-way procedure call interface. This interface is an abstraction barrier in that the Engine is not required to know the form in which information is sent. So, to send messages to other Engines, an Engine calls procedures defined by the CMech. The procedures take abstract objects as arguments. On the other end, the receiving CMech unpacks the messages and calls upward to its Engine again with abstract objects.

VI.2 Communication Type

Consistency policies are inherently tied to the way in which the members communicate. For the Eventual Consistency condition, the Engines are concerned with the ordering of operations so the transmission delay of operations is important. Also, the exchange of messages is dependent on the reliability of the messages. In [1], the authors present different protocols to enforce the policies dependent on the type of communication being used. Of the four types defined in [1], the Reliable Explicitly Named List type was assumed for the Engine. This means that the Engine assumes from the CMech certain guaranteed characteristics:

1) Messages are delivered error free or loss of communication is detected
2) Messages from a single source arrive in order
3) The concept of directed messages to explicitly named destinations is supported
4) Message delivery delay has an explicit upper bound
Given these characteristics, the Engine can execute one of the protocols given in the paper. The two-phase protocol is still used to get agreement on voting, but a locking system is added to handle the Consistent Voting condition. In order to guarantee the consistent views in this condition, the critical state must be locked during an operation. For the Eventual Consistency condition each Engine maintains operation buffers which order all eventual operations by timestamp before committing the change to the session state.

As noted above, the agreement algorithm is based on group communication. The ideal communication mechanism would be one which implements a multicast style message delivery. In other words, a message could be sent to all intended recipients at one time rather than individual point to point messages. Obviously, for a large conference, point to point messages would incur significant communication costs. To date, there exists no general mechanisms that implement this multicast while satisfying the necessary communication characteristics. So, for the purposes of this thesis, the Engine assumes a point to point system and explicitly sends separate messages to each member.

VI.3 Flow of Control

The distributed model executes a significantly different flow of control than the centralized model due to the changed duties, position and interfaces of the Engine. A flow diagram is given in Diagram 6. For a formal description of the protocol procedures see Appendix E.

Session Initiation:
1) Request: User requests initiation by calling SMO procedure
2) Request Processing: SMO converts message to abstract description and calls procedure in Engine
3) Policy Check: Engine checks if vote is needed. If so, sends polling message to all members
   i) Initiation poll: CMech sends initiation message to peer CMech
   ii) Poll processing: Peer CMech receives message and informs local Engine. Engine converts request and tells SMO to inform the user.
   iii) User Query: SMO queries user and receives reply
   iv) Reply processing: SMO give reply to Engine. Engine sends reply to initiator via its CMech
   v) Reply collection: Initiating CMech receives reply and calls procedure to tally vote in initiating Engine. If the vote is 'no', the member is
4) Result Check: Initiating Engine waits until all members have replied or timeout occurs

5) For each member still in the session, initiating Engine tells CMech to send out polling result and tells SMO to notify initiating user
   i) Result Processing: Peer CMech receives result and dispatches message to Engine. Engine sends result to SMO.
   ii) Result Notification: SMO informs user of result

Session change:
1) Request: User sends change request to SMO
2) Request Processing: SMO creates object and calls session change procedure in Engine
3) Policy Check: Engine checks initiator validity and determines which members need to vote on operation; If operation is on delta state, voting critical members receive poll-lock message; non-voting critical members receive lock message. Non-critical voting members receive poll message.
   i) Initiation Poll: CMech sends message to peer CMech
   ii) Poll Processing: Peer CMech dispatches message to Engine
      If lock requested, critical member Engines check if lock already held, if so, send lock busy message. If not, non-voting members accept lock.
   iii) User query: Voting member Engines tell SMO to query user. SMO queries user and receives reply.
   iv) Reply Processing: SMO gives reply to Engine. Engine sends reply to initiator via its CMech
   v) Reply Collection: Initiating CMech receives reply and calls procedure to tally vote in initiating Engine
4) Result check: As soon as policy rules pass or fail or if ‘lock busy’ reply is returned, Engine tells CMech to send out polling result and notify initiating user.
   i) Result Processing: Peer CMech receives result and dispatches message to Engine. If lock was being held, lock is released. Engine sends result to SMO.
   ii) Result Notification: SMO informs user of result
Distributed Model Control Flow (cont.)

User — SMO — Engine — CMech — CMech — Engine — SMO — User

- User Notify
- Result Check
- Result
- Result Notify
- Result Convert
- Result Dispatch
- Result Msg
- Ans Dispatch
VI.4 Implementation issues

For the most part, implementation issues in the distributed model are concerned with the communication side of the protocol rather than the framework itself. The consistency designations are the only extension to the framework for this model and implementing the designations was straightforward.

VI.4.1 User registration

As stated above, the Engine assumes a communication mechanism which sends messages based on explicitly named receivers. However, this requires the Engine to have some knowledge of who is available for conferencing. I decided that it is not one of the Engine’s duties to announce the existence of its user. The assumption is then that each SMO will “register” available users with its associated Engine. The Engine is provided with a user name and the user’s Internet address. Since this address is given in string form, appropriate changes to the SMO and CMech objects could be made to reference users in a way different than Internet addressing.

VI.4.2 Timestamps and buffers

To maintain the causal ordering required by the Eventual Consistency condition, operations which affect the eventual state are timestamped in accordance with the algorithm given in [1]. The scheme has two parts. First, each Engine has its own clock with which it timestamps operations which it has initiated and which are ready to be committed. To set the ordering an Engine checks the timestamps of the operations it receives and keeps track of the latest timestamp. The Engine makes sure that any operation it stamps is given a time later than any operation it has received.

Then, all Engines run by eventual members buffer incoming eventual operations for a time based on the maximum communication delay between participants. The operations are buffered in order of their timestamp. When an operation has been buffered longer than the maximum delay, the changes are committed to the session state.

The first issue here is in the implementation of these clocks. The scheme in [1] calls for the Engine to modify its local clock to handle skew which would cause misordering. The implementation for this thesis instead uses computer system clocks and simply keeps track of the latest operation time received. So, the local system clock is used to stamp operations, but the buffer latest time is
checked first. If the stamp needs to be changed, it is set to occur immediately after
the latest buffer time, but of course, the local system clock is not changed.

For buffering operations, the amount of buffering is dependent on the
network delay. However, it was not obvious how the CMech could inform the
Engine of the maximum delay possible in a given situation. At the moment, a
fairly arbitrary number is used. The number was chosen to encompass what
seemed to be reasonable network delay while not being so large as to prevent
progress of the session changes. Operations will not remain in the buffer for
longer than this maximum delay, so a bounded buffer size can also be
determined. Ideally, as a session scales up in distance, the CMech would have
some way of measuring the maximum delay and thus informing the Engine of the
necessary buffering time.

VI.4.3 Timeouts

As part of its communication responsibilities, the Engine needs to be
concerned with protocol failure due to network problems. Specifically, if the
communication breaks down, then an Engine needs to avoid waiting forever for a
message. So, the Engine sets timeouts after which a user’s vote is recorded as ‘no
reply’. The question then arises as to how these timeouts should be set. The
various user applications may have their own predefined timeouts and in fact are
not guaranteed to be the same across the whole session. If applications are
allowed to set the timeouts, should they be set per Engine or per session? Also,
the timeouts should take into account the size of the session to allow for large
communication delay. The Engine’s timeout number is currently set to a large
value.

VI.5 Software implementation

The bulk of the code for the distributed implementation was first
transferred straight from the centralized implementation. Then consistency
designation fields were added to the Variable and Member classes. A new class,
CommObj, was created as an abstract representation of the communication
mechanism. The new flow of control then required two API boundaries. A listing
of the procedures in both API’s is given in Appendix F.

VI.5.1 Engine, SMO, and CommObj

In the new model, the SMO has fewer responsibilities than in centralized
model. Each individual SMO is concerned only with its own user and does not
handle communication with other users. As such, the SMO is just an extra procedure call between the user and Engine. So, for the purposes of this model, the SMO and client were combined into one software object which still maintained the Engine to SMO API. Most of the internal code to the Engine remained the same. The additional checking for consistency conditions was added to the class functions of appropriate classes.

The CommObj took over the duties of communicating with other users. Each CommObj maintained the connections to other CommObj’s and all the functions which were in the SMO for communicating with users were transferred to the CommObj. The SMO does, of course, retain procedures that allow the Engine to notify its own user of session operations and request votes from its own user.

VI.5.2 Policy procedures

The basic policy execution procedures remained the same between the two models. The Engine took the same steps and called the same class functions to execute operations. Within the class functions some changes were made to account for the change in user communication.

The addition of consistency policies required two major modifications to the code. For the Eventual Consistency condition, operation buffers were added to the Session class. The buffer was implemented as a linked list of operations ordered by timestamp. It also contains a function for returning the latest timestamp. The maximum buffer time was hardcoded into the Session class.

For the Consistent Voting condition, a simple lock was added to the Session class. An integer field signifies whether or not the critical state of the Session was locked and new messages were defined for requesting locks on the session state. Also, extra steps are taken during operation execution to make sure that changes aren’t being made to the delta state while the lock is active.

VI.5.3 Timing

In the new model, the Engine remains a tool which does not retain an internal concept of time. So, again, the SMO software is require to call “Time_Ping” to tell the Engine to execute the necessary periodic functions. The Engine still uses this procedure to check on timeouts but with the addition of the CommObj, the Engine needs to make sure incoming messages from other users are checked. The Engine calls the CommObj procedure “Msg_Dispatch” to tell the CommObj to check for incoming messages and dispatch them to the
appropriate Engine procedure.

VI.6 Evaluation

The distributed implementation was coded as a rudimentary test of the consistency policy algorithm. A critical idea here was that the policy execution procedures in the API between the SMO and the Engine could be kept relatively intact when switching from centralized to distributed. This fact helped to further validate the API as an effective abstraction barrier.

For testing, user software consisted of a simple text interface to the Engine procedures. Again, conferences consisting of three users were created and simple test cases verified that the basic policy messages were sent correctly. Further tests verified that critical state locking succeeded for changes to the delta state. Finally, several operations were executed nearly simultaneously to see if the buffers correctly ordered changes to the eventual state. All of these simple tests succeeded, but unfortunately, time and resources did not allow more complicated tests.

As a whole, this implementation succeeded in instantiating a distributed model created to make use of flexible policies, but did not fully implement the algorithm presented in [1]. Issues in communication reliability and delay were not handled in the way that a complete conferencing system would need. The communication model was assumed to be one particular type of the four given in [1]. The algorithm based on this model depends on certain communication characteristics, such as maximum message delay, being available to the software. Since there was no method for providing those characteristics the implementation didn't explore the issues involved there. Also, the framework in [1] further acknowledged that special consideration was needed for membership changes in a distributed conference. Of the procedures described, unilateral departure of members was implemented while message forwarding to new members was not. In summary, the distributed implementation provided an example of how the basic model could be designed, but did not explore all of the precise details.
VII Summary and Conclusions

VII.1 Specification Framework Summary

The specification framework was designed to capture the essential components of the base framework and provide the details which would allow the ideas to be realized. The base framework required a method for describing an abstract session consisting of participants and variables. Creating the Session, Member, and Variable objects accomplished this by providing an abstract way to relate to the main session components.

This abstraction then needed a method for expressing changes to the session state. Specifically, any change is composed of single operations on individual state variables. The Operation object expresses a change to a variable while the Operation Set allows these changes to be composed into one action. Within the base framework, variables were given only single values and operations only changed that value. The specification framework, however, provides for variables which are lists of other variables. Thus, the Operation object required an extension of the original concept. An Operation can be an addition or deletion from a list instead of just a change of a single value.

The agreement policies for executing the operations were accordingly defined at the level of individual variables. The base framework describes these policies as voting requirements assigned to each variable. To execute an operation, the appropriate members are polled for an answer of 'yes', 'no', or 'abstain'. How the voting requirements are specified and how the appropriate members are determined are not detailed. The specification framework provides these details while also extending the agreement policy idea. For the voting requirement, the specification framework allows very specific detail on how 'yes' votes need to be collected. Beyond just a minimum number of 'yes' votes, votes which are dependent on particular members or groups of members can be expressed. However, more complicated policies involving the counting of 'no' or 'abstain' votes are not expressible. The specification framework also extended the policy idea in terms of fine grained control. Policy can not only be specified for each variable but can also be specified for distinct values of a variable, for types of operations on the variable, and for particular initiators of the operation.

For consistency policies, the specification framework follows the exact outline given in [1]. The session state variables are assigned to certain subsets of the state depending on their consistency requirements. Members are similarly
divided into their respective groups. These divisions are accomplished through consistency designation fields in each object.

So, given the model of a session consisting of participants and variables, the specification framework has the capabilities to express nearly the full range of session styles and policies outlined in the base framework. The specification framework has successfully captured the main concept of policy flexibility through abstract state description and fine-grained state control. Questions of the usefulness of the specification framework and policy flexibility are addressed in the following sections.

VII.2 Policy Flexibility Issues

The research and implementation of this project has led to identification of some larger questions of the usefulness of policy flexibility. First, there exist issues with how the specification framework could be used in a full system. How could it be implemented and how could applications make use of it? Then, the other main concern is whether users would find the flexibility useful. Are the tradeoffs necessary or reasonable? Some insight gained into these questions is described below.

Initially, the concept of policy flexibility was conceived as part of the larger research into a general session protocol. The abstract method of specifying policies can be considered as a subset of a protocol designed to handle all aspects of session management. So, the Engine model has shown one way in which the specification of policy can fit into session management. The policy handling mechanisms are kept distinctly separate from other management functions by providing an abstract description of a session. One could imagine other configurations where the specification framework objects were included into a larger protocol instead of being kept separate.

To use a system with this style of general session management, however, requires that users understand the type of flexibility being provided. While the abstract framework could be used to match the session control functions of existing applications, this would be a waste of the policy flexibility. Essentially, existing end applications already limit the policy choices seen by the end user. To effectively benefit from policy flexibility, new applications would need to be designed which understand the range of the framework and can express the available options to the users.

So, the next question is then whether or not users need to have such low
level control of the session. The specification framework allows a degree of control significantly more fine grained than is provided by other systems. The examples from other systems include the IBM DiCE access rights, the CCP 'modes of access', and the Touring Machine permissions flag. It is conceivable that users gain little from the ability to control individual variables when the necessary control parameter can be summarized in such simple permissions methods. As a result of this research, I believe that fine grained control may be useful, but systems which would use the framework would probably hide the fine grained details anyway in order to allow users to initiate sessions more easily. This problem is less a technical issue and more a human factors issue.

From a developers point of view, policy flexibility implemented in a software tool can aid in the development of new conferencing applications. Once a developer has determined a set of policies to be made available to users, an application could be built to use a subset of the framework's range. Although this does not provide the full flexibility to users, it does save the cost of re-implementing policy mechanisms for each new application.

In conclusion, the specification framework and policy flexibility as an idea would be useful given that a few issues were resolved. If a human factors study showed that users desired a general system with fine grained session control, then the specification framework could be an effective part of a session management protocol. Otherwise, if a developer determines that users want a subset of the possible policies which hides the fine grained details, applications can be built on top of the framework.

VII.3 Consistency Policies and Conference Scalability

Flexibility in consistency policies brings up separate issues which should be examined for their value to potential users. As mentioned above, the main concern driving consistency issues is the handling of conferencing scalability. The scaling up of conferences in terms of participants and area generates several problems, some of which flexible policies can address. [7]

A major difficulty in maintaining state consistency among a large number of participants is the processing and communication time required. Suppose that a conference contained thousands of users. If the system attempted tight consistency, the amount of messaging required for every simple state change would be enormous. Some kind of state lock would need to be maintained and the system would try to keep everything synchronized. A single change initiator
would need to contact every other participant, wait for them to process the message, and then collect thousands of replies. The ability to specify consistency policy at the level of individual variables allows the user to determine what parts of the state are important enough to incur such a cost. Also, the processing necessary for a single change can be reduced by choosing tight control for only the subset of the members who are concerned with the change. So, designating looser requirements for certain members in a conference can reduce the time necessary to execute an operation. In these ways, the specification framework is effective in making it easier to maintain state consistency over large conferences.

On the other hand, the algorithm that the framework is based on is not as useful in handling some other types of conference problems. Two serious types of failures which become more common in large conferences are message loss and network outages. Some systems devote substantial effort to synchronizing their protocols and recovering from these problems. The framework currently does not have synchronization or recovery procedures. Also, as sessions get large and participants become widely spread out, it becomes more probable that several session requests will be initiated at the same time causing contention. The typical way this is resolved is through some kind of negotiation process. However, the agreement algorithm makes no allowance for this and hence, the framework does not support negotiation policies. To be realistic, a more practical implementation of the framework would need to include procedures and syntax to handle the failure and contention problems.

It is important to note that the main trade-off here is between consistency "tightness" and processing and communication cost. Since the user decides how he wishes to take this trade-off for a particular session, he must then be given some outline of what the costs are as a function of the size of the conference he initiates and the policies he chooses. So, a system which implements the policy flexibility should be accompanied with a general formula which accounts for the major cost factors and presents the trade-off to the user in an easily understandable form. Research into such a formula was beyond the scope of this thesis.

In summary, the concept of flexible consistency policies has clear value to users who wish to participate in large scale conferences. The handling of consistency requirements for each individual participant and for subsets of the session state gives users a much greater range of options than is currently available. However, the usefulness is contingent on two things: the
implementation's solution of failure modes and the presentation of the tradeoffs in such a way that a user can effectively choose from the available options.
VIII Future Work

VIII.1 Specification Language

The specification framework is an abstract model of how sessions can be described. The set of C++ classes used in this thesis is just one possible instantiation of the framework. Another possible instantiation could be a language. The grammar of such a language would be designed to specify the objects in the framework.

To make a specification language useful, a compiler could be written to take in the text description of the session and generate the code to be used by an SMO. In this way, software developers could more easily create a new SMO which runs flexible policies. Also, a standard compiler would allow different developers to match session policy descriptions in interoperable software.

VIII.2 Session Negotiation

The Engine has no mechanism for allowing users to negotiate among themselves regarding the specifics of a new session. A user who receives a request to join a new session can only agree or disagree to participating. In order to propose an alternative session the user must wait for the first to fail and then propose another one. Some extra protocol steps and messages could be added to enable such negotiation. This would reduce extra communication if disagreement on session specifics occurs often. However, it is not known whether negotiation would be necessary in enough instances to warrant the extra protocol logic.

VIII.3 Sufficiency

The specification framework is built on two basic sets of primitives: the allowed operations and the various parameter choices available in the rules. These primitives were chosen based on experience with teleconferencing systems and a set conferencing functions I decided were needed. From the choices of primitives an automatic question arises. Is the set of primitives sufficient to express the full range of desired policies?

The initial form of the framework grew out of a simple set of conferencing functions. During implementation and attempted translations of current systems, additional requirements were revealed. For example, the concept of the owners of a list of variables was an option added only after it became an issue with the Touring Machine translation. Further research could be done on the sufficiency of the current version of the framework. Translating other conferencing systems
into the framework could provide some well-defined bounds on the expressiveness of the framework. At this time, the framework is considered sufficient for the purposes of this thesis but it is acknowledged that further investigation is needed.

VIII.4 Communication Characteristics

In order to make effective use of the Engine, some method should exist to characterize the communication being used for control messaging. Specifically, the Engine needs to be informed of the type of mechanism employed.

As noted above, the current Engine implementation assumes a certain type of communication mechanism. The procedures it runs to maintain the consistency conditions are dependent on the assumptions which accompany the type. This type, of course, is not the only possible mechanism with which a system might want to use the Engine. Other types mentioned in [1] include an unreliable explicit destination mechanism and a broadcast mechanism. So, future work could involve adding a function to the Engine which, when given a pre-defined mechanism type, would execute the proper set of procedures.

VIII.5 Consistency Policy Formula

In the consistency policy conclusions, I mentioned the need for a formula which would allow a user to make an informed decision regarding his desired consistency policies. Essentially, the software would first need to have some notion of how scaling a conference in terms of users or area would affect processing and communication cost for the particular underlying system being used. The problems which would be seen by the user in a large conference would need to be reduced to measurable values.

If the relevant values could be found and measured, then a formula would need to have some method for relating these values to the consistency policies. How do each of the consistency designation choices for variables and members affect the costs? If this relationship could be quantified, then the tradeoffs would need to be presented to the user in a comprehensible form. Once the user was given a simple set of trade-off options, he could choose the consistency conditions which best suited his needs.

VIII.6 Protocol Extensions

As described above, two significant protocol extensions have been considered without a definitive conclusion on whether they should be included.
For the time being these were left out for the sake of simplicity, but it is acknowledged that a more in-depth look at the tradeoffs involved could provide enough reason to add them.

The basic add, delete, and change operations could be used to change policy as well as session state. Currently, a new policy requires a completely new session. The extensions to the framework would involve additional fields in the Operation object to allow for changes in Rules and consistency designations. Also, some part of the Session state would need to hold Rules for changing policy.

Another extension could provide specification of groups of members. A user may wish to describe a particular set of members for voting purposes or designate particular policies for a group. Such an addition would add a considerable amount of code to the Engine. Additional options would be required within the Rules and some sort of group object would need to be created.
References


Appendix A

Specification Framework

Syntax Key

Framework objects are described in the following form:

Object Name:

{ object components }

Component Explanations

Special conditions

Components can be of the form:

parameter

< Framework object >

[ component, component, ... ] - choice of components

" component name " - special component

A list of objects is denoted with

'+' for one or more occurrences

'*' for zero or more occurrences

Consistency designations are denoted by:

C - critical  D - delta  E - eventual  N - none

W - C, D, E  X - C & E  Y - D & E  Z - C & D

Currently available member group designations:

M - set of all members active in the session

Me - Critical members  Me - Eventual members
Framework Objects

Session:

\{ session name, creation type, "MemberList", "SessionVars" \}

creation type - unanimous, majority, any, none
specifies the set of polled users who need to agree to participate before a session can be instantiated

"MemberList"
This is a special object similar to a Variable. It contains the list of session participants and always has the name MemberList. It is not a normal Variable in that it contains a list of Members rather than Variables. This object is created automatically when the session object is created. It is given default Variable parameters which can be altered with special session object methods.

"SessionVars"
This is a special instantiation of a Variable object. It contains the list of session-owned variables and is always named SessionVars. This object is created automatically when the session object is created. It is given default Variable parameters which can be altered with special session object methods.

Member:

\{ member name, consistency designation, "MemberVars" \}

consistency designation - C, E, X, N

"MemberVars"
This is a special instantiation of a Variable object. It contains the list of variables that this member owns and is always named MemberVars. This object is created automatically when the member object is created. It is given default Variable parameters which can be altered with special member object methods.
Variable:

\{ variable name , variable owner , consistency designation , variable type ,
    [ value , <Variable>+ ] , <Rule>* \}

variable owner - string name of the member who owns the variable
consistency designation - C, E, D, N, W, X, Y, Z
variable type - string, integer, floating, varlist
value - contains a string, integer, or floating point number depending on type

A Variable can be referenced by a string of the form “owner ; full pathname” where
owner is the Variable’s owner and full pathname is a ‘;’ separated list of the
Variables in a list hierarchy above the desired Variable.
Example: “John ; MediaPorts ; VideoPorts ; Camera “
A Variable of type varlist must have at least one Variable in it’s list.
The list of Rules specifies the voting policy for operations on the Variable. A Variable
with no Rules is considered to be unchangeable.
An operation which matches more than one rule will be executed if it passes any of the
matching rules.

Rule:

\{ operation type , value type , value , <Initiator>+ , <Voter>* , voting requirement ,
    voting fraction \}

operation type - add , delete , change
    Add and delete allowed only if associated Variable is of type varlist
    Change not allowed if associated Variable is of type varlist
value type - string, integer, floating, none
    Types string, integer, and floating must match associated Variable type
    If Variable is type varlist, then choice must be none otherwise none indicates
    that the Rule applies to all changes
value - string, integer or floating depending on value type.
    Integer value is 0 for type none.
voting requirement -  agree , unanimous , majority , fraction , any , automatic

The agree choice is used for agreement from the list of voters

Automatic means that the operation needs no vote if initiated by a valid initiator

voting fraction -  fraction of needed voters if requirement choice is fraction.

The Initiator list specifies the members who are allowed to initiate an operation which matches the operation type and value type. A Rule must have at least one Initiator in its list.

The Voter list specifies the members who need to be notified and who need to vote on an operation which matches the operation type and value type. An empty list of Voters must be accompanied by a voting requirement of automatic.

Initiator :

{  initiator type , initiator string  }

initiator type -

  member   -  member name
  group    -  member group limited to [ M , Mc , Me ]
  owner    -  variable owner
  designation -  reference to a Variable whose value is a member name
  anyone   -  any user who makes the request (note that is not limited to session members)

initiator string -  based on initiator type

  If initiator type is anyone, then initiator string is ignored and should be set to “anyone”.

Voter :

{  voter type , voter string  }

voter type -

  member   -  member name
  group    -  member group limited to [ M, Mc, Me ]
  owner    -  variable owner
  designation -  reference to a Variable whose value is a member name
target - member who is targeted by the operation
target owner - owner of variable targeted by the operation
list owners - owners of other variables in the variable list being modified
target list owners - owners of variables in target variable list
voter string - based on voter type

The target of an operation is the Member or Variable being added or deleted from a list.

Operation Set:

\{ session name, set initiator, \langle Operation\rangle+ \}\n
session name - name of target session
set initiator - name of member who initiates the set

An Operation Set groups associated Operations together. All Operations must be part of some Operation Set.
For an Operation Set to take effect on the session state, all of the Operations in the set must pass their corresponding voting policies.

Operation:

\{ variable name, variable owner, operation type, value type, variable keyword, 
  \[ value, \langle Variable\rangle, \langle Member\rangle \] \}

variable name - name of variable being modified
variable owner - owner of variable being modified
operation type - add, change, delete
value type - string, integer, floating, none
variable keyword - full name of variable or member being added or deleted
value - based on value type. Integer value 0 if type is none.
Appendix B

Protocol Translation Example

To test the specification framework, a model was created to show how the framework could express the policies of a real conferencing system. Touring Machine (Version 2) was chosen mainly because of the availability of a detailed session protocol description.

Touring Machine is designed as an architecture which supports multi-media conferencing by hiding low-level details from the conferencing applications. The architecture consists of various software objects which handle session control, resource management, and communication with media devices. On top of the objects is an application programming interface (API) which defines a set of messages which an application can use to create and participate in conferences.

In the Touring Machine model the Engine would be used as a policy tool by the Session Object. See Diagram C1. It is important to note that the policies available to an application are limited by the Touring Machine API. The API would need to be expanded to take advantage of the range of policies which the Engine could handle. Translation of the current API and session model reveals that a session model can be expressed by the specification framework and that the Touring Machine system implicitly contains inflexible policies.

The state of a Touring Machine session consists of clients (participants), media ‘endpoints’, and media ‘connectors’. Each client owns a set of endpoints which represent the ports for transmitting and receiving media streams. A session’s connectors are the logical configurations of the media streams.

The following sections show translations of sample Touring Machine API messages. The most significant translation is of the sessionCreate message. This shows how the Touring Machine session state and policies are expressed in the specification framework. The remaining translations just demonstrate some examples of simple operations.
I. Session Initiation

Touring Machine:

```
( sessionCreate 9876 "bob:app:s1" 3456 (addCon "vCon" "video")
  (addSource "vCon" "bob:app:camera" "joe:app:camera")
  (addSink "vCon" "bob:app:monitor" "joe:app:monitor")
  (setPermission "protected") )
```

This message creates a session with two clients - “bob:app” and “joe:app”. The session is named “bob:app:s1”. The lone media connector, “vCon”, is of type video. Each client has a source and a sink within the connector. The ‘protected’ permission setting designates that only participating clients can initiate changes to the session.

Engine:

**SESSION:**

- `session name = “bob:app:s1”`
- `creation type = unanimous`
- “SessVars” <Variable>
- “MemberList” <Variable>

Touring Machine session initiation voting is always unanimous.

“MemberList” VARIABLE:

- `variable name = “MemberList”`
- `variable owner = “Session”`
- `variable type = special list`
- `variable value = [ Members ]`

**RULES:**

- `{ add ; none ; 0 ; ‘memgroup , M’ ; ‘target’ ; agree }`
- `{ del ; none ; 0 ; ‘memgroup , M’ ; ‘target’ ; agree }`

Adding and removing members from a session can be initiated by anyone in the session and only requires agreement from the affected client.
Members:

“bob:app” MEMBER:

“MemberVars” VARIABLE:

variable name = “MemberVars”
variable owner = “bob:app”
variable type = varlist
variable value = [ bob’s Variables ]

RULES: { add ; none ; 0 ; ‘owner’ ; <novoters> ; automatic }

The owner can add to his own variables automatically.

bob’s Variables:

“camera” VARIABLE

variable type = string
variable value = “unmapped”
A media endpoint is not connected to physical media transport when ‘unmapped’.

RULES: { change ; none ; 0 ; ‘owner’ ; <novoters> ; automatic }

The owner can turn his endpoint on or off at will.

“monitor” VARIABLE

variable type = string
variable value = “unmapped”

RULES: { change ; none ; 0 ; ‘owner’ ; <novoters> ; automatic }

“joe:app” MEMBER: same as bob:app with different name
"SessVars" VARIABLE:
variable name = "SessVars"
variable owner = "Session"
variable type = varlist
variable value = [ Session Variables ]

RULES: < no rules >
There is no capability to add session owned variables beyond those created at initiation.

Session Variables: (all are owned by "Session")

"Initiator" VARIABLE:
variable type = string
variable value = "bob:app"

RULES: <no rules>

"Privacy" VARIABLE:
variable type = string
variable value = "all"

RULES: { change ; none ; 0 ; 'memgroup , M' ; 'memgroup , M' ;
          unanimous }

The Privacy setting refers to who may access information about the session. It can be changed by any member with unanimous consent of all of the members.

"Connectors" VARIABLE:
variable type = varlist
variable value = [ Connectors ]
**RULES:**
{ add ; none ; 0 ; 'memgroup' , M' ; 'target list owners' ;
  unanimous }
{ del ; none ; 0 ; 'memgroup' , M' ; 'target list owners' ;
  unanimous }

Adding and deleting connectors can be initiated by anyone. The voter list here demonstrates the use of the option 'target list owners' where the target list is a connector variable. To add or delete a connector, the owners of all of the variables in the connector must agree.

Connectors:

"vCon" VARIABLE:
  variable type = varlist
  variable value = [ Connector Variables ]

**RULES:** <no rules>
The main variables in a connector are fixed. Additions and deletions are done within the sublists.

Connector Variables:

"type" VARIABLE:
  variable type = string
  variable value = "video"

**RULES:** <no rules>

"Sources" VARIABLE:
  variable type = varlist
  variable value = [ "bob:app.camera" , "joe:app.camera" ]

**RULES:** { add ; none ; 0 ; 'memgroup' , M' ; 'target owner' ; agree }
{ del ; none ; 0 ; 'memgroup' , M' ; 'target owner' ; agree }

This variable contains references to member-owned variables. This
example demonstrates the need for variable references and the need for the ‘target owner’ voting option. Adding or deleting a variable here requires agreement from the owner of the variable being added or deleted.

“Sinks” VARIABLE:

```
variable type = varlist
variable value = [ “bob:app.monitor” , “joe:app.monitor “ ]
```

RULES:  { add ; none ; 0 ; ‘memgroup , M’ ; ‘target owner’ ; agree }
        { del ; none ; 0 ; ‘memgroup , M’ ; ‘target owner’ ; agree }

II. Session Changes

Example 1

Touring Machine:
Client joe:app sends

( sessionChange 123 “bob:app:s1” 3457 (addClient “ted:app”) )

Engine:

OPERATION SET:
  session = “bob:app:s1”
  initiator = “joe:app”

Ops:
OPERATION:
  variable name = “MemberList”
  variable owner = “Session”
  operation type = add
  value type = none
  variable keyword = “ted:app”
  newMem:
    “ted:app” MEMBER
    <no variables>
Example 2

Touring Machine:

Client ted:app sends

( sessionChange 124 "bob:app:s1" 3458
  (addSource "vCon" "ted:app:camera") )

Engine:

OPERATION SET:

  session = "bob:app:s1"
  initiator = "ted:app"

Ops:

OPERATION:

  variable name = "Connectors.vCon.Sources"
  variable owner = "Session"
  operation type = add
  value type = none
  variable keyword = "ted:app;camera"
  value = 0

OPERATION:

  variable name = "MemberVars"
  variable owner = "ted:app"
  operation type = add
  value type = none
  variable keyword = "ted:app;camera"

newVar:

  "camera" VARIABLE
  <same as other cameras>

This session change demonstrates the need for variable pathnames to reference variables which are nested. Also, note that to add the endpoint to the connector, one operation adds the reference to the connector and the other adds the actual variable to the list of Member variables in ted:app.
Example 3

Touring Machine:
Client ted:app sends
( endpointMap 125 “bob:app:s1” 3458 “camera” )

Engine:

OPERATION SET:

session = “bob:app:s1”
initiator = “ted:app”

Ops:

OPERATION:

variable name = “camera”
variable owner = “ted:app”
operation type = change
value type = string
value = “mapped”
Appendix C

Flow of Control: Centralized Model

General information:
Communication is accomplished through messages over TCP sockets. The message formats are described by an ASN.1 specification. Packing and unpacking of messages are done with routines generated by the snacc ASN.1 compiler. Message formatting and transportation is handled by the SMO and each member's CommObj.

Session Initiation
Assumption: All of the potential participants have established connections to the SMO.

Stage 1: Request
The requesting member builds a Session object. Members, Variables, and Rules are created with their associated constructor functions and then added into the Session object.

The Session is converted into an ASN.1 structure. { Session::Parse } 
A session create request is sent to the SMO. { CommObj::SessionCreate }

Stage2: Policy Check and Poll
The SMO receives request and dispatches to the Engine.
{ Engine::CreateSession }

If Session has duplicate name return an error to user. { member_error }
If Session needs no vote then send done message to user. { member_ack }
If Session needs vote then send polling status message to user.
{ member_ack }

Based on the session create type, the Engine determines which members need to be polled for a vote on creating this session. { Session::CreatePoll }
For each member who needs to be polled, a message is sent.
{ member_crpoll }
Stage 3: User Notification and Reply

Member's CommObj receives message. The message is translated into a session creation request and sent to the user. { session_cr_poll }

User sends reply through CommObj. { CommObj::CreateVote }

Stage 4: Vote Collection and Result Notification

SMO receives user reply and dispatches it to the Engine. { Engine::CreateAnswer }

Engine sends answer to appropriate session object. { Session::CreateVote }
If answer is no, Session object checks to see if vote has failed.
If vote has failed, abort messages are sent. { Session::CreateAbort }
For each member in the session, message sent through SMO.
{ Member::CreateAbort, member_newabort }
If answer is 'yes' Session object checks to see if all responses have come in.
If all responses are in and vote has passed, send commit messages.
{ Session::CreateCommit }
For each member who agreed to join, commit sent through SMO.
{ Member::CreateCommit, member_newcommit }
If still waiting for responses, or a 'no' answer does not cause the vote to fail, Engine continues to wait for responses.
If timeout occurs, Engine assumes all no-response votes to be 'no' answers and checks if vote has passed or failed.

Session Change

Stage 1: Request

The initiating member builds an Operation Set object consisting. For each desired change, an Operation object is created and added to the Operation Set. The Operation Set is converted into an ASN.1 structure. { OpSet::Parse }
An execute operation request is sent to the SMO. { CommObj::OpRequest }

Stage 2: Policy Check and Poll

The SMO receives the request and dispatches to the Engine.
{ Engine::ExecuteOp }
If session does not exist, send error message to initiator. { member_error }
For each variable being changed:
Get variable from the session. { Session::GetVar }
Execute operation on the variable. \{ \text{Variable::ExecuteOp} \}

If operation type is invalid, send error message to initiator.
\{ \text{member\_error} \}

If initiator is invalid, send message to initiator. \{ \text{member\_ack} \}

If operation needs no vote, commit changes to the session and notify initiator. \{ \text{member\_ack}, \text{Session::OpCommit} \}

If operation needs a vote send polling message to each required member who has not previously been polled for this OpSet.
\{ \text{Session::PollMember, Session::PollGroup, Session::PollDesignated} \}
\{ \text{member\_poll} \}

Stage 3: User Notification and Reply

Member's CommObj receives message. The message is translated into a session change request and sent to the user. \{ \text{session\_oppoll} \}

User sends reply through the CommObj. \{ \text{CommObj::OpVote} \}

Stage 4: Vote Collection and Result Notification

SMO receives user reply and dispatches it to the Engine. \{ \text{Engine::Poll\_Vote} \}

For each variable in being changed:

Get variable from Session. \{ \text{Session::GetVar} \}

Enter vote in to the Variable. \{ \text{Variable::EnterVote} \}

Enter vote in each Rule. \{ \text{Rule::EnterVote} \}

If any rule passes, then the vote passes for this variable.

If vote fails for any variable, entire OpSet fails. Send abort message to all members.

If OpSet included operation to add member, send abort message to target member.
\{ \text{Session::OpAbort, Member::OpAbort, member\_opabort} \}

If vote passes for all variables, send commit message to all members.
\{ \text{Session::OpCommit}, \text{Member::OpCommit}, member\_opcommit \}

If OpSet included operation to add member, send create commit message to target member. \{ \text{Member::CreateCommit, member\_newcommit} \}

Commit changes to the session state. \{ \text{Variable::Opcommit} \}
If vote is still waiting for any variable, Engine continues to wait for more responses.

If timeout occurs, all no-reply responses are treated as 'no' answers. Votes are entered accordingly and result determination follows as above.
Appendix D

Engine-SMO API: Centralized Model

Key:
Procedure arguments:

- ALL CAPS - C++ class instance
- italics - character string
- plain text - enumerated type

Engine Procedures:
(the SMO calls each procedure)

Engine::CreateSession (SESSION, initiator)
Create a new session described by SESSION. Check creation type and send out all necessary requests.

Engine::ExecuteOp (OPSET)
Execute the operation set described by OPSET. Check the Session for the relevant rules and send out all necessary requests.

Engine::MemberExit (session name, member name)
"member name" has unilaterally left "session name".

Engine::MemberLeave (member name)
"member name" has disconnected from the SMO.

Engine::CreateAnswer (session name, member name, answer)
Enter "answer" as "member name's" vote regarding the creation of "session name".

Engine::Poll_Vote (OPSET, member name, answer)
Enter "answer" as "member name's" vote regarding the operation described by OPSET.
Engine::Time_Ping()

    Check on timeouts for operations which are waiting for user replies.

SMO Procedures

(these procedures are called by class functions within the Session) Note that the current SMO does not set up real conferences. So, it’s main duty is message passing. In a full implementation, the Engine would call procedures to inform the SMO of state changes after it finished passing messages to users.

member_cpoll (SESSION , member name )

    Send a message to “member name” to request a vote on the creation of SESSION.

member_newcommit ( SESSION , member name )

    Send a message to inform “member name” that SESSION has been created.

member_newabort ( session name , member name )

    Send a message to inform “member name” that the creation of “session name” has failed.

member_poll ( OPSET, member name )

    Send a message to “member name” to request a vote on OPSET.

member_opcommit ( OPSET , member name )

    Send a message to inform “member name” that OPSET has successfully completed.

member_opabort ( OPSET , member name )

    Send a message to inform “member name” that OPSET has failed.

member_sesskill ( session name , member name )

    Send a message to inform “member name” that “session name” has been killed.
member_error (error type, error object, description, member name)
Send an error message of type “error type” to “member name”. “error object”
and “description” specify the source of the error. Used only to send error
messages back to users initiating operations.

member_ack (ack type, ack object, member name)
Send an acknowledgement of type “ack type” to “member name”. “ack object”
specifies the source of the message. Used only to send acknowledgements
to users initiating operations.
Appendix E

Flow of Control: Distributed Model

General Information

Communication is accomplished through messages over TCP sockets. The message formats are described by an ASN.1 specification. Packing and unpacking of messages are done with routines generated by the snacc ASN.1 compiler. The CommObj object used here is a modified version of the one used in the centralized model, and one is created and maintained by each Engine. Sockets between users are created on a need basis and remain connected for the duration of a member’s participation in a session.

In this model, the Engine is theoretically run by an SMO. The implementation of the project, however, contains simple client software which runs individual Engines. So, steps which involve communication through the SMO are omitted in this control flow description. For the purposes of testing the Engine, these steps are redundant. The proposed API to the SMO is still retained.

Session Initiation

Assumption: After each user’s Engine has been created, potential participants are registered with the Engine. 

Engine::RegisterPeer

Stage 1: Request

The requesting member builds a Session object. Members, Variables, and Rules are created with their associated constructor functions and then added into the Session object.

The session create request is sent to the Engine. 

Engine::CreateSession

If Session has no members besides initiator ‘done’ status returned to initiator. The Engine opens a connection to each proposed member.

Session::ConnectMembers , CommObj::Connect

Creation request message is sent to each member.

Session::CreatePoll , CommObj::SessionCreate

Stage 2: User Notification and Reply

User's CommObj receives message and sends session creation request message to the user. 

session_crpoll
User sends reply through its Engine.
    { Engine::CreateAnswer, CommObj::CreateVote }

Stage 3: Vote Collection and Result Determination
Initiating member’s CommObj receives reply and dispatches message to Engine.  { Engine::CreateVote }
Engine finds appropriate Session and enters vote.  { Session::CreateVote }
If vote fails, send abort messages and notify initiator.
    { Session::CreateAbort, CommObj::CreateAbort, session_crfail }
If all votes are in and vote passes, send commit messages and notify initiator.
    { Session::CreateCommit, CommObj::CreateCommit, session_crpass }
If vote is ‘no’ remove member from the Session.

Engine waits for all votes to come in.  If a timeout occurs, no response is taken as a ‘no’ answer and the vote result is determined as above.

Session Change

Stage 1: Request
User creates Operation Set object and sends request to the Engine.
    { Engine::ExecuteOp }
For each operation in the Operation Set, Engine gets the Variable and executes the Operation.  { Session::GetVar, Variable::ExecuteOp }
If any execution returns invalid for the variable, an error is returned to initiator.
If none of the operations needs a vote, then Operation Set is complete.
Engine checks to see if any operation affects the delta state.
If not, announce messages are set and operation changes are committed.
    { Session::SendAnnounce, CommObj::OpAnnounce, Session::CommitOp }
If so, lock request is sent to all critical members.
    { Session::SendLock, CommObj::Lock }
Receiving Engines, determine current lock status and reply with either ‘yes’ or ‘busy’.  { CommObj::LockReply }
If lock succeeds, commit messages are sent, and initiator is notified.
    { Session::SendCommit, CommObj::OpCommit, op_commit }
If lock fails, release messages are sent, and initiator is notified.

\{ Session::SendRelease, CommObj::Release, op_abort \}

If an operation needs a vote, then polling messages are sent. One message per Operation Set is sent instead of one per Operation.

\{ Session::PollMember, CommObj::Poll \}

If operation affects the delta state, poll_lock messages are sent to appropriate critical members instead of normal poll messages.

\{ CommObj::PollLock \}

Stage 2: User Notification and Reply
User's CommObj receives message and dispatches to Engine.

If message is a lock request, Engine checks to see if lock is currently busy.
If so, then 'busy' reply is sent. \{ CommObj::OpAnswer \}
If message is not a lock, or the lock is available, then user is notified of request. \{ op_poll \}

User sends reply to initiator through the Engine.
\{ Engine::PollAnswer, CommObj::OpAnswer \}

Stage 3: Vote Collection and Result Determination
Initiator CommObj receives reply and dispatches to the Engine.
If the reply is from a critical member and is 'busy' then operation set fails.
Abort messages are sent for entire Operation Set and initiator is notified.
\{ Session::AbortOp, Session::SendRelease, op_abort \}

Engine enters vote in appropriate Variables and checks to see if votes pass.
If the vote for any operation in the set fails, the entire set fails. Abort messages are sent and initiator is notified. \{ Session::AbortOp, op_abort \}
If the vote on any operation is still waiting, the Engine continues to wait for replies.
If the vote for every operation in the set passes, the entire operation passes.
If any operation in the set affects the delta state, commit messages are sent. \{ Session::SendCommit \}
If no operation in the set affects the delta state, announce messages are sent. \{ Session::SendAnnounce \}
Stage 4: Consistency Maintenance

User's CommObj receives announce or commit message and dispatches to Engine.

If user is a critical member and message is a commit, then changes are committed immediately, and user is notified.

{ Session::CommitOp, op_commit }

If user is not an eventual member, or the operation set does not affect the eventual state, then operation set is committed immediately, and user is notified.

Otherwise, if user is an eventual member, Engine checks if operation set affects the eventual state.

If so, operation set is buffered in order of timestamp. Operation set is committed after it has been buffered for the buffer time.

If not, then operation set is committed immediately and user is notified.
Appendix F

Engine-CommObj API & Engine-SMO API: Distributed Model

Key:

Procedure arguments:
- ALL CAPS - C++ class instance
- *italics* - character string
- plain text - enumerated type

USER - the user who is running the instances of the Engine, SMO and Comm Obj

Engine Procedures

The Engine procedures are separated into two groups. The first group contains those procedures that the SMO calls mainly to handle requests from USER. The second group contains those procedures called by the CommObj to process messages received from other Engines.

Group 1:

**Engine::CreateSession ( SESSION )**

Create a session as described by SESSION. The initiator of the session is USER. Send out all necessary requests.

**Engine::Create Answer ( session name, answer, member name )**

Reply to “member name” regarding “member name’s” session creation request. “answer” is USER’s vote on the creation of “session name”.

**Engine::ExecuteOp ( OPSET )**

Execute the operation described by OPSET. The initiator of the operation is USER. Send out all necessary requests.

**Engine::PollAnswer ( OPSET, answer )**

Reply to the initiator of OPSET. “answer” is USER’s vote on the execution of OPSET.
Engine::ExitSession (session name)
Remove USER from “session name”.

Engine::RegisterPeer(peer name, peer address)
Register the existence of “peer name” as a potential conference participant.
“peer address” is the address which the CommObj can use to contact “peer name”.

Engine::Time_Ping()
Check all timeouts on current operations. Also, tell CommObj to check whether incoming messages are waiting to be processed.

Group 2:

Engine::CreateVote (session name, member name, answer)
Enter a vote from “member name”. “answer” is “member name’s” vote on the creation of “session name”.

Engine::CreateCommit (SESSION)
Establish SESSION as successfully created. Notify USER.

Engine::CreateAbort (session name)
Notify USER that “session name” creation has failed.

Engine::OpPoll (OPSET)
Request a vote from USER on the execution of OPSET.

Engine::OpPollLock (OPSET)
Check if lock is currently held on the critical state of the session named in OPSET. If not, set the lock and request a vote from USER on the execution of OPSET.
If lock is already held, return ‘busy’ vote.

Engine::Lock (opset id, member name)
Check if lock is currently held on the critical state. If not, set the lock. If lock is already held, return ‘busy’ reply.

Engine::LockReply (opset id, member name, answer)
Enter reply from “member name”. “answer” is “member name’s” reply to a lock request sent by this Engine.

Engine::PollVote (opset id, member name, answer)
Enter a vote from “member name”. “answer” is “member name’s” vote on the execution of the opset designated by “opset id”.

81
Engine::Release ( OPSET )
Release the lock on the critical state associated with OPSET.

Engine::OpAnnounce ( OPSET )
Enter changes described by OPSET. Buffer OPSET if necessary.

Engine::OpCommit ( OPSET )
Make changes described by OPSET. If USER is a critical member of the session, release lock on session’s critical state.

Engine::UserExit ( session name, member name )
Remove “member name” from “session name”. Take necessary actions to clean up the session state.

SMO Procedures
Since the SMO is only a shell in this implementation, the procedures called by the Engine are only required to pass messages to USER. As noted in Appendix D, a full implementation would include an SMO which executed other conferencing functions. In such an implementation the following procedures would not only inform USER, but would also make the actual changes to the physical session, i.e. media devices would be turned on or off, etc.

session_crpoll ( SESSION, member name )
Request a vote from USER on the creation of SESSION. “member name” is the initiator of SESSION.

session_crcommit ( SESSION )
Inform USER that SESSION has been successfully created.

session_crfail ( session name )
Inform USER that the creation of “session name” has failed.

session_crpass ( session name )
Inform USER that a vote on the creation of “session name” has passed. Used only for sessions initiated by USER.

op_poll ( OPSET )
Request a vote from USER on the execution of OPSET.

op_commit ( OPSET )
Inform USER that OPSET has successfully completed.
op_abort ( OPSET , reason )
Inform USER that OPSET failed to complete. “reason” describes the source of the failure.

member_exit ( member name )
Inform USER that “member name” is no longer connected.

CommObj Procedures
These procedures are mostly called by class functions within the Session and Member classes.

CommObj::RegisterPeer ( user name , address )
Register “user name” as potential conference participant. Use “address” to connect to “user name”

CommObj::Connect ( user name )
Connect to “user name”

CommObj::Disconnect ( user name )
Close connection to “user name”.

CommObj::MsgDispatch ( ENGINE )
Check whether incoming messages are waiting to be processed. Dispatch any waiting messages to appropriate Engine procedure.

CommObj::SessionCreate (SESSION , member name )
Send message to “member name” requesting a vote on the creation of SESSION.

CommObj::CreateVote ( session name , answer , member name )
Send vote message to “member name”. “answer” is USER’s vote on the creation of “session name”

CommObj::CreateAbort ( session name , member name )
Send message to “member name” that the creation of “session name” has failed.

CommObj::CreateCommit ( SESSION , member name )
Send message to “member name” that SESSION has been successfully created.

CommObj::OpPoll ( OPSET , member name )
Send message to “member name” requesting a vote on the execution of OPSET.
CommObj::OpPollLock ( OPSET, member name)
Send message to “member name” requesting a lock on the critical state of the
session named in OPSET and a vote on the execution of OPSET.

CommObj::Lock ( session name, member name)
Send message to “member name” requesting a lock in the critical state of
“session name”

CommObj::LockReply ( session name, member name, answer)
Send message to “member name” regarding a lock on the critical state of
“session name”. “answer” is the Engine’s reply to a lock requested by
“member name”.

CommObj::OpAnswer ( OPSET, answer, member name)
Send vote reply message to “member name”. “answer” is USER’s vote on the
execution of OPSET.

CommObj::Release ( OPSET, member name)
Send message to “member name” to release the critical state lock associated
with OPSET.

CommObj::OpAnnounce ( OPSET, member name)
Send message to “member name” to announce the successful vote on the
changes in OPSET.

CommObj::OpCommit ( OPSET, member name)
Send message to “member name” to announce the successful vote on the
changes in OPSET and to release the critical state lock associated with
OPSET.

CommObj::ExitSession ( session name, member name)
Send message to “member name” that USER is has exited from “session
name”