ABSTRACT

PRASHANT GUPTA. A Floor Control Protocol for SIP-based multimedia conferences (Under the direction of Dr. Douglas S. Reeves).

The purpose of this research is to define a protocol to regulate resources among participants in SIP-based centralized multimedia conferences. Centralized conferences are typical of contemporary conferencing architectures. SIP is emerging as the signaling protocol of choice for multimedia multiparty sessions. An important problem that needs to be addressed, in such sessions, is that of controlling and ordering access to multimedia resources among participants. This is also known as floor control. There is, to the best of our knowledge, no standardized protocol that addresses the problem of floor control, though there is one other competing proposal in the pipeline. The work in this thesis proposes a set of primitives that solve the above problem, for a variety of situations. We present a comparison of the approach taken in this thesis and the existing proposal. We have developed software that realizes a subset of the primitives as a preliminary proof of concept of the proposed protocol.

A FLOOR CONTROL PROTOCOL FOR SIP-BASED MULTIMEDIA CONFERENCES

by

Prashant Gupta

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

DEPARTMENT OF COMPUTER SCIENCE

Raleigh

2002

APPROVED BY:

Dr. Mladen A. Vouk	Dr. Peng Ning

Dr. Douglas S. Reeves
(Chair of Advisory Committee)

To my parents,
Chhaya and **Ashok Gupta**,
for being my guiding star;

my sisters,
Prerna and **Yamini**,
their happiness means the world to me;

and **Kalpana Jogi**,
my *Joie de Vivre*.

Biography

Prashant Gupta was born in Madras (now Chennai), India, in 1977. After completing his high school in Chennai, he received his Bachelor of Engineering (Hons.) degree in Computer Science from Birla Institute of Technology and Science (BITS), Pilani, Rajasthan, India, in 1999. After that he worked for Aditi Technologies, Bangalore, India as a software developer for about a year.

Since August 2000, till the time of this writing, he has been working full time towards his MS degree in Computer Networking at the Computer Science Department at North Carolina State University, Raleigh, North Carolina, USA.

Acknowledgements

I am grateful to my advisor Dr. Douglas S. Reeves for his support and encouragement. I thank Dr. Mladen A. Vouk and Dr. Peng Ning for consenting to be on my thesis committee. I would like to acknowledge Dr. Jonathan Rosenberg for giving me valuable pointers on work to be done in the area of SIP.

A special mention of Karthik Sundaramoorthy and Vijay Iyer for reaffirming their friendship in countless selfless ways. Last but not the least, I acknowledge Akshay, Anneke, Girish, Harsha, Harshdeep, Harshit, Jessica, Matthew, Prabhas, Ravi, Sameer, Sandeep, Satyajit, Vignesh and Yeshwant, for making my stay at NC State a memorable one.

Table of Contents

Acronyms

3GPP	Third Generation Partnership Project
IETF	Internet Engineering Task Force
SIP	Session Initiation Protocol
MMUSIC	Multiparty Multimedia Session Control
RFC	Request For Comments
HTTP	Hyper Text Transport Protocol
IP	Internet Protocol
SMTP	Simple Mail Transfer Protocol
SDP	Session Description Protocol
SAP	Session Announcement Protocol
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XML	Extensible Markup Language
NIST	National Institute of Standards and Technology
RTP	Real-time Transport Protocol
RTCP	RTP Control Protocol
IGMP	Internet Group Management Protocol

List Of Figures

List Of Tables

1 Introduction

This chapter provides some background information on the contents of this thesis. It starts off by discussing the Session Initiation Protocol [RFC 3261] and its impact on the current telecommunication industry followed by a brief outline of a related protocol suite, the H.323. We then present a discussion on multimedia conferencing. An overview of SIP vs. H.323 deployment is presented next. The area of work in this thesis is given in the focus section. This is followed by a list of common terminology used. The next section gives an overview of the contributions made in the chosen area by this thesis. Finally the overall structure of the thesis is presented.

1.1 The Session Initiation Protocol

Work on a signaling protocol started in 1997 in the MMUSIC working group of the IETF. This work resulted in the development of SIP in 1999 as detailed in [RFC 2543]. It is a generic and flexible protocol capable of satisfying diverse signaling needs. Since 1999, it has undergone significant changes and is now published as [RFC 3261].

In the past few years, there has been considerable interest in the emerging Session Initiation Protocol (SIP). It has been accepted as the protocol of choice for signaling multimedia multiparty sessions in IP-based networks. It is an application layer control protocol that is used to setup, modify and terminate sessions. Voice over IP, multimedia conferencing and mobile multimedia services are just a few of the areas where SIP has made significant impact.

In the enterprise domain, SIP phones have become commonplace with the move towards converged networks. Many long distance telecom providers are adopting SIP within their core network because of its significant cost advantage as well as flexibility. SIP offers an easy way to create and deploy new services such as unified messaging, presence based applications, etc.

Another important domain that SIP is now establishing its importance in, is the wireless domain. It has been chosen as the signaling protocol for

future 3G networks by the 3GPP initiative [TS 23-228]. This architecture moves the signaling functions from the network layer into the application layer, leading the way to an all-IP wireless network, as envisioned by the 3GPP.

A related protocol quite of interest is the H.323 standard that is widely used for Voice over IP and multimedia conferencing.

1.2 H.323

H.323 is an umbrella standard that provides a well-defined system architecture and implementation guidelines that cover the entire call set-up, call control, and the media used in a call. The specifications are defined by the ITU (International Communications Union). H.323 was initially aimed at multimedia communications on LANs, has now been reworked to provide for services over WANs. Some of the protocols that are included in this specification are H.235 (authentication), H.225.0 (Call signaling protocols and media stream packetization), H.245 (Control protocol) and T.120 (data conferencing). More information on H.323 may be found at [ITU H323].

1.3 Multiparty Multimedia Conferencing

An application that has been a focus of tremendous interest, in terms of commercial potential, is conferencing. The term conference may be defined as an interaction among an arbitrary number of participants, using an arbitrary set of resources, for an arbitrary length of time. Any of the above parameters may change during the lifetime of a conference. Conferences may be either centralized or distributed. They may also be prearranged or ad-hoc.

Current conferencing architectures are predominantly centralized and are at an early stage of development. They use proprietary solutions leading to vendor dependence and non-interoperability among available products. The IETF multi-media toolkit, of which SIP is a component, attempts to solve this problem by providing standardized protocols for conferencing.

1.4 SIP vs. H.323 in Conferences

The wide deployment of H.323 is primarily due to its initial design requirement of interoperability with the PSTN. Other features - such as support for audio/video/data conferencing, remote camera control and lip synchronization, have lead to an available and working solution, making H.323 more attractive for deployment.

Although current deployment figures of H.323 based conferencing systems are overwhelmingly in favor of H.323, there is a definite move towards supporting SIP-based conferencing. The first SIP-based conferencing products are available in the market today (e.g. SiPEAK available at www.ubiquity.net).

Factors that have hindered deployment of SIP in the telecommunication infrastructure include fluid standards, non-backward compatibility and no inbuilt support for PSTN interoperability.

The deployment of SIP is expected to take off with large scale deployment of end-devices that are SIP enabled. Support for SIP has been inbuilt into Microsoft Messenger XP that allows point-to-point desktop calls. Other features of SIP that are not currently addressed by H.323 are Presence and Unified Messaging.

According to research findings at IEC ([IEC REP-01]), a well-known market research organization, in a survey of all major telcos 52.4% named SIP as the protocol that would be the most important for deploying services in near-future to future networks, while 21.4% named the well established H.323. These findings are supported by articles on the web site of IMTC ([IMTC]) a non-profit organization of about 100 companies to further the growth of and interoperability between conferencing solutions. Initiatives such as the ViDe [VIDE] also support this view.

1.5 Focus

The work contained in this thesis is in the area of centralized conferencing using SIP. Some key functions that need to be addressed by a conferencing protocol suite are -

1. **Conference Discovery/Announcement** – This aspect deals with distribution of conference information to potential participants. This requires that a conference be described in a standard manner and this is achieved by SDP [RFC 2327]. SDP is a format that defines conference parameters and the constituent media streams. SDP bodies are distributed, as attachments, by other protocols such as SAP [RFC 2974] to enable dynamic conferences. SAP is a protocol used to announce conferences primarily on multicast networks. Static means such as making the SDP body available on a web page may also be used.

2. **Conference Setup** – This facet of the conference deals with set up, modification and teardown of the conference. Important aspects are the exchange of capabilities and session management. SIP [RFC 3261] is used to handle both these aspects. It uses SDP [RFC 2327] to describe the session capabilities.

3. **Membership Management** – This deals with the issues of defining and implementing conference access rights of participants. Maintaining allowed members list, identity authentication, etc. are some of the tasks involved. The suite doesn't currently address this function and hence its implementation is largely proprietary.

4. **Media Management** – deals with managing media specific aspects such as encoding schemes, suitable transport protocol, etc. To handle encoding, there are a large number of schemes available that are specific to the nature of the media. To handle the transport, protocols such as RTP/RTCP [RFC 1889], UDP [RFC 768], etc. are used.

5. **Resource Management** - Once the conference is setup an important function to be performed is that of regulating the resources among the various participants. This is also called floor control.

   The area of resource conflict management is the focus of this thesis. Activities that form part of floor control include granting/revoking access to a shared resource by a moderator, requesting access to a resource by a user, etc. Currently there are no standards defined for floor control. Work is in-progress in this area in the IETF with one personal contribution defined in [ID CONTROL-02].

1.6 Terminology and Conventions used

Terms that are used commonly in this text are defined below. Certain terms have additional meaning in a specific context; these are elaborated when encountered. Definitions are also given in the section where they are first used.

Resource – any media stream or control stream.

Conference - an interaction among an arbitrary number of users, using an arbitrary set of resources, for an arbitrary length of time.

Participant - an entity taking part in a conference.

Moderator - an entity in a conference that owns resources and regulates them among the participants.

User Agent – is a logical entity that represents a participant. In the context of our discussion, this is typically a piece of software.

Conference Server – is a generic term for an entity that acts as the focal point of a conference. In this text, it functions as a signaling server and a media server.

A typical conference, with the defined entities, is shown in Fig. 1.1

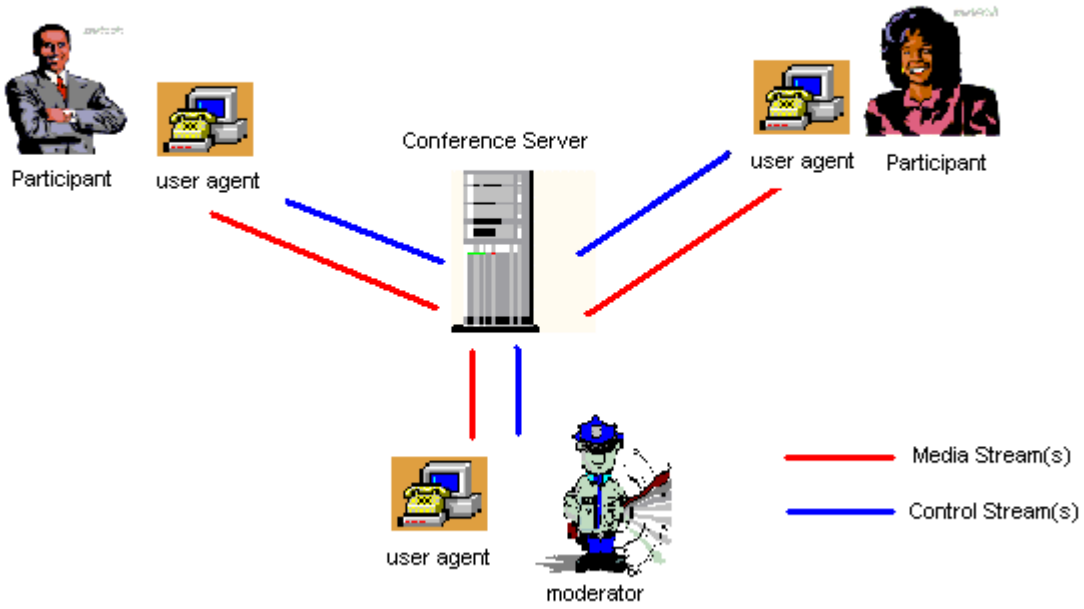


Figure 1.1 – Entities In A Centralized Conference

1.7 Contribution

The contribution of this thesis, in the area of floor control, has been to propose a new protocol for floor control. This proposal is parallel to the one outlined in [ID CONTROL-02]. The protocol proposed in this thesis, the author believes, adds significant enhancements to the functionality given by [ID CONTROL-02]. These enhancements may be classified into three major

areas - new functionality, clearer and more elaborate data structures and support for a wider variety of floor control needs.

In the area of new functionality, new participant requests to *yield* and *cancel requests* are introduced. Moderators may *freeze* floors to prevent any further requests for a floor. It is now possible to specify per user access permissions for each floor. Requests to retrieve floor information have been defined.

There are several changes in the structure definitions of floors, holdings and queues. In floor definitions, parameters such as pre-emptibility and user-permissions are introduced. For floor related state, a differentiation has been made between a queue specifying the order of access and a queue representing incoming participant requests. Queue information has been classified as queue parameters and queue state. A max-queue-size parameter has been introduced to control queue size.

The protocol supports a wider variety of policies, a discussion on these policies is given in Section 4.1. Other enhancements include creating floors only for required resources and leaving the rest unmoderated.

1.8 Structure of thesis

In the following chapter a review of the IETF multimedia toolkit is presented. The protocols in the toolkit relevant to this thesis are briefly discussed highlighting specific useful features. Chapter 3 elaborates on the floor control function of a conference. It surveys the current conferencing architectures supported by SIP and details the centralized conferencing model. Chapter 4 outlines the working of the proposed floor control protocol and elicits the elements of protocol design. Chapter 5 provides the architectural details of the components implemented to serve as proof of concept of protocol working. Chapter 6 summarizes the work done and discusses future work, concluding the thesis.

2 Background

An architectural view, of the protocols comprising the IETF Multimedia Conferencing Architecture ([ID CONFARCH-03]), is shown in Fig. 2.1. Descriptions of the protocols are provided in the following subsections.

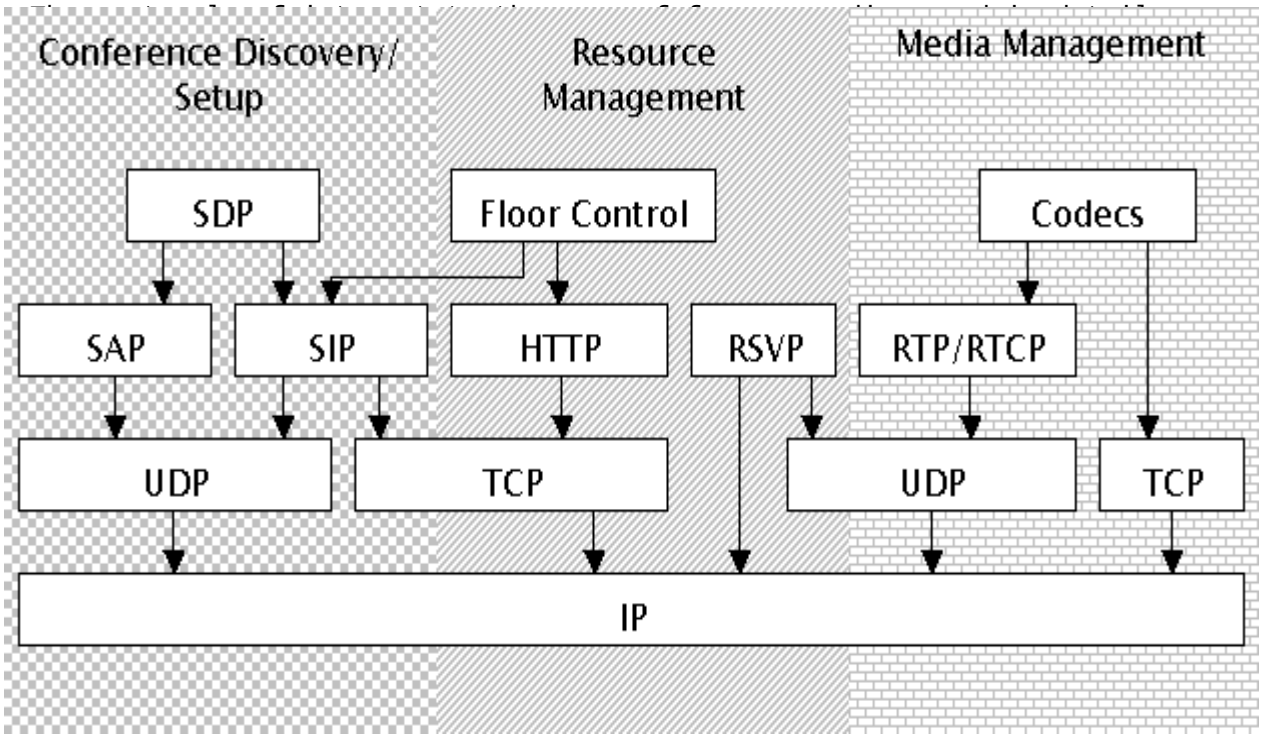


Figure 2.1 – IETF Multimedia Conferencing Stack ([ID CONFARCH-03])

2.1 IP

The Internet Protocol offers a connectionless, best-effort datagram delivery service. It provides a physical network independent abstraction to the transport layers. Important functions provided by this protocol are addressing, routing and fragmentation/reassembly. The IP Specifications can be found in [RFC 791].

2.2 TCP

The Transport Control Protocol offers a reliable, connection-oriented, stream-based service. Other important services offered by this protocol are end-to-end flow control, congestion control, error detection and full

duplex connections. It is a fairly complex protocol and is used as the transport on the World Wide Web. The specifications for TCP can be found in [RFC 793].

2.3 UDP

The User Datagram Protocol is an important transport layer protocol in the protocol stack. It offers an unreliable, connectionless datagram delivery service. The strengths of this protocol are simplicity and minimal overheads. It serves as a building block for transport protocols that have payload specific needs, e.g. RTP/RTCP. The UDP specifications can be found in [RFC 768].

2.4 SAP

The Session Announcement Protocol is used to advertise multimedia conferences. It was designed to announce multicast conferences. It announces the existence of a conference by providing to the recipient details such as conference connection info, media streams info, etc. Typically a SAP announcer sends out periodic multicast packets on a well-known address and port. There is no reliability built into the protocol other than what is offered by the lower layers. The protocol does not seek acknowledgement from the listeners and indeed is not even aware if there is any listener receiving the packets hence it is a one-phase protocol. The SAP specifications are detailed in [RFC 2974].

2.5 SIP

The Session Initiation Protocol is a lightweight text-based protocol that has borrowed design elements and mechanisms from other well established and well understood IETF protocols such as HTTP ([RFC 2616]) and SMTP ([RFC 2821]). This has led to the design of a robust set of operations that compliment existing protocols and pave the way to a more complete suite of protocols. The SIP specification is detailed in [RFC 3261]. An extension of interest, to the core SIP specification, is the SIP - Specific Event Notification mechanism defined in [RFC 3265]. A brief outline is given below, and examples of its usage are given afterwards.

2.5.1 Protocol Entities

A **Transaction** contains a request sent from a client to a server and the response to the request generated by the server.

A **User Agent Client (UAC)** is a logical entity that creates a new request and gives it to the underlying layers to transmit. The role of a UAC exists for as long as the transaction.

A **User Agent Server (UAS)** is a logical entity that accepts requests from a UAC and generates either a redirect, accept or reject response in return. Like the UAC, its role also lasts only for the duration of the transaction.

A **Dialog** is signaling relationship that exists between two UAs for some amount of time. The CallID, local tag and remote tag fields of requests and responses identify a dialog.

2.5.2 Methods

The following methods are defined in SIP

REGISTER requests are used to create or update current location information, of a user, in a location service. This location service abstraction is queried to route requests to the appropriate contact address. This is a two-phase protocol. The message exchange in the REGISTER operation is shown in Fig. 2.2(a).

INVITE requests are sent by the User Agent to signal either the intent to set up a session or to modify an existing session. If the INVITE is sent the first time, it is to setup a new session; else it is to modify an existing session. The INVITE request procedure is three phase protocol, meaning that a request-response-ack sequence completes the operation.

ACK requests are used to complete the three-phase session setup procedure using INVITE. It is sent by the UAC to the UAS after the receipt of a

response to the INVITE from the UAS. A typical INVITE operation is diagrammed in Fig. 2.2(b).

CANCEL requests are used to abort pending INVITE requests. It is not recommended to use this method to cancel any other operation, since that would result in a race condition. This is a two-phase protocol. A successful CANCEL operation is illustrated in Fig. 2.2(c).

BYE requests are used to terminate existing sessions. This is a two-phase protocol. The BYE operation is shown in Fig. 2.2(b).

OPTIONS requests are used by a UA to query another UA about its capabilities. Capabilities queried can include the methods and content types supported as well as other information such as available codecs, etc. This operation is illustrated in Fig. 2.2(d).

2.5.3 Responses

Responses are divided into classes, identified by the first digit in a three-digit number (e.g. for a response code 354, the class is 3xx). This scheme has been adopted from protocols like HTTP. Each class of responses represents a generic outcome of the request. More specific information about a response can be found by looking at the other two digits, but it is sufficient for an entity receiving a response to be able to identify the class of response and react accordingly. The classification is given in Table 2.1 ([RFC 3261]).

Table 2.1 - Response Code Classes

Response Class	Response Description	Response Explanation
1xx	Informational	Request received, continuing processing.
2xx	Success	The action was successfully processed and accepted.
3xx	Redirection	Further action required to complete request.
4xx	Client Error	The request contains bad syntax or can’t be fulfilled.
5xx	Server Error	The server failed to fulfill a valid request.
6xx	Global Failure	The request cannot be fulfilled at any server.

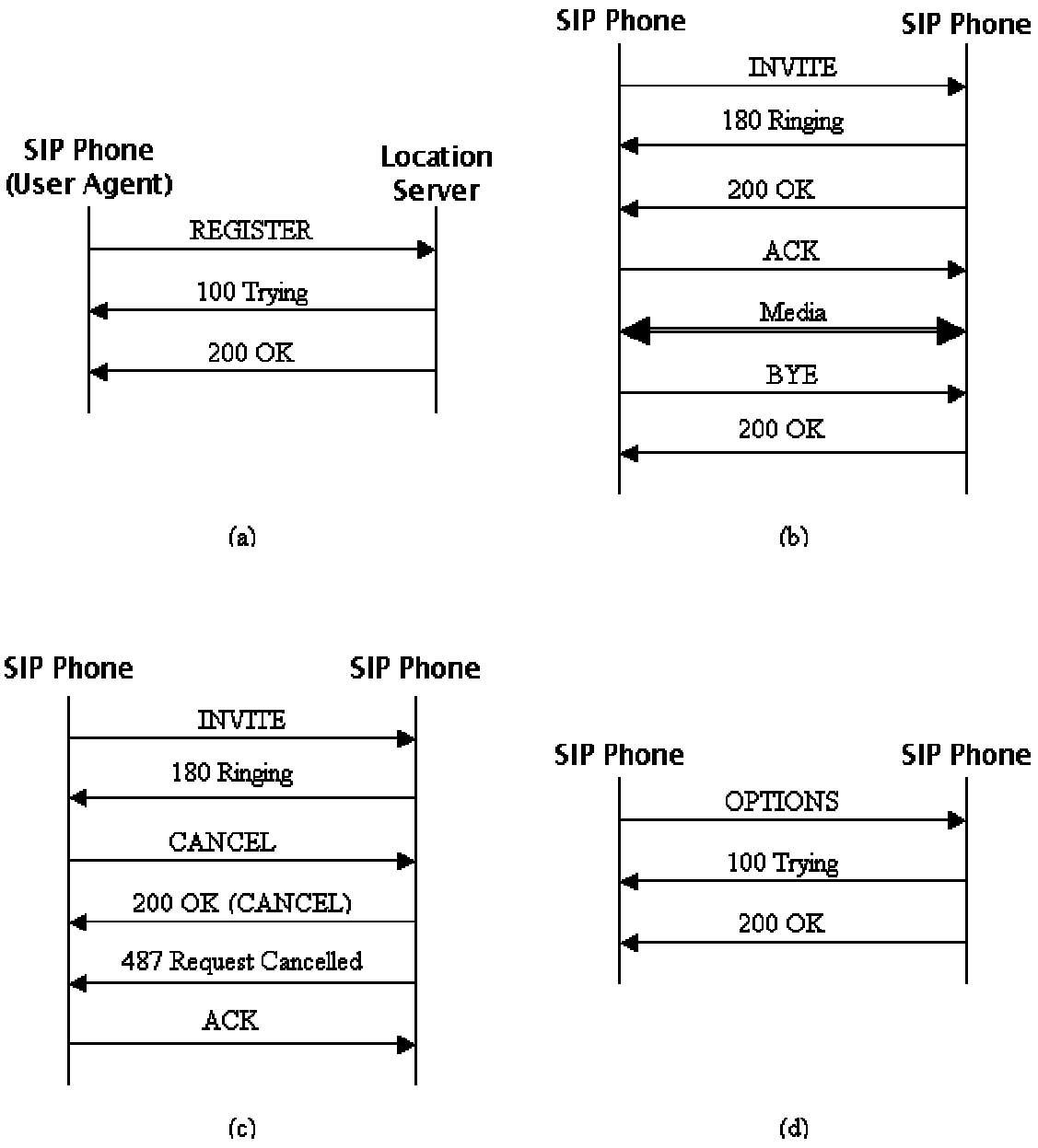


Figure 2.2 – SIP Message exchange (a)REGISTER, (b)INVITE and BYE, (c)CANCEL and (d)OPTIONS

2.5.4 Event Notification Mechanism

SIP also provides a mechanism for asynchronous event notification. User Agents interested in receiving certain events list themselves with the

event source as a listener. When an event of interest occurs, all the listeners are notified. There are two operations defined within this framework.

SUBSCRIBE is used by a party to subscribe to events of interest, with the event source or event distributor. It is also used to renew or remove a subscription. Among other fields, it contains information on where to send the notification and also a time after which the subscription expires. This is a two-phase exchange.

NOTIFY is used by the event source to notify the interested parties of the occurrence of the event. Apart from when the event occurs, NOTIFY is also sent at every instance that a SUBSCRIBE is completed, i.e., when a subscription is created, modified or removed. The notification procedure is also a two-phase exchange.

When a SUBSCRIBE message is received by the Notifier, it sends a NOTIFY message along with the current state of the system. This is shown in Fig. 2.3 (a). An event occurrence at the Notifier results in the generation of a NOTIFY message to all the Subscribers. The message exchange is shown in Fig. 2.3 (b).

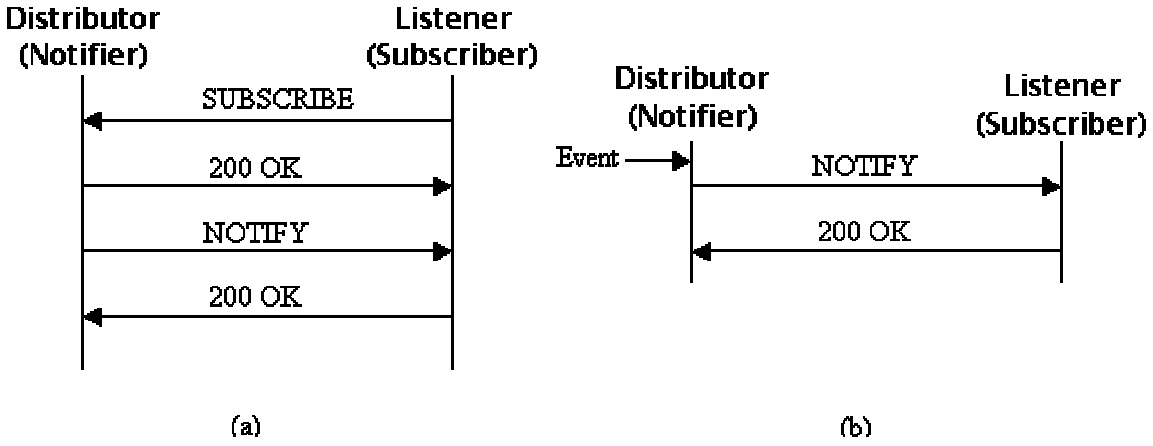


Figure 2.3 – SIP Events Mechanism (a)REGISTER and (b)NOTIFY

2.6 SDP

For a conference to be productive, it is necessary to distribute information such as conference address, media compression being used and other conference tool specific information. Session Description Protocol has been designed to fill in this need. It does not define how this information is distributed, but rather provides a standard format for exchanging this information. As defined in [RFC 2327], SDP defines a format to convey sufficient information to discover and participate in a conference. SDP is also a text-based protocol.

Some important session parameters that are defined in the SDP protocol data unit are session name and purpose, contact information about the person responsible for the conference, conference time (start time, end time, etc), media formats being used, connection information about the media streams (protocol, address, port, encoding scheme, etc), bandwidth limitations of the conference, etc. The fields are of two types, either session level or media level. An SDP message has at least one session level section and optionally multiple media level sections.

The fields in an SDP message as defined in [RFC 2327] are shown below in Fig. 2.4.

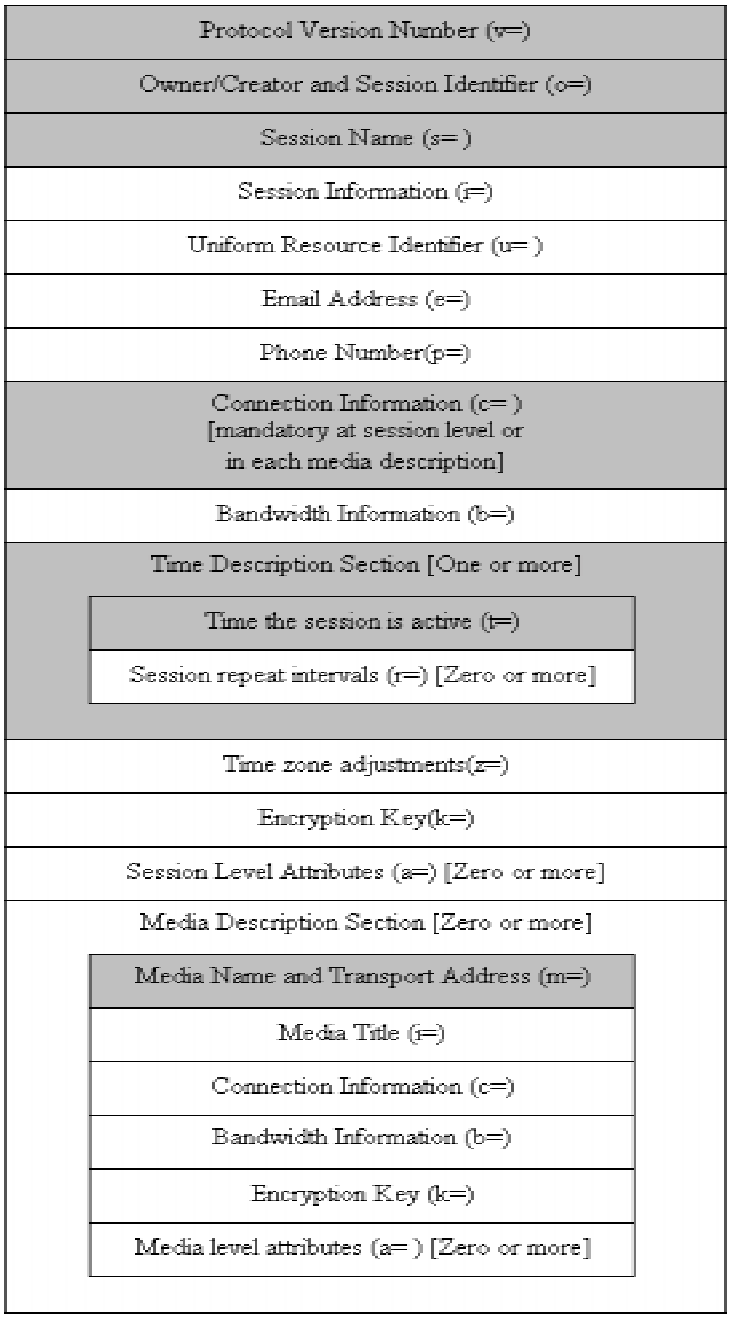


Figure 2.4 – SDP Message Format

An important design consideration for SDP was extensibility. The protocol has inbuilt support for extending itself. The way in which this is done is described below.

The primary means of extending SDP is by using the attribute tag (a=). Since this attribute is available at the session level and the media level, extensions can be defined at either level. Attributes are of two types, either property attributes of the form a=<flag> (e.g., a=sendonly) or value attributes of the form a=<attribute>:<value> (e.g., a=type:moderated). Extensions that are used commonly are registered with the IANA. Unregistered attributes should use a prefix of “X-“ to prevent inadvertent collision with registered names.

2.7 RSVP

Resource ReSerVation Protocol is designed to provide a way for a host to request for reservation of resources to satisfy an application’s quality of service (QoS) requirement. It is unidirectional in the sense that it requests reservation for a single direction of flow. RSVP is composed of two major components - an admission control module and a policy control module. The admission control module ascertains whether sufficient resources are available to satisfy the request while the policy control module determines whether the user has permissions to make the request. The receiver of a data flow makes the reservation requests in RSVP. RSVP is primarily designed to work at the transport layer and hence directly over IP, but it also supports working over UDP for hosts that don’t support raw network I/O. The specifications for RSVP can be found in [RFC 2205], [RFC 2208] and [RFC 2209].

2.8 RTP/RTCP

Real-time Transport Protocol provides an end-to-end data delivery service for data with real-time characteristics. It typically works over UDP, though it can be made to work over other transport layer protocols. RTP by itself does not guarantee timely delivery or a realization of a QoS requirement. It provides constructs such as sequence numbers, timestamps and payload identification. It does not guarantee delivery nor can it ensure ordered delivery. RTP does not have inbuilt mechanisms to recover from packet loss.

RTCP is a related protocol and works in tight conjunction with RTP. It monitors the quality of service and conveys participant information to

other users. In particular, participants can learn about the sources contributing to a particular stream by using RTCP. RTCP transmits packets periodically to participants with control information. It uses the same distribution mechanism that is used for the data. Typically RTP/RTCP operate over a pair of ports over UDP.

The core RTP/RTCP specification can be found in [RFC 1889]. It is however not intended to be completely defined, since many parameters are application specific, to completely define RTP for an application, other documents such as a profile specification document, and a payload format specification document are necessary.

This concludes a presentation of the protocols relevant to conferencing that are available in the IETF. The next chapter delves into the topic of floor control in conferences. The protocols explained in this chapter are used to manage the various aspects of such conferences.

3 Floor Control in SIP

This chapter explains the activity of floor control within the context of conferences. It starts off by defining what floor control is, followed by a review of work done in this area in the past. An overview of conferencing in SIP follows next. This is followed by a presentation on the architecture of floor control in SIP-based conferences. The support available/required for floor control from SDP [RFC 2327] and SIP [RFC 3261] is detailed next. A message level overview of conference setup is given next. The chapter concludes with a brief synopsis of the protocol proposed in [ID CONTROL-02].

3.1 Definition

As defined in [ID REQ-01], a floor is temporary permission given to a conference participant to access or manipulate a shared resource. Floor control (**FC**), hence, deals with the regulation of resources, such as media streams, among users in a conference. The primary aim of this activity is to handle resource conflicts that might arise due to simultaneous access of resources in a shared environment.

There are several approaches to handling resource conflicts. Broadly they can be classified into either those that allow conflicts (permissive) and those that don’t (restrictive).

Examples of the permissive approach are (a) allowing conflicts and providing means to resolve them and (b) ordering requests such that dependencies are detected and minimized. Among approaches that are restrictive, we have those that (a) provide exclusive locks on resources and (b) those that disallow conflicts by using constructs such as tokens. A more thorough handling of this subject is presented in [Dommel 97].

In devising a solution to the floor control problem, it is useful to distinguish between ***mechanisms*** (what are the system capabilities) and ***policies*** (how these capabilities are used to achieve the desired effect). Typically mechanisms have a longer life than policies and a well-designed mechanism is capable of supporting many policies.

To illustrate, consider the floor control problem. To regulate resources, we might implement a token passing mechanism, i.e., only participants with a token may access the given resource. Examples of policies that may be implemented, based on this mechanism, are round-robin access, moderator controlled order, first-come first-served, etc.

A few policies that have been implemented in experimental multimedia conferencing systems are listed in Table 3.1 These have been extracted from literature presented in the next section and are included here to give an idea of commonly used policies.

Table 3.1 - Floor control policies

Policy	Description
No Floor	No floor control.
Pre-emptive	Any participant can request a floor and the current holder has to give it up.
Explicit release	No other participant can obtain the floor unless the current holder explicitly releases it.
FIFO with explicit release	Floor requesters are queued using the FIFO discipline and once the current holder releases the floor, the participant at the head of the queue is assigned to it.
Pause detection	The floor is made available to other participants if no activity is detected from the current holder for some amount of time.
Designation mode	A participant in the role a moderator decides the next floor holder.
Baton mode	The current holder determines next floor holder.
FCFS mode	The floor is granted in order of requests
Free mode	All participants are free to access the resource after requesting for it.

3.2 Literature Review

Floor Control has been studied in depth and a great deal of literature is available, albeit in the context of shared workspaces.

The earliest work on this topic formally defines a floor control protocol for tele-collaboration in [Sarin 85]. It presents a single person floor holding mechanism to avoid resource contention. This however tends to be too restrictive. At the other end of the spectrum is the floor control described in [Ellis 91]. This approach does away with floor control by

relying on social protocols and other out-of-band means to prevent conflicts.

The MMConf system described in [Crowley 90] is designed to support group interactions in a distributed, real-time environment. It was the first system that stressed on the difference between floor control mechanisms and floor control policies. Multicasting was used at the mechanism layer to distribute floor control information. The system supported various floor control policies ranging from implicit-request, implicit-grant to no floor control. Work presented in [Ishii 90] introduces the technique of overlaying a user's collaborative workspace with the user's individual workspace, hence allowing the use of his or her favorite applications in the virtual shared workspace.

A thorough review of co-ordination systems existing at that time is presented in [Ellis 91]. Taxonomies are presented to classify these systems. An attempt at the formal definition of coordination theory has been made in [Malone 90]. It is shown that coordination is a necessary overhead when several groups are performing a common activity. [Kamel 93] presents another classification scheme to group shared workspaces based on various orthogonal criteria. Audio and video channels are also considered as shared workspaces. This work lists key concepts in floor control such as control modes, floor control privileges, clashes, roles and subject/object groupings. In [Greenberg 91] an architecture is presented wherein it is possible to implement and switch between different floor control policies with a few primitives.

A generic framework for floor control has been presented in [Dommel 97]. This work delves into the difference between floor control, access control and concurrency control. It attempts to classify media applications and identifies their floor control requirements. It also presents issues to consider when designing floor control protocols. In [Dommel 98] the authors show that hierarchical group coordination is the most scalable and efficient approach. They also propose a novel protocol that exploits IP Multicasting capabilities to achieve hierarchical group coordination.

The work presented in [ID CONTROL-02] proposes a moderator based floor control protocol for use in SIP based conferencing. It is handled in more

detail in later sections, as it relates directly to the content of this thesis.

3.3 Conference Models in SIP

SIP, initially designed to support large multicast conferences on the MBONE, inherently lends itself to centralized conferencing architectures. The work done on conferencing using SIP is in its infancy, since a large part of the core SIP specification focuses on point-to-point calls. Current work in progress focuses on centralized conferences. Within this scope, definitions and usage in various architectures has been described in [ID MODELS-01] and [ID FRAMEWORK-00]. The elements of a conference and a few of the conference configurations that are of interest are detailed below.

3.3.1 Definitions

A **Tightly coupled conference** is one where there is a central point of communication. This central point provides the various features required for the conference such as media mixing, maintaining SIP point-to-point relationships with the participants, etc. This is the only type of conference considered in this thesis and hence a conference shall refer to this implicitly, unless stated otherwise.

A **Loosely coupled conference** is one where there is no central point of control. Media is distributed through multicast media groups. There are no signaling relationships between the participants.

A **Focus** is a SIP user agent that acts as the central point in a conference. A SIP URI identifies a focus. Participants maintain one-to-one SIP dialogs with this entity. This is a logical role and is composed of other functions such as event notification, conference policy enforcement and ensuring media delivery (indirectly).

A **Participant** is a SIP user agent that maintains a dialog with a Focus in a conference. A participant might itself be a Focus for other participants.

A **Conference Notification Service** is a service provided by the Focus. It implements the event notification mechanism of [RFC 3265] to provide interested parties with notifications of changes in state of the Focus, the Conference Policy and the Media Policy.

A **Mixer** is an entity that processes media streams and sends them to participants, after combining them in some media-specific manner, according to some policies. For voice this may mean that a mixer sends to each participant a mixed voice stream of all other participants.

A **Conference Policy Server** is a function that contains conference rules such as allowed participants, roles of the participants, etc. and means to manipulate these rules. It serves as a storage point of conference policies. The realization of these policies is the work of the Focus.

A **Media Policy Server** is a function that manages the rules associated with the media comprising the conference such as a participant’s choice of media that it wishes to receive and from whom, etc. The actual implementation of these policies is the responsibility of the Focus.

A **Floor Control Server** (FC Server) is a function that regulates the resources of a conference, such as media, among the participants. It interacts with the Media Policy Server as a client to update media policy changes due to floor control procedures. The server includes at the minimum a Floor Controller function and an Event Notification Service to relay floor events.

A **Floor Controller** is a function within the FC Server that handles requests from the Participants. Its responsibilities include managing floors, handling requests from Participants and finally communicating with the Media Policy Server to update the Media Policy if required.

A **Conference Policy** is a set of rules that affect the conference flow. A policy can include, but is not limited to, information such as conference access rights. There is no standard way of defining a policy, but there is a need for a protocol that allows users to manipulate a policy.

A **Media Policy** is a set of rules that specify how a mixer is to operate on the streams that it receives. An example of a media rule may be that a participant wishes to receive the voice stream of a particular participant only. As with conference policies, there is a need for a standard protocol to manipulate these policies, but the policy description is in no way restricted.

A **Floor Control Policy** (FC Policy) is a set of rules that specifies how the floor is regulated among the participants. As an example, the floor policies may require that a moderator approve each request before the floor is given to a participant.

A **Floor Holder** is a Participant who has access to the floor or resource in question. For resources like audio and video, there may be multiple Participants holding a single floor since the Mixer can combine these streams into a single stream to transmit.

With the above definitions in place, it is possible to present conferencing architectures of interest.

3.3.2 Multicast Conferences

This architecture is an example of a loosely coupled conference. The usage of SIP is only to convey media information, to a participant, such as multicast group to join in order to receive a particular media stream. Once a participant is aware of the multicast group to join, he/she uses multicast protocols to join/leave the group, in effect joining and leaving the conference.

Indeed, in this architecture, SIP can be completely bypassed since it only serves as a means to distribute information about the session. This can be handled by SAP or other means such as emails, web pages, etc. So if a participant is able to obtain information about the conference, he/she may join it without the usage of SIP.

To illustrate the usage of SIP in this conference, lets assume that User A, who knows the conference information, wants to invite User B to join in. User A then sends an INVITE to User B that contains the SDP body. This

SDP body provides the multicast address/port for the media streams comprising the conference. User B then uses other means to join the multicast group(s) received in the SDP body. Fig. 3.1 depicts this type of conference.

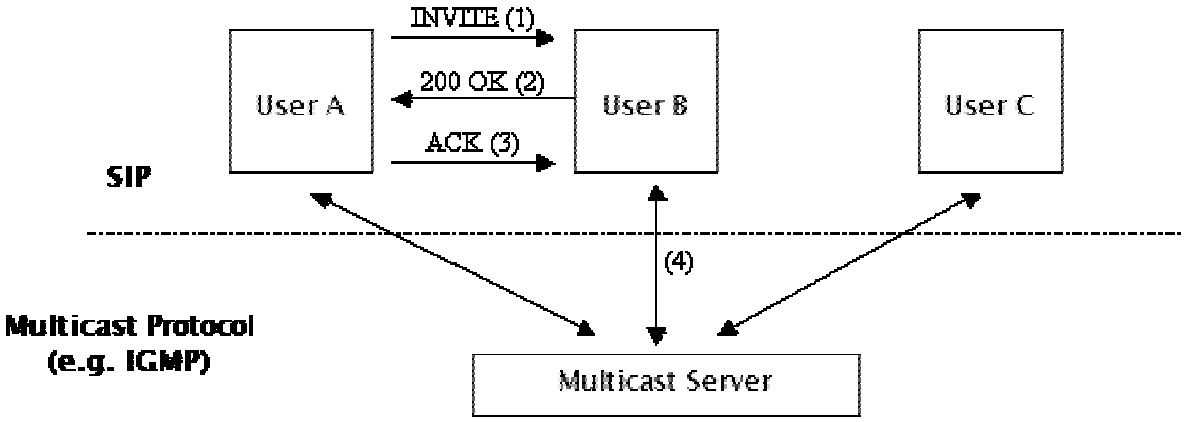


Figure 3.1 – A Multicast Conference Example

3.3.3 Centralized Conferences

The architectures presented in this section are all examples of tightly coupled conferences. These are the most commonly encountered configurations in the present world of telephony. These models are characterized by the presence of a central signaling entity that handles interactions with the participants. Media may/may not be handled by this entity.

3.3.3.1 Centralized Server

Participants connect to a central server that is an all-in-one unit. The conference functions of such a server include a Focus, a Mixer and possibly a Media Policy Server and a Conference Policy Server.

The server acts as the hub of the conference. Each participant maintains a point-to-point SIP call with the server. It is assumed that the server also handles the media, i.e., it receives media streams from participants

and mixes them in a media specific way before sending them to the participants. In this case, the participants become aware of other media-contributing participants using means such as RTP/RTCP or SIP Events.

Participants can join a conference in this architecture either by sending an INVITE to the conference server or by responding to an INVITE from the conference server.

If a participant knows the conference server URI, a participant sends an INVITE to join the conference. He/She is not aware of the media capabilities of the session. So the initial INVITE doesn’t contain any SDP body. The response to the INVITE from the server contains the SDP body to be used for the conference. The participant uses this information to setup the media channels. If a conference server INVITEs a participant to a conference, for example using a pre-set participant list, the initial INVITE contains the media description that the conference is composed of. The participant uses this description to setup the media channels.

It is noteworthy that this architecture assumes that the conference information, such as location to send the initial INVITE to, is available to the participant/server beforehand. As an example scenario, lets consider the case where three participants User A, B and C are conferencing using the server ConfServer. An architectural view of this type of conference is shown in Fig. 3.2.

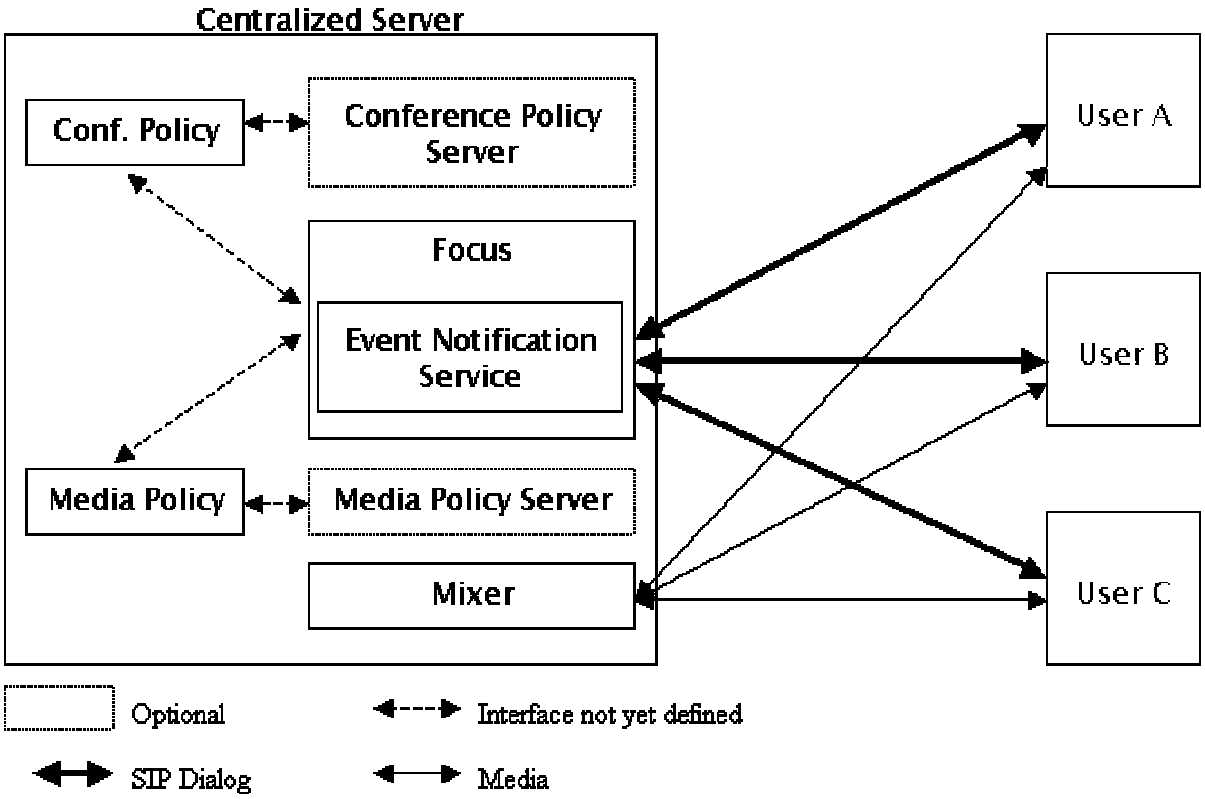


Figure 3.2 – Centralized Server ([ID FRAMEWORK-00])

3.3.3.2 Endpoint Server

In this model, the participant who wishes to conference acts as a focus and a mixer for the other participants. These conferences are also known as ad-hoc locally mixed conferences.

Initially there exists a regular two-party call between two participants. At some point, one of the participants decides to conference in one or more other participants. This participant now changes its role from being a participant to being a focus/mixer + a participant.

The external interfaces to the focus/mixer, at the new server, are the same as those in the central server model. As with a central server, the functions such as Media Policy Server and Conference Policy Server are optional. Similarly, the existence of other participants in the conversation is learned through other means such as RTP/RTCP or SIP Events.

Details of interest are that in this transition to a conference server, the participant creates a new conference URI that is used by all participants; even the existing dialog is modified with a re-INVITE containing the new contact URI. Also, any participant may invite new participants to the conference by acting as an end system mixer, leading to arbitrary conference topologies.

In Fig. 3.3(a) we see a regular two-party call between User A and User B. Fig. 3.3(b) shows the transition of User A to a conference server to accommodate User C. It must be noted that the interface between the server and participant roles of User A is not defined.

A drawback with this approach is that if User A decides to leave the conference, then the conference is over. This approach is suitable for small conferences such as three way calling.

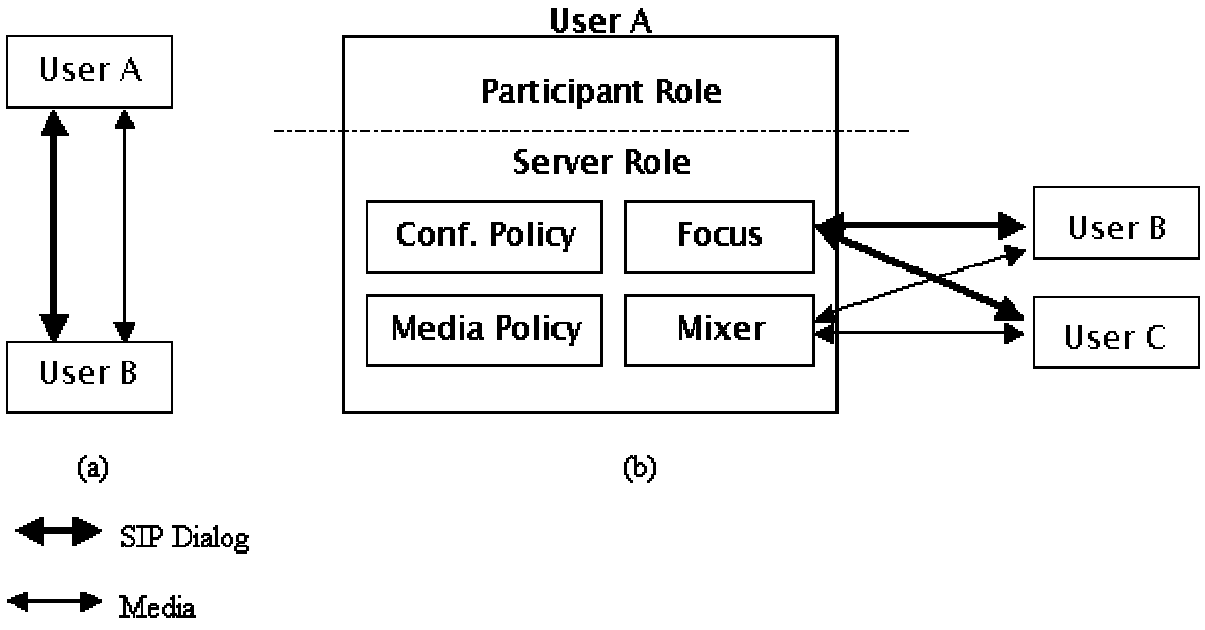


Figure 3.3 – Endpoint Server ([ID FRAMEWORK-00])

3.3.3.3 Media Server

This architecture is of commercial importance since it allows the decoupling of the media mixer and the signaling entity. In this model, each conference has two centralized servers. One of them is called the “Application Server” and acts as the focus of the conference. The other is called the “Mixing Server” that handles the actual mixing functions. The Application Server acts as a controller of the Mixing Server, and is hence called the top-level server.

The Application Server acts as the focus of the conference and users maintain signaling relationships with this server. This server may implement, apart from the mandatory Focus function, a Media Policy Server and a Conference Policy Server.

The Mixing Server is the media handler of the conference. The minimum functionality includes a Focus and a Mixer. This server does not have a Conference Policy Server or the Notification service of the Focus. It has a default conference policy that allows all invitations from the Application Server. Its media policy also accepts all controls from the top-level server.

The participants connect to the Mixing Server to send/receive media streams. The Application Server uses third party call control to setup the media streams between the Participants and the Mixing Server. An architectural view of this model is shown in Fig. 3.4

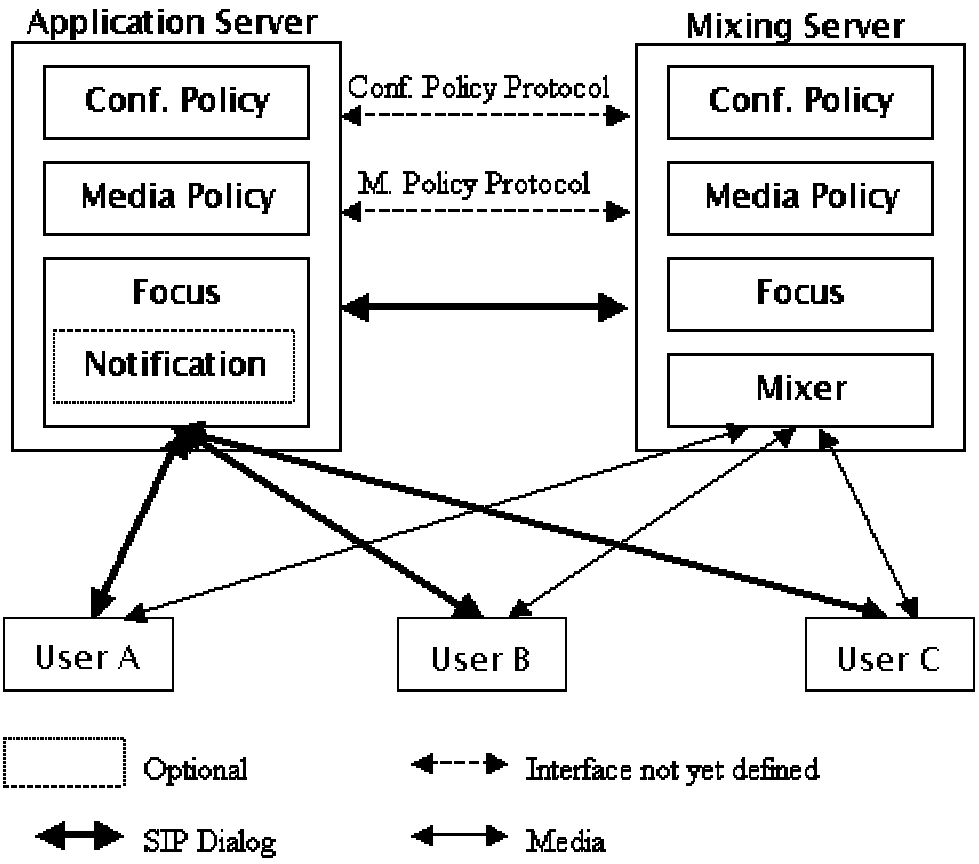


Figure 3.4 – Media Server ([ID FRAMEWORK-00])

3.4 Floor Control Architecture

To support floor control in the architectures presented above, the FC Server entity is introduced. The entity is associated with the conference server and interacts with the Media Policy Server to implement FC Policies. This is consistent with the approach suggested in [ID FRAMEWORK-00] and requirements laid out in [ID REQ-01]. The modified overall conference architecture is shown in Fig. 3.5.

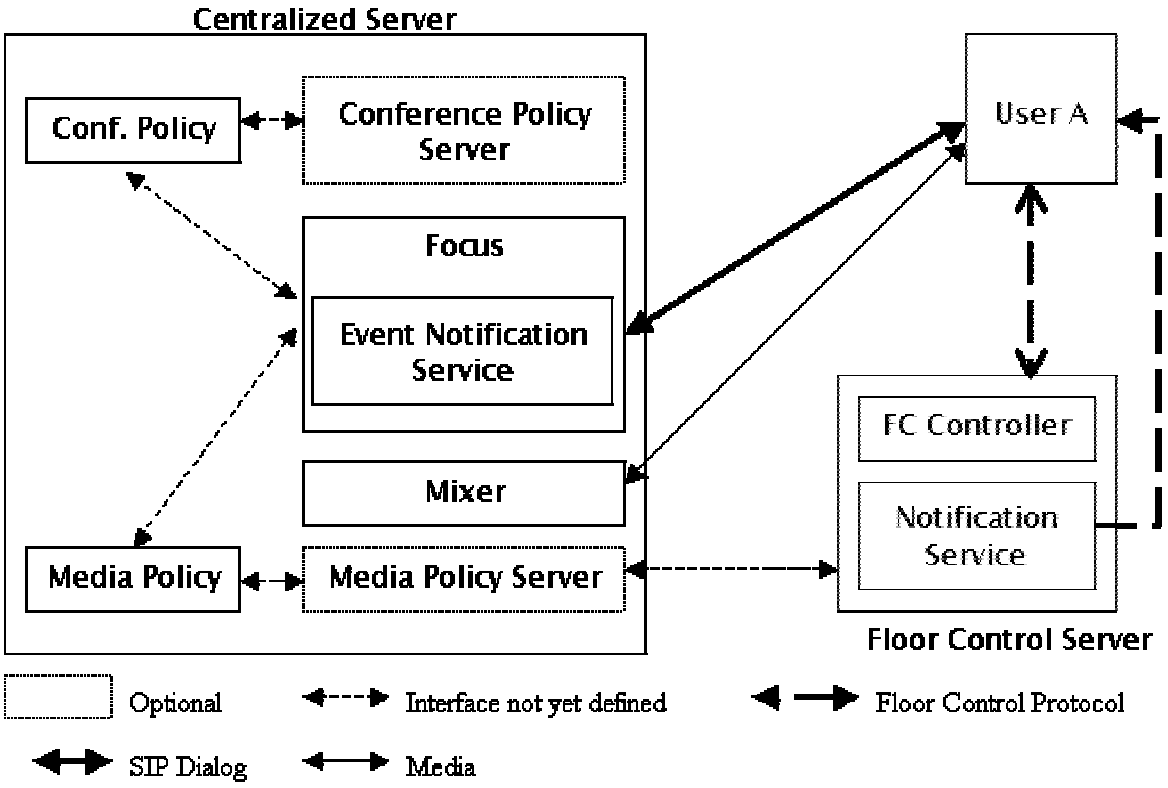


Figure 3.5 – Floor Controlled Conference ([ID REQ-01])

From the Media Policy Server viewpoint, the FC Server is just a client that updates media access policies that are managed by it. The actual message exchanges, which ultimately result in these policy changes, are implemented within the FC Server.

To illustrate, let us consider a moderated media stream where a participant (User A) has to request permission to access the stream from the Moderator (User M). In this scenario, the process of requesting the resource, the moderator's reply and the consequent notification to the participant are implemented within the FC Server. If the request is approved, the Media Policy Server receives a request from the FC Server asking it to update the access permissions to include the approved participant.

With the above architectural overview in place, a more detailed discussion on the support required from SIP/SDP is presented below.

3.5 FC support in SDP

Typically, SDP is used to describe sessions in SIP-based conferences. SDP and its extensions provide a generic set of operations to achieve a broad range of functionality. To be able to use these existing extensions to support FC is the focus of this section. This section starts off by listing the functionality required from SDP to support FC. It then discusses how these functions are implemented using available extensions, as described in [ID CONTROL-02]. Finally it outlines an alternate usage of the extensions, which we are proposing, along with the rationale behind these changes.

The functions that need to be performed to enable the invocation of an FC entity, within the context of a conference, are -

Resource Identification - To be able to specify which of the streams in the SDP body is floor controlled we need to label each of the media descriptions using a unique identifier.

FC Channel Definition - To be capable of describing the transport channels that are to be used for floor control information exchange. These channel descriptions are similar to media descriptions, but instead of them being “audio”/”video”, they are of type “control” and are used by the floor control entity to contact the FC Server.

FC Indication - To be able to set which of the media streams is floor controlled using the identifier mentioned above.

Media stream to FC Channel mapping - To be able to associate a media stream with the transport channel to be used for floor control. Multiple media streams may be associated with the same control channel and multiple control channels may exist in one session.

3.5.1 Existing Proposal

This sub-section lists how each of the functions listed above is handled in the protocol proposed in [ID CONTROL-02]. References are made to documents that describe extensions to SDP [RFC 2327] that are available for use to floor control protocols.

3.5.1.1 Resource Identification

To identify each resource uniquely, the media description lines can be labeled using the “a=mid:” attribute defined in [ID FID-06]. These labels are used to identify the floor associated with the resource and to label FC Channels. The syntax is

a=mid:<identification-tag>

<identification-tag>s are unique within a session description. A fragment that shows its usage is given below. The fragment, labels the audio stream on port 5555 as floor1.

```
m=audio 5555 RTP/AVP 0
a=mid:floor1
```

3.5.1.2 FC Channel Definition

Defining floor control channels can be done using the “m=control” line. This support already exists in SDP. The syntax is

m=control <port> <transport> <fmt list>

The <port> denotes the port number on which the stream is received/sent. The <transport> specifies which transport layer protocol is being used (e.g. TCP) and <fmt list> represents a protocol specific list of parameters. As an example consider the following description of a floor control channel.

```
m=control 80 HTTP SOAP
a=mid:fcchannel01
```

Here the control channel specifies the transport as HTTP on port 80 and additionally specifies the usage of SOAP as part of the fmt-list.

3.5.1.3 FC Indication

Using SDP it is possible to indicate whether a session is moderated or not. This is done using the “a=type:moderated” session attribute. There is currently no way to specify whether a particular media is moderated or not. The result is that all media channels are assumed to be moderated.

3.5.1.4 Media Channel(s) to Floor Control Channel Mapping

To associate media streams with control channels the “a=group:” attribute proposed in [ID FID-06] is used. This construct specifies the grouping of media streams that are floor controlled to the associated FC Channel. The syntax is shown below.

a=group:FL *(space identification-tag)

The identification-tag must be from among the identification-tags in the "a-mid:" lines. Typically the first identification-tag will be a floor control channel’s tag followed by one or more media channel tags. An example SDP fragment incorporating of all the above is shown in Fig. 3.6.

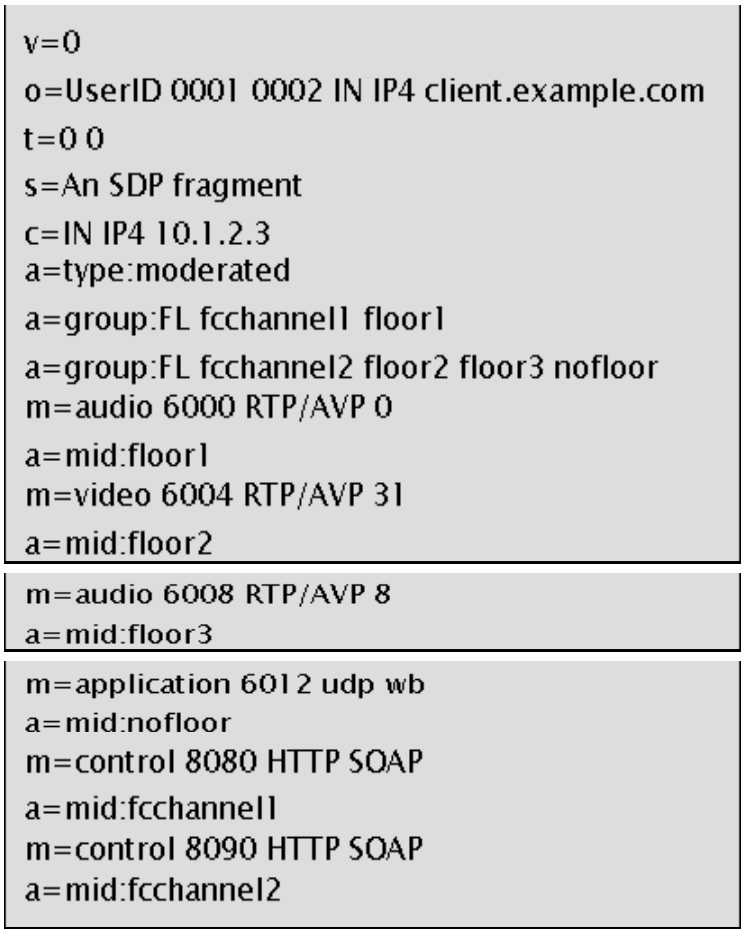


Figure 3.6 – SDP fragment with FC support

In Fig.3.6, we see that there are 4 media channels (floor1, floor2, floor3, nofloor) and 2 floor control channels (fcchannel1, fcchannel2). The associations (floor1 - fcchannel1) and (floor2, floor3, nofloor – fcchannel2) is achieved by using the “a=group:FL” lines.

3.5.2 Proposed Changes

In situations where not all media streams in a conference need to be moderated, the approach presented in [ID CONTROL-02] adds significantly overhead in managing these un-moderated media streams. Our proposal is to avoid this overhead by defining new semantics for the Media Channel to FC Channel mapping mechanism. We remove the session-level “a=type:moderated” attribute unless all media are moderated. Also, we propose that the “a=group:FL” semantics be changed to achieve FC Indication as well as the

mapping. The syntax of the proposed “a=group:FC” attribute is the same as before, but it is to be interpreted as follows

```
v=0
o=UserID 0001 0002 IN IP4 client.example.com
t=0 0
s=An SDP fragment
c=IN IP4 10.1.2.3
a=group:FC fcchannel1 floor1
a=group:FC fcchannel2 floor2 floor3
m=audio 6000 RTP/AVP 0
a=mid:floor1
m=video 6004 RTP/AVP 31
a=mid:floor2
m=audio 6008 RTP/AVP 8
a=mid:floor3
m=application 6012 udp wb
a=mid:nofloor
m=control 8080 HTTP SOAP
a=mid:fcchannel1
m=control 8090 HTTP SOAP
a=mid:fcchannel2
```

Figure 3.7 – Modified SDP Fragment

Each media description that appears in an “a=group:FC” attribute is assumed to be floor controlled. Media descriptions that do not appear in any “a=group:FC” attribute are assumed to be not regulated by floor control.

The SDP of Fig. 3.6, modified to support the proposed semantics, is shown in Fig. 3.7. It is of interest to note that there is no mapping for the “nofloor” media line.

3.6 FC support in SIP

SIP is not a suitable transport for floor control messages since the usage of SIP methods would modify their semantics. It has been proposed in [ID CONTROL-02] to use HTTP as the transport with embedded XML content in the form of SOAP ([W3C SOAP-00], [W3C SOAP-01] and [W3C SOAP-02]) documents. This approach is used in this thesis. Hence SIP requires no modification or new extensions to support floor control. There are however a few definitions that need to be in place to satisfy the requirements for floor control [ID REQ-01].

1. New events to notify participants about floor control events such as change in floor holder, status of request, etc.
2. New events to notify a moderator(s) about moderator-specific events such as new participant request, etc.

The definition and usage of these events is presented in the next chapter since a more complete and meaningful explanation is possible in the complete context of the proposed floor control messages.

3.7 Conference Setup in Centralized model

This section gives a message level view of the operations involved in setting up of a SIP based centralized conference with floor control. To keep the message flow lucid and to avoid cluttering, it is assumed that all the participants dial-in to the conference server. The exchange of messages between the various entities is shown in Fig. 3.8

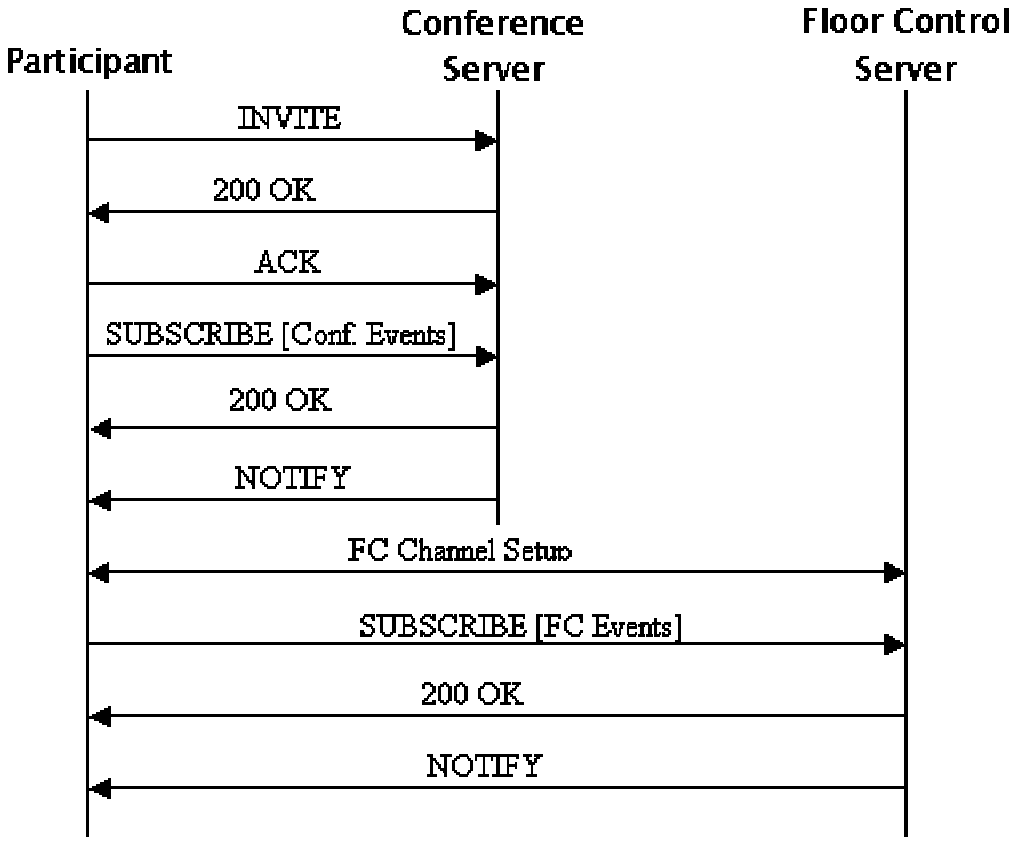


Figure 3.8 – Conference Setup with Floor Control

It may be noted that there are 4 operations here.

1. The Participants send INVITEs to the Centralized Server (Focus), and if the conference policy allows it, the Participant is sent the 200 OK along with the SDP body containing the media channel and FC Channel descriptions. The ACK from the Participant completes the operation of joining the conference.

2. The Participants optionally SUBSCRIBE to the conference event package at the Focus. This results in the generation of a NOTIFY by the Notification Service at the Focus.

3. The Participants, process the SDP body and use it to setup the channels (either media or floorcontrol) to the corresponding Mixer or FC Server.

4. Finally the Participants SUBSCRIBE, with the Notification Service at the FC Server, to obtain floor related event notifications. This also results in the generation of a NOTIFY message by the event notification service.

Once the conference is setup, the Participants may perform floor control operations such as request, release, etc. These requests are made to the Floor Controller and a response to the request implies that the request has been received and is being processed. The notification mechanism is used to inform the Participant about the outcome of the request. If there is any change in the status of the floor, the FC Controller updates the Media policy at the Conference Server using a yet-to-be proposed protocol. This may result in the generation of a re-INVITE to the Participant with the media permissions updated to reflect the current floor state.

If the intervention of a Moderator is required to complete the request, then a Notification message is sent to the Moderator providing details of the request. The Moderator then responds by sending a floor control message to the Floor Controller. The above-described procedure is shown in Fig. 3.9

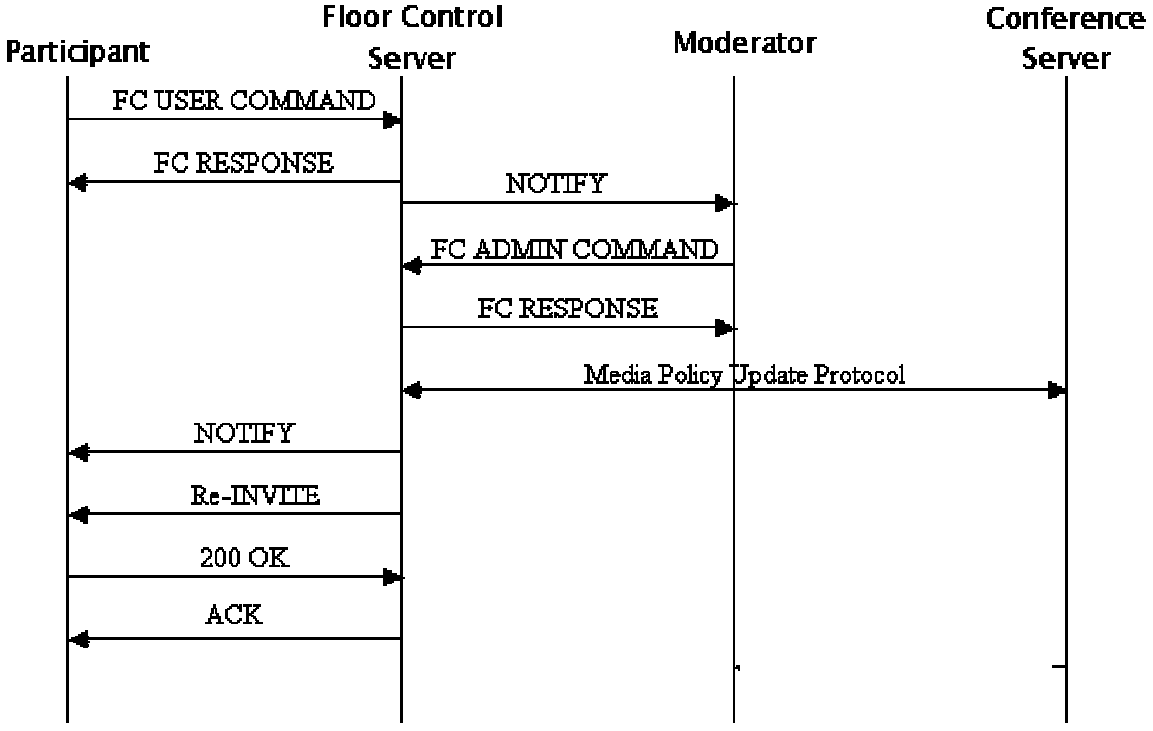


Figure 3.9 – Floor Control Operations Overview

3.8 Currently Proposed Floor Control Protocol

This section presents a brief overview of the protocol proposed in [ID CONTROL-02]. Some key points in this proposal are that a floor may consist of multiple resources and that this protocol is primarily aimed at moderator based floor control. Before listing the message types and the events, the data structures that are used in the protocol are described in Table 3.2.

Table 3.2 - Floor Control Data Structures

Data Structure	Description
floorType	This represents the parameters of a floor such as max-holders, resources that constitute this floor, users, moderators, etc.
resourceType	This represents each m= line that is floor controlled. Its content is the “a=mid:” value for the media.
usersType	This represents Participants in the conference that are allowed to request for floors.
moderatorsType	This represents a list of moderators for a floor.
holdingType	This is a mapping between participants and the resources that they hold.
claimType	This represents the request for a floor and contains information such as resources, expected period of hold, etc. A list of claims is represented by claimsType.
operationType	This is used to manipulate the queue of requests by the moderator and has operations such as up, down, top, bottom.

3.8.1 Messages

All the messages in this protocol return boolean values indicating whether the command was successfully processed or not. In the case of Moderator commands the value indicates whether the execution of the command was successful or not. In the case of Participant requests such as ClaimFloor, the value represents whether the command was received and accepted at the Server. A C-like syntax of the commands and a brief explanation is given in Table 3.3. The list of events generated is shown in Table 3.4.

Table 3.3 - Floor Control Messages

Messages	Description
CreateFloor(floorType)	The Moderator uses this to create floors on the FC Server.
RemoveFloor(resourceType)	The Moderator uses this to remove floors on the FC Server.
ChangeConfig(floorType)	The Moderator uses this command to modify parameters of the floor.
ClaimFloor(claimsType)	This is used by a Participant to request for a floor.
ReleaseFloor(holdingType)	A Participant uses this to give up the floor that he/she currently holds.
GrantFloor(holdingType, claimType)	This is used by a moderator to give floor access to Participants.
RevokeFloor(holdingType)	The Moderator uses this to force a Participant to give up a floor that he/she is currently holding.
RemoveClaims(claimsType)	This command is used by the Moderator to remove claims from the claims queue.
ReorderClaims(resourcesType, claimType, operationType)	This command is used to modify the order of claims in the claims queue by the Moderator.

Table 3.4 - Floor Control Events

Events	Description
FloorCreated	This event is generated when there is a new floor created by a moderator. It is distributed to the Participants.
FloorRemoved	This event is generated when a floor is removed. It is sent from the FC Server to the Participants.
ConfigChanged	When floor parameters (such as moderators, max-holders) are changed, this event is generated. It is sent from the FC Server to the Participants.
floorChanged	When there is a change in the list of floor holders, this event is sent from the FC Server to the Participants.
queueChanged	This event is generated when there is a change in the claims queue. This may be the result of a new request. It is sent from the FC Server to the Moderator of the floor.

The above messages and events are sent between the various floor control entities as shown in Fig. 3.9. The messages exchanges between the Participant and the FC Server are ClaimFloor and ReleaseFloor. The messages sent from the Moderator to the FC Server are CreateFloor, RemoveFloor, ChangeConfig, RevokeFloor, RemoveClaims and ReorderClaims.

All events are generated by the FC Server and distributed to the Participants except the QueueChanged event that is sent to the Moderator.

This concludes our discussion on floor control and existing work in this area. The next chapter presents the protocol proposed in this thesis.

4 Protocol Design

As mentioned earlier, it is important to differentiate between mechanisms and policies when designing the protocol so that a relatively small set of primitives satisfy a diverse range of floor control requirements. This chapter starts off by presenting a set of useful policies that the floor control protocol to be designed must support. It is then followed by the elements of protocol definition.

The protocol definition contains a *service description* that provides a high level overview of the services provided by the protocol. The realization of these services in a particular *environment* leads to the creation of *operations* and these are presented next. The *format* or *syntax* of these operations is then described. Finally the *procedure rules* are explained.

4.1 Floor Policies

Three orthogonal criteria are used to classify the configurations being considered.

1. The first is whether there is a moderator present or not. The presence of the moderator indicates that the selection of the next floor holder is made by an entity rather than by factors such as first-come-first-serve.

2. The second criterion used is whether or not there exists a queue of participant requests that specifies the order in which the floor is to be granted to the participants (Access queue or AccessQ for short).

3. The third criterion is whether or not a floor holder is pre-emptible, i.e., whether the current holder is replaced when a new request is received or whether the request is simply rejected.

Using these three criteria, we obtain the combinations shown in Table 4.1 and Table 4.2. A brief explanation of each criteria combination follows.

Table 4.1 – Floor without pre-emption

	Moderated	**Unmoderated**
AccessQ	X	X
No AccessQ	Rejected	Rejected

4.1.1 Moderated with AccessQ

A configuration of this nature is used when a moderator sets the order in which the participants get the floor. Requests for the floor are processed and is accepted are put into the AccessQ. The moderator, to change the order of access, can manipulate this queue.

4.1.2 Unmoderated with AccessQ

This combination of elements serves as a preordered queue with no moderator. As soon as the floor becomes free, the next participant in the queue is give access to the floor. An unmoderated FCFS discipline is its most likely usage, though other queuing disciplines can be implemented (priority based, etc.)

4.1.3 Moderated/Unmoderated without AccessQ

For the sake of completeness, we address this case here. This is not a configuration per se. It always results in the rejection of requests made by the participants in case the floor is already taken. Since there is no queue to store requests and there is also no pre-emption policy, new requests can only be rejected.

Table 4.2 – Floor with pre-emption

	Moderated	**Unmoderated**
AccessQ	NA	NA
No AccessQ	X	X

4.1.4 Moderated with No AccessQ

This configuration can be used to implement a variety of policies. As requests come in, they are handled and responded to by the moderator. The selection made is of the next participant to replace the current floor holder. Some examples of policies that can be implemented are baton passing and pre-emption. An open issue is which of the current floor holders to replace in case of multiple floor holders.

4.1.5 Unmoderated with No AccessQ

This setup may be used to implement situations where there might be a limitation in the number of holders a floor can support, but without the intervention of a moderator. The central server may directly accept or reject the request depending on the number of simultaneous holders a floor can accommodate.

4.1.6 No Floor

In addition to the above configurations, we also have situations where there is no requirement of regulating a resource in any way. This can be implemented using the unmoderated criterion. But this would involve unnecessary overhead such as creation of control channels, processing involved in handling requests, etc. To avoid this, I propose the use of a configuration called “no floor”.

4.2 Service specification

The purpose of the protocol is to regulate multimedia resources among members of a centralized multiparty conference. The protocol works in conjunction with, and makes maximum utilization of, mechanisms already present in SIP. The protocol consists of request-response messages pairs and asynchronous event notification messages. It takes into consideration that some resources in a session do not require regulation. Mechanisms are designed to support automation of resource allocation. Each resource is associated with one floor. A participant is allowed to request floors and may access them only after the required authorization has been obtained.

4.3 Environment description

The environment in which the protocol is expected to execute consists normally of two or more parties in a centralized SIP based conference. The FC Server acts as the hub for coordinating the message exchanges of the protocol as well as handling session parameter changes. This process has been discussed in Chapter 3. The server maintains state variables such as users with permissions to create floors, etc. These are obtained out-of-band with respect to the protocol being discussed.

The FC Server maintains (optionally) two queues, one is for incoming, unprocessed requests (Request Queue) and one is for requests that have been processed and are to be queued for floor access (Access Queue).

If the media is floor controlled, there is a moderator for that resource. Participants request a resource from the moderator of that resource through the FC Server. The moderator then decides whether to accept or reject the request. The involved parties are then notified of any changes that might have taken place in the floor control domain.

It is assumed that there is a reliable transport, such as TCP, available for the messages/events of the floor control protocol. Even such protocols may fail and to handle these failures typical protocol constructs such as timers are required to recover state. These timers are typically tied to the transport timers.

In the event of a message loss, there is no requirement to reset the state, since no update has taken place on the server. If, however, the response gets lost, the participant will need to timeout and resend the message. Message duplication is handled by the use of sequence number in the messages. So if a server receives a duplicate request, the response is the resent. The loss of events is handled by the event mechanisms proposed in [RFC 3265].

4.4 Vocabulary (Operations)

This section describes the various operations that need to be supported by the protocol being designed. The vocabulary gives an overview of the functions provided by the protocol. These are not to be confused with protocol primitives, which give the exact message syntax. An operation may require multiple protocol primitives to implement. As an example, the operation of *session setup* using SIP requires three protocol primitives, an INVITE message, a Response and an ACK.

Depending on the usage, operations are classified into moderator and participant operations. Moderator operations deal with managing of floors and user requests while the participant operations deal with the types of requests made by participants. Unless otherwise noted, all operations are designed to encapsulate multiple requests.

4.4.1 Moderator Operations

Create Floors – The moderator uses this operation, to create floors and associate resources to them. This also involves setting up of other parametric information such as maximum number of simultaneous floor holders, moderator list, etc.

Freeze Floors – This operation is used to suspend any further requests for the indicated floors. On success of this operation, the FC Server no longer accepts new requests for the suspended floors.

Destroy Floors – The moderator issues this operation to remove floors.

Manipulate Floors – To change any floor related parameter or state this operation is used. Examples of change in floor info are manipulation of floor queues, changing the list of moderators for the floors, etc.

Grant Floors – This operation is used to allow a Participant to access a floor. Typically, the moderator uses it, to respond to a request made by a Participant.

Transfer Floors – The purpose of this operation is to enable the Moderator to substitute a current floor holder with another Participant. It is useful in situations where a Participant requests that a floor be transferred to another Participant.

Revoke Floors – The aim of this operation is to take away access permissions from an existing floor holder.

Get Floor Info - The purpose of this operation is for the moderator or any allowed participant to obtain information about the floor. This includes information such as the parameters of the floor (e.g. max-holders) as well as the status of the floor (e.g. queue status).

4.4.2 Participant Operations

Request Floors – Participants use this operation to request access to floors.

Release Floors – This operation is used by the Participant to release the floors that he currently holds.

Yield Floors – This operation is used by a Participant to transfer the floor to another Participant.

Cancel Floor Requests – To cancel requests that are queued up at the FC Server, Participants use this operation.

4.5 Syntax (Message formats)

This section presents the floor control request, response and event primitives and their formats. Along with each request message, the possible responses are discussed. The events generated are detailed after that. As proposed in [ID CONTROL-02], we use XML Schemas to describe the messages. These can be encapsulated as SOAP messages and transported using protocols such as HTTP. As in the previous section, request messages are discussed under two sub-sections Moderator and Participant Requests. Before presenting the message syntax, we present the XML structures used to describe the data.

Before presenting our work, we first list the elements that have been borrowed from work done in [ID CONTROL-02]. Each of these elements have been modified to account for the difference in approaches between this protocol and the one in [ID CONTROL-02].

1. The idea of using an XML Schema to describe the protocol messages and events.
2. usersType and moderatorsType ([ID CONTROL-02]) are analogous to UserIDType.
3. floorType ([ID CONTROL-02]) is analogous to the FloorStateType with modifications to the structure to accommodate for the two queues.
4. holdingType ([ID CONTROL-02]) is analogous to HoldingType with changes to accommodate for permissions.
5. floorType ([ID CONTROL-02]) is analogous to FloorParametersType with changes to accommodate for initializing the two queues, pre-emption and a per-user permissions list.
6. claimType ([ID CONTROL-02]) is analogous to the parameters in the *request-floors* operation with changes to accommodate for user preferred permissions.

4.5.1 Schema Definitions

This section defines the common XML Schema element types used by the messages. These are analogous to defining classes in object-oriented languages. The actual instances of these types are realized within the context of the messages/events. A complete listing of the XML Schema is given in [Appendix I].

4.5.1.1 FloorID

Each floor has a Floor Identifier. This is the same as the value in the "a=mid:" tag from the SDP body. FloorIDType represents a single floor identifier while a FloorIDsType represents a list of identifiers.

```
<xs:simpleType name="FloorIDType">
        <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:complexType name="FloorIDsType">
        <xs:sequence>
                <xs:element name="floor-id" type="FloorIDType" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
```

Figure 4.1 – Floor Identifiers

4.5.1.2 UserID

A User Identifier uniquely identifies each Participant. The UserIDType represents this identifier.

```
<xs:simpleType name="UserIDType">
        <xs:restriction base="xs:string"/>
</xs:simpleType>
```

Figure 4.2 – User Identifier

4.5.1.3 Permissions

This represents the permissions that are associated with a floor (media). These match the SDP values allowed for media streams (sendonly/ recvonly/ sendrecv).

```
<xs:simpleType name="PermissionsType">
        <xs:restriction base="xs:string">
                <xs:enumeration value="sendonly"/>
                <xs:enumeration value="recvonly"/>
                <xs:enumeration value="sendrecv"/>
        </xs:restriction>
</xs:simpleType>
```

Figure 4.3 – Permissions

4.5.1.4 Queues

As mentioned earlier, there are two queues maintained at the FC Server. The following types represent the information maintained in the queues. The information is divided into two categories, parametric information and state information.

Queue Parameters

Currently the only parameter supported for the queue is the maximum queue size.

```
<xs:complexType name="QueueParametersType">
        <xs:sequence>
                <xs:element name="max-size" type="xs:integer"/>
        </xs:sequence>
</xs:complexType>
```

Figure 4.4 – Queue Parameters

Queue State

The queue state information consists of data such as the current queue contents and current queue size. The queue contents are request-list of requests made by Participants and its current size. The structures of the individual participant requests are presented in [Section 4.5.3].

```
<xs:complexType name="QueueStateType">
        <xs:sequence>
                <xs:element name="size" type="xs:integer"/>
                <xs:element name="request-list" type="ParticipantRequestListType"/>
        </xs:sequence>
</xs:complexType>
```

Figure 4.5 – Queue State

The *request-list* is defined in ParticipantRequestListType as a sequence of participant request types. Its structure is shown below.

```
<xs:complexType name="ParticipantRequestListType">
        <xs:sequence>
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element ref="request-floors"/>
                        <xs:element ref="release-floors"/>
                        <xs:element ref="yield-floors"/>
                        <xs:element ref="cancel-requests"/>
                        <xs:element ref="get-state-floors"/>
                        <xs:element ref="get-parameters-floors"/>
                </xs:choice>
        </xs:sequence>
</xs:complexType>
```

Figure 4.6 – Request List

4.5.1.5 Holdings

A HoldingType represents information about a current floor holder. It has parameters such as the user-id, floor-id, permissions, start-time and end-time of the access rights. If the end-time is not specified, the holding time is unbounded. Also if no permissions are specified, then it defaults to sendrecv. A HoldingsType is a sequence of holdings.

```
<xs:complexType name="HoldingType">
	<xs:sequence>
		<xs:element name="user-id" type="UserIDType"/>
		<xs:element name="floor-id" type="FloorIDType"/>
		<xs:element name="start-time" type="xs:dateTime" minOccurs="0"/>
		<xs:element name="end-time" type="xs:dateTime" minOccurs="0"/>
		<xs:element name="permissions" type="PermissionsType" default="sendrecv"
minOccurs="0"/>
	</xs:sequence>
</xs:complexType>

<xs:complexType name="HoldingsType">
	<xs:sequence>
		<xs:element name="holding" type="HoldingType" minOccurs="0"
maxOccurs="unbounded"/>
	</xs:sequence>
</xs:complexType>
```

Figure 4.7 – Holdings

4.5.1.6 Floor Information

All information pertaining to a floor maintained at the FC Server is encapsulated in the following schema representation. As with the Queues, the information is divided into parametric information and state information.

Floor Parameters

This type represents the properties of a floor. It consists of the following information – description string, policy string, parameters for the two queues, maximum allowed holders, the moderator list and the allowed permissions for each user. The structure is shown in Fig. 4.8.

```
<xs:complexType name="FloorParametersType">
  <xs:sequence>
    <xs:element name="description" type="xs:string" minOccurs="0"/>
    <xs:element name="policy" type="xs:string" minOccurs="0"/>
    <xs:element name="pre-emptible" type="xs:boolean" minOccurs="0"/>
    <xs:element name="access-queue-parameters" type="QueueParametersType"
minOccurs="0"/>
    <xs:element name="request-queue-parameters" type="QueueParametersType"
minOccurs="0"/>
    <xs:element name="max-holders" minOccurs="0"/>
    <xs:element name="moderator-list" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
            <xs:element name="moderator" type="UserIDType" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="user-permissions-list" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
            <xs:element name="user-permissions">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="user" type="UserIDType"/>
                        <xs:element name="permissions"
type="PermissionsType"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="floor-id" type="FloorIDType" use="required"/>
</xs:complexType>
```

Figure 4.8 – Floor Parameters

Floor State

The floor state maintained at the FC Server is represented by this type. It has the current contents of the queues and the current holding list. The structure is shown in Fig. 4.9.

```
<xs:complexType name="FloorStateType">
        <xs:sequence>
                <xs:element name="holding-list" type="HoldingsType" minOccurs="0"/>
                <xs:element name="access-queue-state" type="QueueStateType" minOccurs="0"/>
                <xs:element name="request-queue-state" type="QueueStateType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="floor-id" type="FloorIDType" use="required"/>
</xs:complexType>
```

Figure 4.9 – Floor State

4.5.2 Moderator Requests

4.5.2.1 moderator-response **create-floors(**floor-parameters+**)**

The moderator uses this command to create new floors on the FC Server. The arguments of this command are floor-parameters for each floor. The return values for this command are a boolean value and a reason phrase, for each floor, whether the creation was successful or not. The request structure is shown in Fig. 4.10. The per floor response is represented by *ModeratorFloorResponseType* while a list of floor responses is represented by *moderator-response*. The response structure is shown in Fig. 4.11.

```
<xs:element name="create-floors">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-parameters" type="FloorParametersType"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.10 – Create Floors

```
<xs:complexType name="ModeratorFloorResponseType">
        <xs:sequence>
                <xs:element name="status" type="xs:boolean"/>
                <xs:element name="status-phrase" type="xs:string" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="floor-id" type="FloorIDType" use="required"/>
</xs:complexType>

<xs:element name="moderator-response">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-response" type="ModeratorFloorResponseType"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.11 – Create Floor Response

4.5.2.2 moderator-response **freeze-floors(**floor-id+**)**

The moderator uses this command to make this floor unavailable for further requests from Participants. The arguments for this command are a list of floor-ids that are to be suspended. The request structure is shown in Fig. 4.12. The response indicates whether the floor was suspended or not and is shown in Fig. 4.11.

```
<xs:element name="freeze-floors">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-id" type="FloorIDType"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.12 – Freeze Floors

4.5.2.3 moderator-response **remove-floors(**floors-id+**)**

This command is used to withdraw the floor and hence make it unavailable to the Participants. The arguments for this command are a list of floor-ids that are to be removed. All information stored on the FC Server pertaining to the floor is removed. The request structure is shown in Fig. 4.13. The response is the same as shown in Fig. 4.11.

```
<xs:element name="remove-floors">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-id" type="FloorIDType"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.13 – Remove Floors

4.5.2.4 moderator-response **modify-parameters-floors(**floor-parameters+**)**

The moderator uses this command to change any of the floor parameters such as moderator list, number of maximum holders, etc. The command arguments are a list of floor-parameters. The response, indicating whether the modification was successful or not, is shown in Fig. 4.11. The command is shown in Fig. 4.14.

```
<xs:element name="modify-parameters-floors ">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-parameters" type="FloorParametersType"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.14 – Modify Floor Parameters

4.5.2.5 moderator-response **modify-state-floors(**floor-state+**)**

To modify any of the floor state related information such as the queue contents and the current floor holder list, this command is used. The request contains the new floor-state information. The response (Fig. 4.11) indicates whether the modification was successful or not and a reason phrase. The request is illustrated in Fig. 4.15. It is worthwhile to note that the moderator uses this command to grant a floor to a participant, enqueue the request in the access queue and clear out the queues.

```
<xs:element name="modify-state-floors">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-state" type="FloorStateType"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.15 – Modify Floor State

4.5.3 Participant Requests

This section provides the details of the requests that may be made by the participant. As in the previous section, a brief explanation is given for each command along with a C-like syntax and then the XML Schema fragment defining the command is presented.

All user requests have a mandatory request-id attribute that is used to match requests to responses. Additionally the request creation/expiration timestamps are specified.

Almost all participant responses include only minimal information that indicates whether the FC Server has admitted the request or not. They do not indicate whether the request was successful or not. That information is obtained asynchronously through the events package. For e.g., UserA makes a request for Floor1, the FC Server enqueues the request and immediately returns a boolean value “true” indicating that the request has

been received. This however in no way determines whether the moderator has granted the floor to UserA or not.

4.5.3.1 participant-response **request-floors**(user-id, …)

Participants use this request to gain access to a floor or list of floors. Within one command, asking for multiple floors means that the users require all the listed floors or none. If this constraint is not important to the participant, he/she uses multiple such requests one for each of the floors desired. The optional parameters of this request are the floor-id-list, permissions, reason-phrase and the expected-holding-time. The response structure is given in Fig. 4.17.

```
<xs:element name="request-floors">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="user-id" type="UserIDType"/>
                        <xs:element name="floor-id-list" type="FloorIDsType" minOccurs="0"/>
                        <xs:element name="permissions" type="PermissionsType"
default="sendrecv" minOccurs="0"/>
                        <xs:element name="reason-phrase" type="xs:string" minOccurs="0"/>
                        <xs:element name="expected-holding-time" type="xs:duration"
minOccurs="0"/>
                        <xs:element name="request-create-time" type="xs:dateTime"
minOccurs="0"/>
                        <xs:element name="request-expire-time" type="xs:dateTime"
minOccurs="0"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.16 – Request Floors

```
<xs:complexType name="ParticipantFloorResponseType">
        </xs:annotation>
        <xs:sequence>
                <xs:element name="status"/>
                <xs:element name="status-phrase" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
</xs:complexType>

<xs:element name="participant-response" type="ParticipantFloorResponseType"/>
```

Figure 4.17 – Participant Response

4.5.3.2 participant-response **release-floors**(user-id, …)

To release floors that a participant holds, this request is used. The optional parameters include the floor-id-list and the reason-phrase. If a floor-id-list is not included, then this request is to release all floors held by the specified user. The return values are the same as shown in Fig. 4.17. The request structure is shown in Fig. 4.18.

```
<xs:element name="release-floors">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="user-id" type="UserIDType"/>
                        <xs:element name="floor-id-list" type="FloorIDsType" minOccurs="0"/>
                        <xs:element name="reason-phrase" type="xs:string" minOccurs="0"/>
                        <xs:element name="request-create-time" type="xs:dateTime"
minOccurs="0"/>
                        <xs:element name="request-expire-time" type="xs:dateTime"
minOccurs="0"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.18 – Release Floors

4.5.3.3 participant-response **yield-floors**(user-id, to-user-id, …)

A participant uses this to request transfer of their holding to another participant. The mandatory parameters of this command are the current holder's user-id and the target user-id. Optionally a list of floor-ids and a reason-phrase may be specified. If the floor-id-list is not specified, the request is to transfer all currently held floors to the specified user, else the request applies to the listed floors. The response is the same as in Fig. 4.17 and the request is shown in Fig. 4.19.

```
<xs:element name="yield-floors">
	<xs:complexType>
		<xs:sequence>
			<xs:element name="user-id" type="UserIDType"/>
			<xs:element name="to-user-id" type="UserIDType"/>
			<xs:element name="floor-id-list" type="FloorIDsType" minOccurs="0"/>
			<xs:element name="reason-phrase" type="xs:string" minOccurs="0"/>
			<xs:element name="request-create-time" type="xs:dateTime"
minOccurs="0"/>
			<xs:element name="request-expire-time" type="xs:dateTime"
minOccurs="0"/>
		</xs:sequence>
		<xs:attribute name="req-id" type="RequestIDType" use="required"/>
	</xs:complexType>
</xs:element>
```

Figure 4.19 – Yield Floors

4.5.3.4 participant-response **cancel-requests**(user-id, …)

To cancel pending requests in either of the queues on the FC Server, the participant uses this command. The required parameter is the user-id. Optional parameters include a list of request-ids to cancel. If no request-id-list is specified, this command cancels all the user's request on the FC Server, else only the specified requests are removed. The request structure is shown in Fig. 4.17, the request structure in Fig. 4.20.

```
<xs:element name="cancel-requests">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="user-id" type="UserIDType"/>
                        <xs:element name="request-id-list" minOccurs="0">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element name="request-id"
type="RequestIDType" maxOccurs="unbounded"/>
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                        <xs:element name="reason-phrase" type="xs:string" minOccurs="0"/>
                        <xs:element name="request-create-time" type="xs:dateTime"
minOccurs="0"/>
                        <xs:element name="request-expire-time" type="xs:dateTime"
minOccurs="0"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.20 – Cancel Requests

4.5.3.5 get-parameters-response **get-parameters-floors**()

A Participant uses this request to get parametric information about a floor. The command takes in optionally, as parameters a list of floor-ids. The response contains a list of floor-parameters for the requested floors. If no floor-ids are provided the command returns parameters for all floors. The response is shown in Fig. 4.22 and the command syntax is shown in Fig. 4.21

```
<xs:element name="get-parameters-floors">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-id" type="FloorIDType" minOccurs="0"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.21 – Get Parameters

```
<xs:element name="get-parameters-response">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-parameters" type="FloorParametersType"
minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.22 – Get Parameters Response

4.5.3.6 get-state-response **get-state-floors()**

A Participant uses this command to request for floor-states. The command takes as an optional parameter a list of floor-ids. The response to this command includes a list of floor-states for each of the floor-ids specified. If no floor-id-list is specified in the command, floor-state of all floors is requested for. The request structure is shown in Fig. 4.23 and the response structure is shown in Fig. 4.24.

```
<xs:element name="get-state-floors">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-id" type="FloorIDType" minOccurs="0"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.23 – Get State

```
<xs:element name="get-state-response">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-state" type="FloorStateType"/>
                </xs:sequence>
                <xs:attribute name="req-id" type="RequestIDType" use="required"/>
        </xs:complexType>
</xs:element>
```

Figure 4.24 – Get State Response

4.5.4 Floor Control Events

Floor control events are distributed by the FC Server Notification mechanism to subscribed participants. Typically all changes in either floor state or floor parameters results in generation of an event. Therefore most of the requests presented previously result in event generation. This section presents the XML Schema fragments for each of the events generated.

4.5.4.1 create-event

This event is generated when a new set of floors is created on the FC Server. This is used by the Participants to learn about new floors. The information distributed in this event is a list of floor-parameters for each of the floors created. The structure is shown in Fig. 4.25

```
<xs:element name="create-event">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-parameters" type="FloorParametersType"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

Figure 4.25 – Floor Create Event

4.5.4.2 freeze-event

This event is generated when the moderator suspends a set of floors. The information contained in this event includes a list of floor-ids that have been suspended. If there are no floors-ids then it indicates that all floors are suspended. The request structure is shown in Fig. 4.26

```
<xs:element name="freeze-event">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-id" type="FloorIDType" minOccurs="0"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

Figure 4.26 – Floor Freeze Event

4.5.4.3 remove-event

This event is generated when a moderator withdraws a floor. It contains a list of floor-ids that have been removed. The schema fragment is shown in Fig. 4.27.

```
<xs:element name="remove-event">
	<xs:complexType>
		<xs:sequence>
			<xs:element name="floor-id" type="FloorIDType" minOccurs="0"
maxOccurs="unbounded"/>
		</xs:sequence>
	</xs:complexType>
</xs:element>
```

Figure 4.27 – Floor Remove Event

4.5.4.4 modify-parameters-event

When the parameters of a floor are modified, this event is generated. The information distributed in this event is a list of the new floor parameters. The schema is shown in Fig. 4.28. It is typically generated in response to a successful modify-parameters-floors request.

```
<xs:element name="modify-parameters-event">
	<xs:complexType>
		<xs:sequence>
			<xs:element name="floor-parameters" type="FloorParametersType"
maxOccurs="unbounded"/>
		</xs:sequence>
	</xs:complexType>
</xs:element>
```

Figure 4.28 – Floor Parameters Modify Event

4.5.4.5 modify-state-event

This event is generated whenever there is a change in the state of any of the queues or the holding list of a floor. This event is typically the most frequently generated event since almost all participant requests to the server result in this event. The structure is shown in Fig. 4.29.

```
<xs:element name="modify-state-event">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="floor-state" type="FloorStateType"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

Figure 4.29 – Floor State Modify Event

4.6 Procedure rules (Time Sequence diagrams)

This section contains flow diagrams of messages/events between the conference entities. These figures are representative of the typical operations in a floor-controlled conference. In all diagrams the arrows represent a particular message or event. The contents of the message/event that are of interest are noted within brackets.

We consider a conference that has a moderator and a set of participants. The participants are allowed to request for floors but have to be approved by the moderator. To keep the scenario uncluttered, there is no pre-determination of access order and the request queue is cleared after floor allocation to a set of participants.

The following convention is used in the diagrams presented below.

1. All messages are labeled with a number.
2. All messages that form a single transaction (synchronous) have the same label number but an additional letter to denote ordering within a transaction.
3. The non-dashed arrows represent messages/events proposed in this thesis.
4. The dashed outline arrow represents a yet-to-be defined protocol.
5. The dashed arrows represent SIP signaling messages.
6. Unless otherwise noted, the order of the messages/transactions shown is significant.

The exchange of messages for a moderator creating a floor is shown in Fig.4.30. In this figure we see that during the creation of the floor, the moderator has specified that a particular user have permissions different from the currently assigned ones. Hence there is a re-INVITE in the

diagram. This re-IINVITE modifies the SDP to account for the new permissions. The ordering of (3) and (4) is not significant.

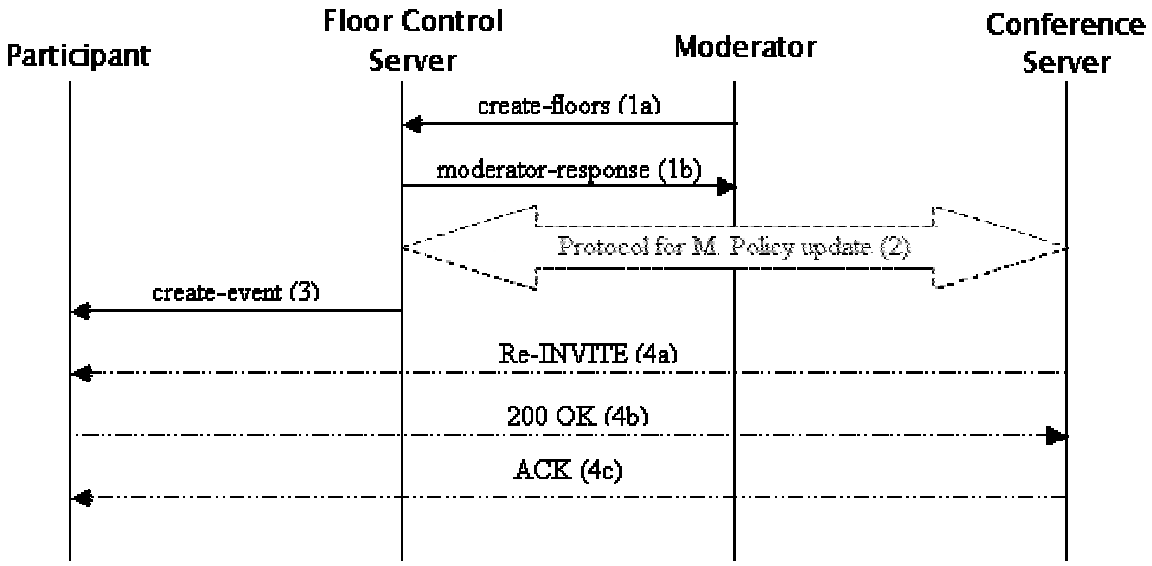


Figure 4.30 – Floor Create Procedure

The message flow, for a participant requesting a floor successfully, is shown in Fig. 4.31. Once the participant makes a request to the FC Server, the event modify-state-event with the changed request-queue parameter is used to notify the moderator that a new request has been added. The moderator then takes the appropriate decision and updates the floor state on the FC Server with the new permissions for this Participant. The FC Server then updates the Media Policy on the Conference Server and that results in a re-INVITE to the Participant to allow him access to the floor. The order of the transactions (5) and (6) may be interchanged.

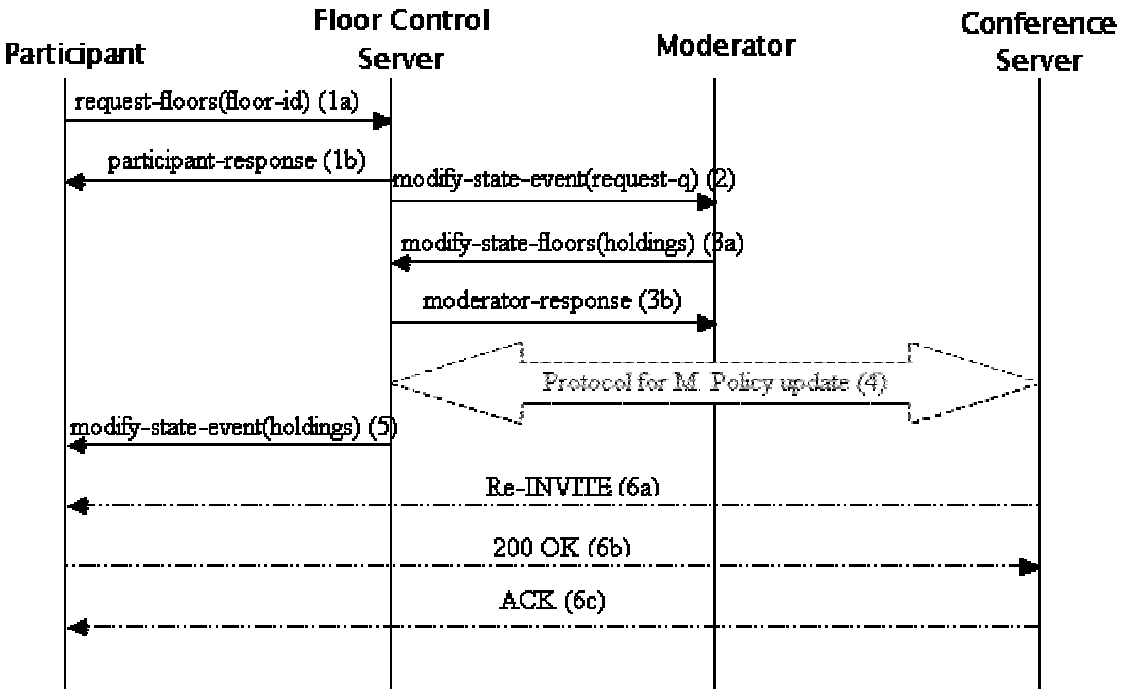


Figure 4.31 – Floor Request Procedure

Fig. 4.32 shows the process of floor release by a Participant. In this case, the Moderator is not required to release a floor. A Participant may directly relinquish the floor. Once the floor has been released, the Media Policy on the Conference Server is updated to reflect the change in the Participant's permissions and this results in a Re-INVITE. The order of (3) and (4) is not significant.

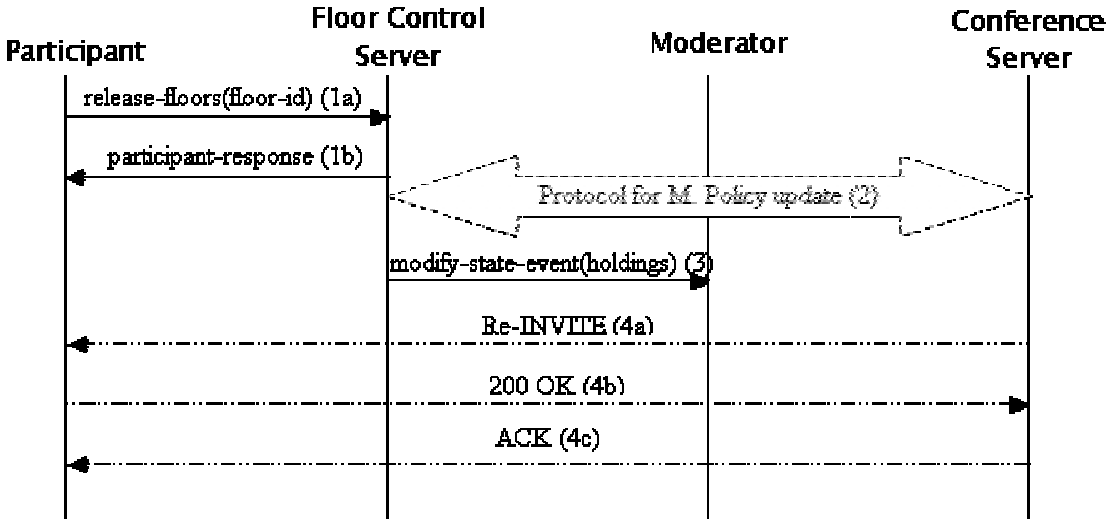


Figure 4.32 – Floor Release Procedure

The yielding of a floor to another participant is shown in Fig. 4.33. Here also there is no Moderator role as Participant A may give his own holding to Participant B, if the policies allow. The yielding of floor to Participant B results in re-INVITEs to both the Participants. The ordering of transactions (3), (4) and (5) is not significant.

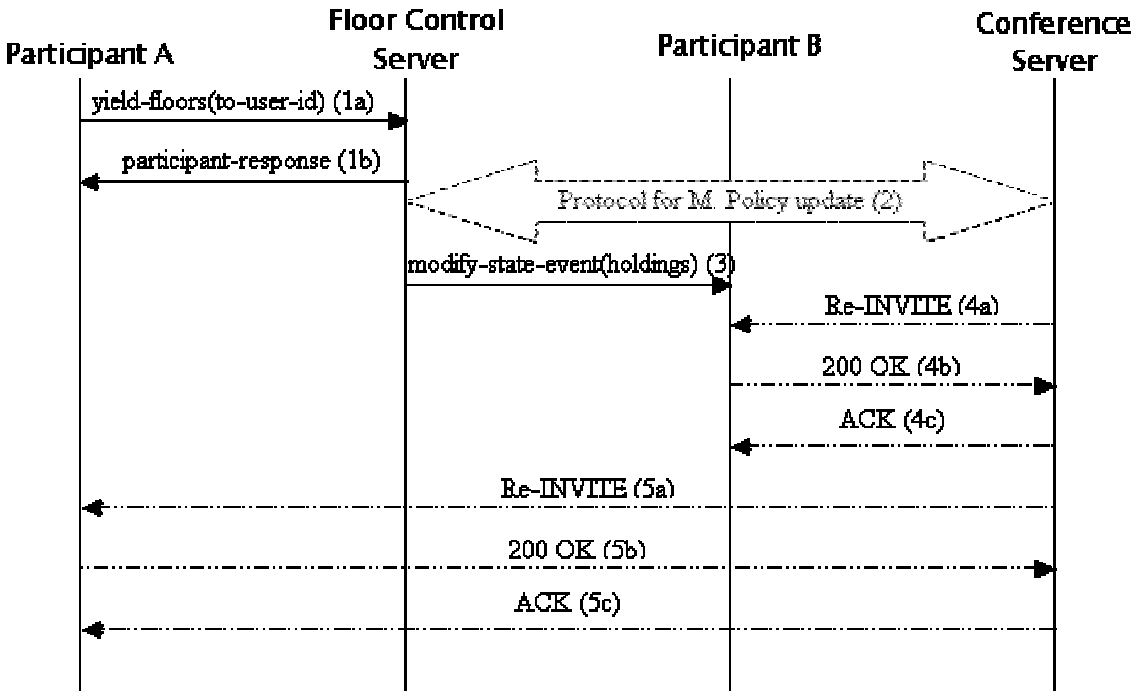


Figure 4.33 – Floor Yield Procedure

Fig. 4.34 represents the operation of canceling a request by a Participant. Once a request has been cancelled, there is no further action needed by the Moderator or the Conference Server since there is no change in Media Policy.

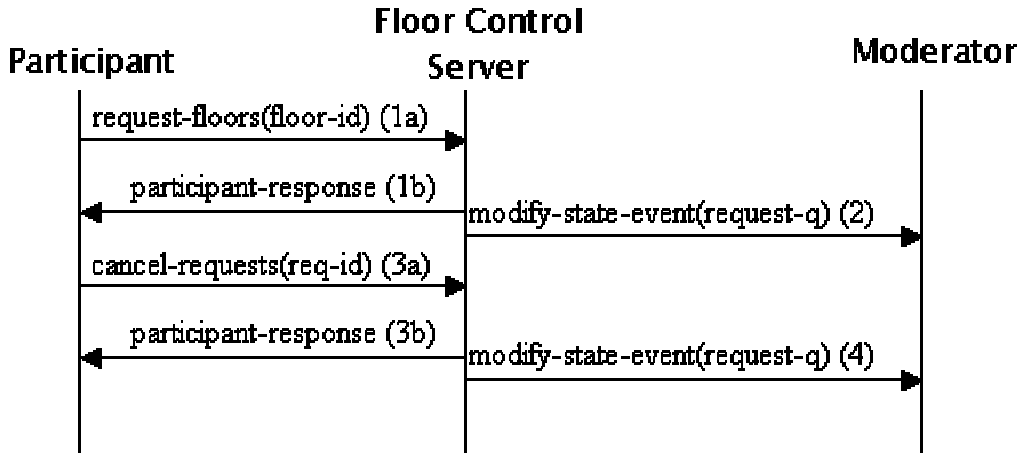


Figure 4.34 – Request Cancel Procedure

In Fig. 4.35 participant requests for floor parameters and floor state are shown. Since these never result in any changes to the floor, there are no events generated due to these messages. The transactions (1) and (2) are independent of each other.

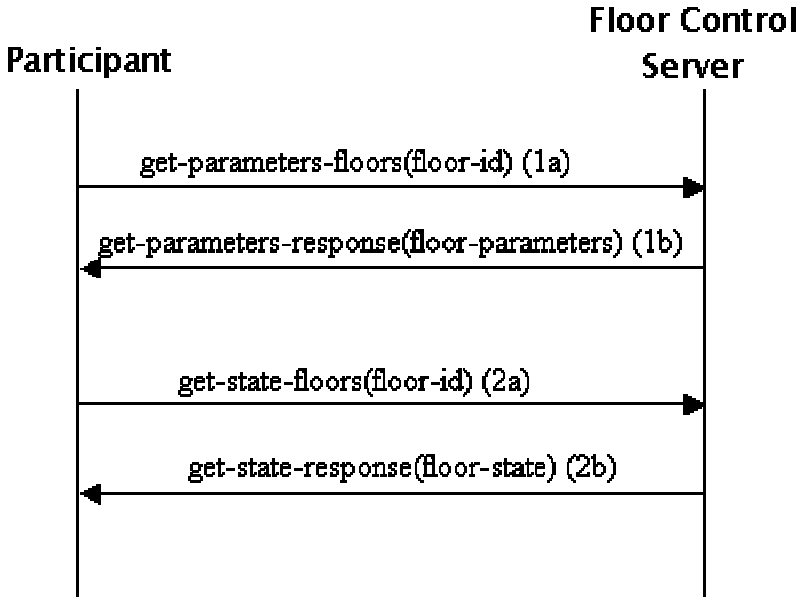


Figure 4.35 – Get Floor Parameters and State Procedures

The message exchange involved in the freezing of a floor is shown in Fig. 4.36. A Moderator freezes a floor to prevent any further requests from

Participants. Once a floor is frozen, no further requests from the Participants are queued and are directly returned a failure status.

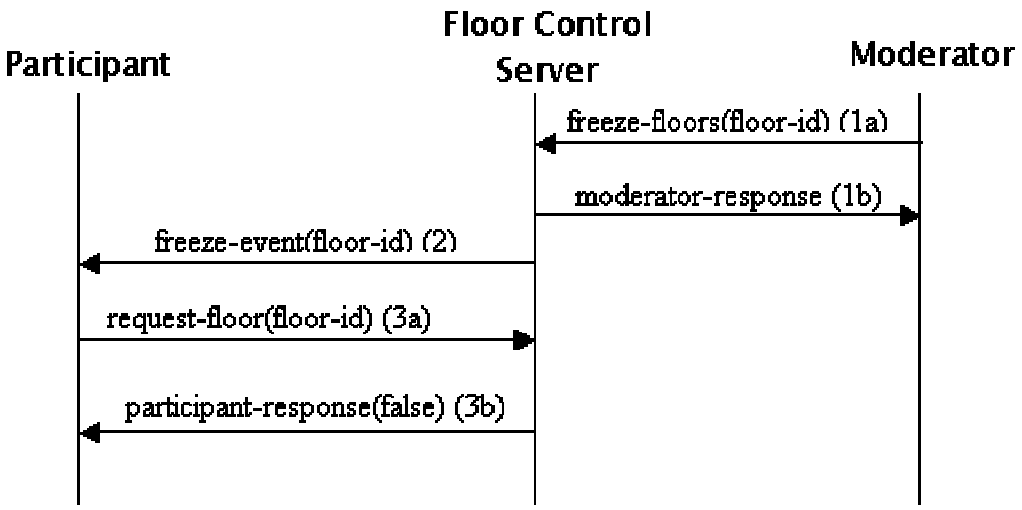


Figure 4.36 – Freeze Floor Procedure

Finally in Fig. 4.37 the process of the moderator removing a floor is shown.

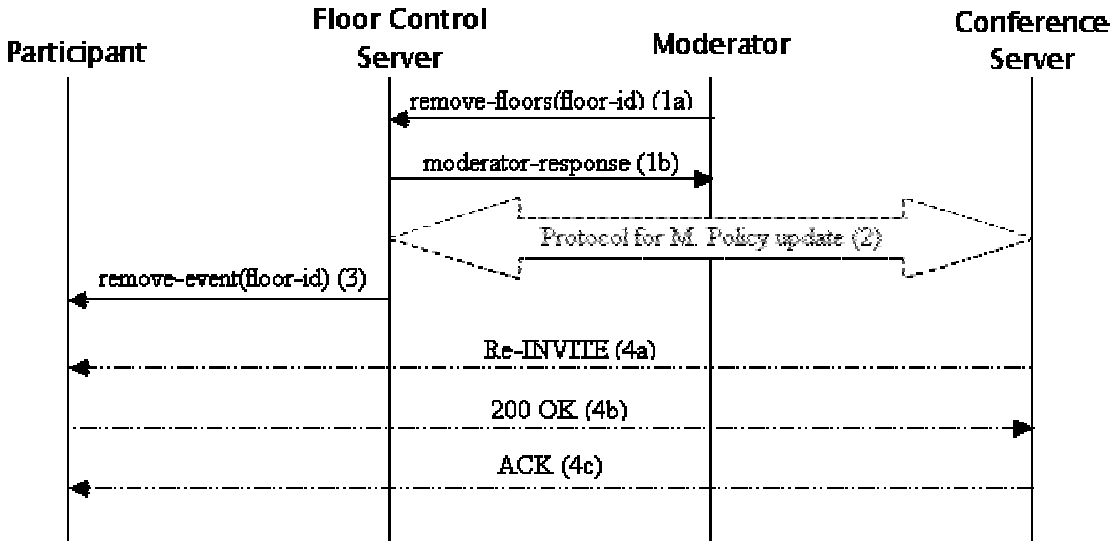
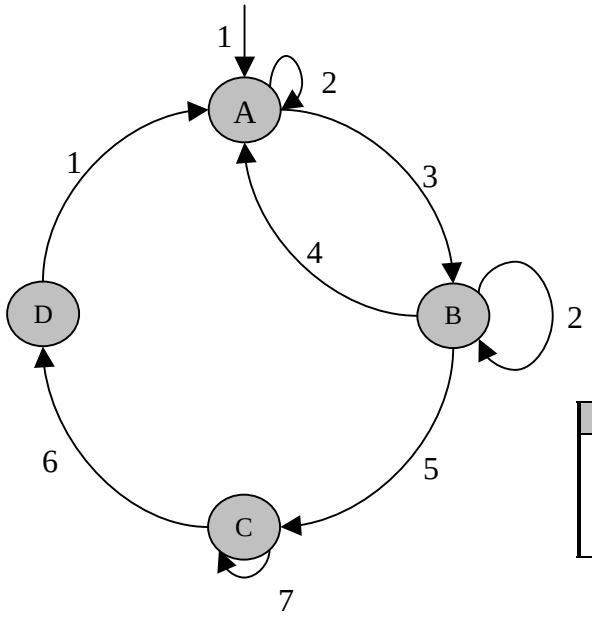


Figure 4.37 – Remove Floor Procedure

Once a floor is removed the Participants are no longer allowed to access the media and hence a re-INVITE is issued to all Participants to make the media stream unavailable. The ordering of transactions (3) and (4) is not significant.

The finite state machine for the state of the floor maintained at the FC Server is shown in Fig. 4.38. To avoid cluttering the diagram, the transitions have been labeled with numbers and their details are shown in Table 4.3.



Label	State
A	ready
B	active
C	suspended
D	removed

Figure 4.38 – Floor State Transition Diagram

Table 4.3 – Transitions

Label	Input	Output
1	create-floors	moderator-response(true), create-event
2	request-floors	participant-response(true), modify-state-event(request-q)
3	modify-floor-state(holdings)	moderator-response(true), modify-state-event(holdings)
4	release-floors	participant-response(true), modify-state-event(holdings)
5	freeze-floors	moderator-response, freeze-event
6	remove-floors	moderator-response, remove-event
7	request-floors	participant-response(false)

This concludes Chapter 4 and the presentation of the floor control protocol proposed by us. The next chapter presents the details of our implementation.

5 Implementation

This chapter provides the details of the software developed to implement the protocol outlined in the Chapter 4. It starts off by presenting the conference model and listing the components developed for the conference. The architectural overview of each of these components is presented next, followed by a description of the test setup. Next the technologies used in implementation are listed. The chapter ends with a section on the limitations of the developed software.

5.1 Conference Model

The model chosen is a conference server (Focus) with one Moderator and multiple Participants. The Participants establish SIP sessions with the Focus for the conference. The Moderator doesn’t have to be a part of the conference but is able to communicate with the Floor Control Server function of the Focus. This is in agreement with requirements in [ID REQ-01]. Fig. 5.1 shows the components developed in the conference model.

Focus - includes the SIP stack to handle sessions. It is the backbone of the conference and waits for connections from the Participants. The FC Server function is also implemented within the Focus. The term “Focus”, as used in the implementation is a misnomer because it refers to the complete Conference Server discussed in Chapter 3.

Moderator – is an implementation of the floor control functions of the moderator. It does not include a SIP stack. However, it is capable of registering itself with the FC Server to receive event notifications related to the floor. A GUI is provided to ease interaction with the FC Server function.

Participant – also has a SIP stack to communicate with the Focus. It also includes an implementation of the participant floor control operations. It is capable of receiving floor control events by registering itself with the FC Server. A GUI is provided to ease interaction with the Focus and to make floor requests.

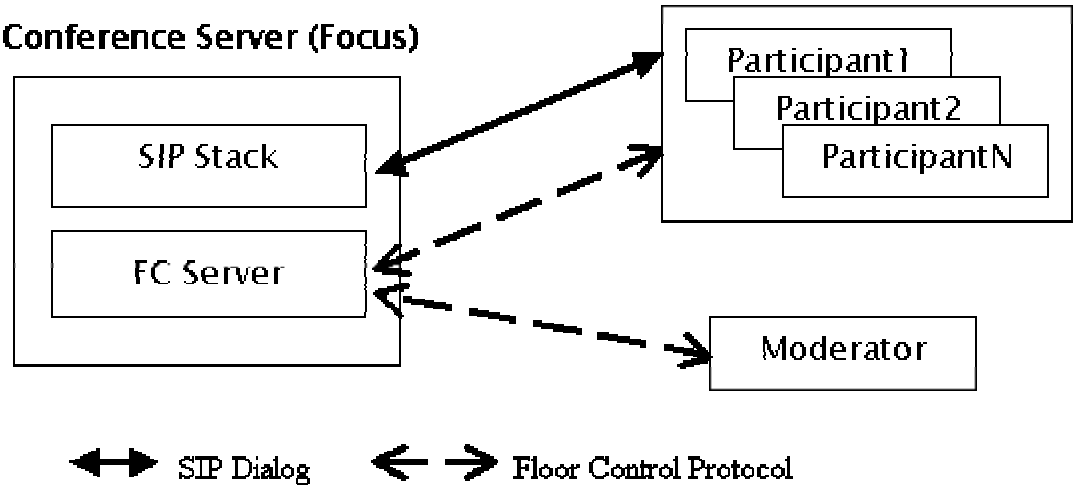


Figure 5.1 – Conference Model

5.2 Architecture of Components

This section provides the high level architecture of the sub-components of the various entities of the conference.

5.2.1 Focus

The Focus forms the hub of the conference. It has the functionality to handle floor control requests from the moderator as well as the participants. Also, participants setup SIP sessions with the Focus. These sessions are described using SDP bodies and as and when the session parameters for a particular Participant change (e.g. granting of access to a media channel by the moderator), the Focus re-INVITES the Participant with the modified SDP.

The major components of the Focus are the SIPUA, FC Server, SDP Manager and a primitive User Interface. The design of the Focus is shown in Fig. 5.2.

The **SIPUA** is a user agent (UA) developed on top of the NIST SIP Stack. This UA is used to manage SIP sessions with participants. The entry point into the UA is the CallManager. It maintains a set of CallLeg entities. Each CallLeg represents a signaling relationship with a Participant in the SIP Stack domain. When setting up a CallLeg, the UA interfaces with the SDP Manager to obtain a copy of the SDP Announce body to be used for this participant. After this, whenever there is a change in sdp body of a CallLeg, the SDP Manager is queried to obtain the latest copy of the sdp and this is then used in the re-INVITE sent to the participant. Currently the implementation supports one conference with multiple participants.

The **SDP Manager** function is shared between the UA and the FC Server. It manages all sdp bodies for the conference. The sdp bodies are indexed by the SIP URI of the participant. The FC Server updates the permissions in the sdp of a particular user using the interface provided by SDP Manager.

The **FC Server** function in the Focus handles all floor control related messages. It consists of two sub-components, the FCServerProtocolHandler to handle methods from the moderator and the participants and an FCEventNotifier to relay events to the parties that are subscribed with it. This function maintains all the floor related information such as floor parameters and floor states.

The **User Interface** classes of the Focus maintain a set of Calls. Each Call represents a SIP signaling relationship with a participant in the user interface domain. The Call and the CallLeg are a one to one mapping. This is a text based UI.

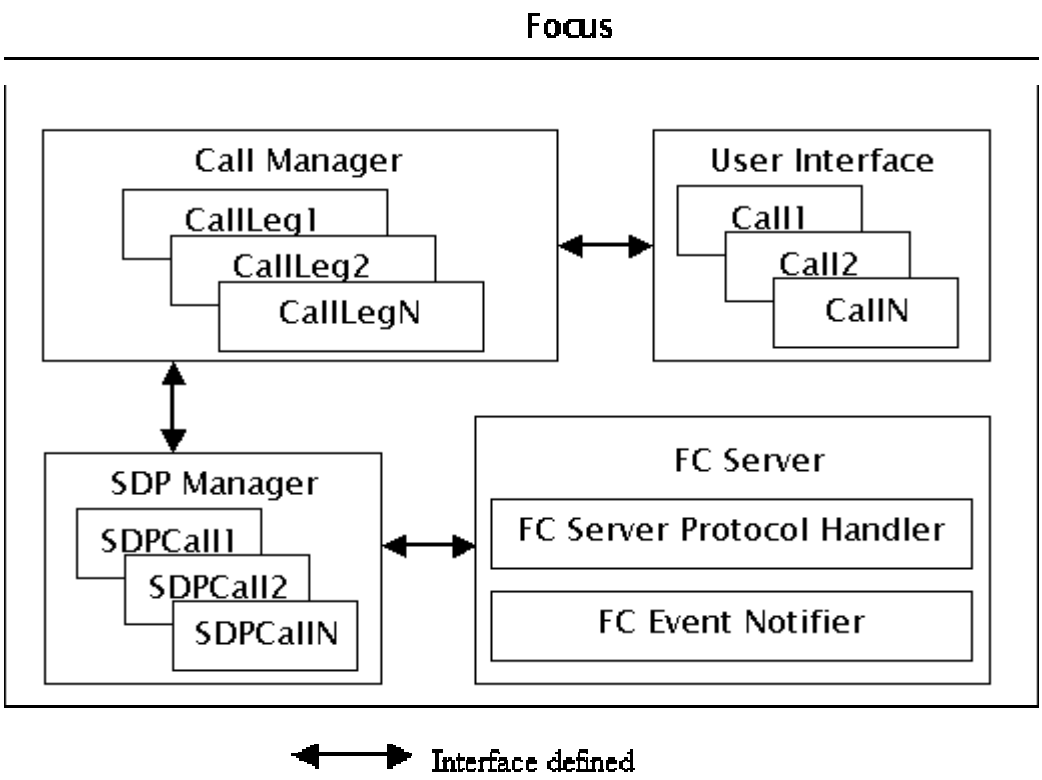


Figure 5.2 – Focus Design

5.2.2 Moderator

The Moderator is in charge of certain floors and all requests to access the floor are to be approved by this role before the access is granted. Whenever a request comes in from the Participant to the Focus, it is forwarded to the Moderator. The Moderator is also in charge of creating, freezing and removing floors.

The Moderator implementation consists of two sub-components, the User Interface and the Floor Control function. It does not contain a SIP Stack within it as it communicates purely with the FC Server function of the Focus.

The **Floor Control** function of the Moderator supports create-floors and modify-state-floors command set of the protocol. Each moderator request is handled in a new FCTransactionThread. There is an FCEventListenerThread that waits for event notifications from the FC Server. Any updates in the

status if requests (such as an incoming response or event) are notified to the UI using the UIUpdater Interface.

The **User Interface** is implemented in the ModeratorUI. It has components to display the list of floors, the floor parameters, the floor states as well as other information such as floor events received. It includes the logic to create the floor control requests based on user input and then gives this to the FCTransactionThread to transmit and return the result. The result is processed and the UI elements are updated to reflect the result.

The above sub-components are shown in Fig. 5.3.

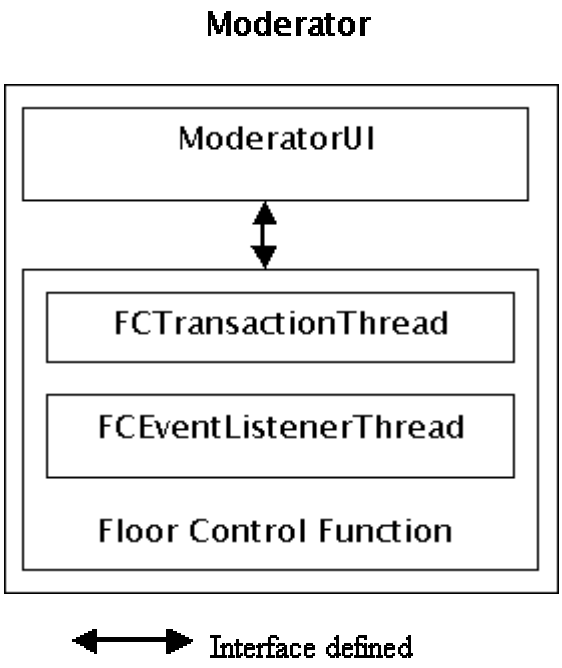


Figure 5.3 – Moderator Design

5.2.3 Participant

The Participant is the component that makes requests for floors to the Focus. If the floor is moderated, then the request is to be approved by a Moderator before granting access to the floor. The Participant implementation consists of three sub-components – SIPUA, Floor Control and User Interface. A block diagram, of the Participant sub-components, is shown in Fig. 5.4.

The **SIPUA** sub-component is similar to the one developed for the Focus. The entry point into the UA is the CallManager. The CallManager of the Participant is equipped to handle only one CallLeg, since it is assumed that the Participant is taking part in only one conference at a time. Other differences are that the CallManager of the Participant is capable of sending new INVITES while the Focus can only receive new INVITEs. Similarly, the re-INVITEs can only be issued by the Focus, while the Participant may only receive re-INVITEs.

The **Floor Control** function of the Participant is implemented within FCManagerThread and FCEventListenerThread. The operations that have been implemented are “request-floors” and “release-floors”. The Participant is capable of receiving events from the Focus as well as sending the above commands.

The **User Interface** is implemented in the ParticipantUI class. It consists of UI elements to display a list of floors and to display other floor control information such as events received, etc. There are methods to join a conference, accept re-INVITEs and leave the conference. As in the Focus, a Call represents the details of the SIP session in the user interface domain.

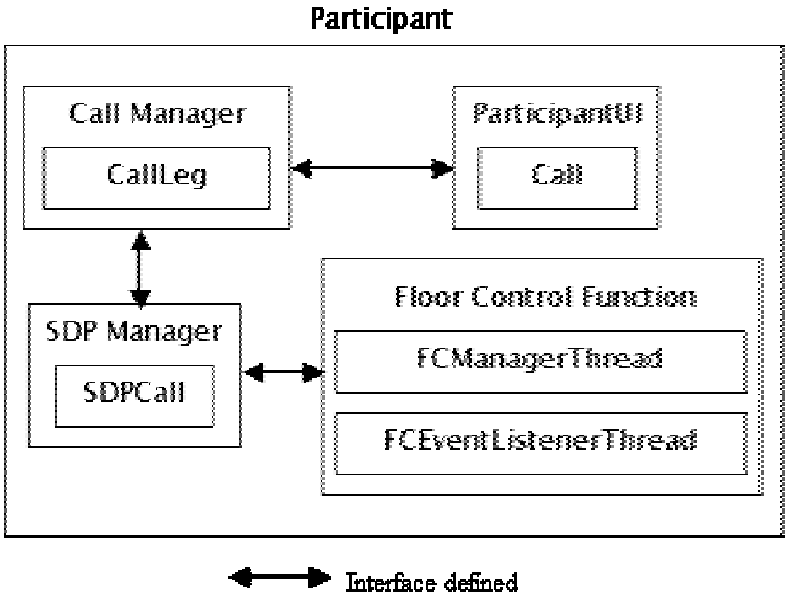


Figure 5.4 – Participant Design

5.3 Experimental Setup

The software components developed were setup and tested in the sequence of events given below. There is one Focus (server@host:5060/tcp), one Moderator (moderator@host/tcp) and two Participants (user1@host:6000/tcp and user2@host:7000/tcp). The FC Server of the Focus is available at (server@host:5062/tcp).

Note that these steps have been discussed in Chapter 3 where the conference setup and the request flows for the moderator and the participants have been presented in a generic scenario.

1. First the Focus is setup to receive SIP requests and FC Server requests. The SDP for the conference is assumed to be already present on the Focus. Currently this is done using a startup parameter to the Focus.

2. The moderator registers with the FC Server to receive events.

3. The Participants setup SIP sessions with the Focus. In this session setup, the SDP exchanged lists the floors and the address of the FC Server to send their floor requests to.

4. The Participants register with the FC Server to receive floor control events.

5. The Moderator then sends a “create-floors” command to the FC Server function on the Focus to create a floor for “floor1”. The FC Server processes this request and returns a success indication if the floor was created. Also, the FC Server sends out the “create-event” to the Participants.

6. Now the Participant may request for the floors that are created on the FC Server.

7. Participant requests to the FC Server are forwarded to the Moderator using the modify-state-event.

8. The Moderator then approves or rejects the request using the modify-state-floors command. The result is then conveyed to the participant using the modify-state-event.

9. Also, the Focus may need to re-INVITE the participant based on whether she/he has access to the media floor or not.

10. A Participant may release a floor that she/he is holding using the “release-floors” command. This command is executed at the FC Server and it is not forwarded to the Moderator for approval.

11. Finally, the participants hang-up using BYE to end the conference.

5.3.1 Sample flows

This sub-section presents the details of the tests conducted to verify the working of the implementation. In the test setup, the following SDP body is used as the initial session description.

```
v=0
o=server 0001 0002 IN IP4 152.14.51.157
t=0 0
s=A conference
c=IN IP4 152.14.51.157
a=group:FC fcchannel1 floor1
m=audio 5080 RTP/AVP 0
a=mid:floor1
m=application 5070 udp wb
a=mid:nofloor
m=control 5062 TCP dc
a=mid:fcchannel1
```

It consists of two media streams, one audio stream (floor1) that is moderated and one data stream (nofloor) that is unmoderated. Note that the floor control channel (fcchannel1) line gives the connection information of the FC Server.

5.3.1.1 create-floors

This sub-section presents the floor control messages and events that were observed when the moderator created a floor for the audio session (floor1). For a message level overview of the flows in this sub-section, please refer to Fig. 4.30.

STEP 1 – Moderator -> FC Server

```
<create-floors req-id="1">
      <floor-parameters floor-id="floor1">
            <description>Voice Channel for employees</description>
            <pre-emptible>false</pre-emptible>
            <access-queue-parameters>
                  <max-size>1</max-size>
            </access-queue-parameters>
            <request-queue-parameters>
                  <max-size>10</max-size>
            </request-queue-parameters>
            <max-holders>1</max-holders>
```

```
            <moderator-list>
                  <moderator>moderator@152.14.51.157</moderator>
            </moderator-list>
            <user-permissions-list>
                  <user-permissions>
                        <user>*</user>
                        <permissions>recvonly</permissions>
                  </user-permissions>
            </user-permissions-list>
      </floor-parameters>
</create-floors>
```

STEP 2 - FC Server -> Moderator

```
<moderator-response req-id="1">
      <floor-response floor-id="floor1">
            <status>true</status>
            <status-phrase>Successfully created</status-phrase>
      </floor-response>
</moderator-response>
```

STEP 3 – FC Server -> Participants

```
<create-event>
      <floor-parameters floor-id="floor1">
            <description>Voice Channel for employees</description>
            <pre-emptible>false</pre-emptible>
            <access-queue-parameters>
                  <max-size>1</max-size>
            </access-queue-parameters>
            <request-queue-parameters>
                  <max-size>10</max-size>
            </request-queue-parameters>
            <max-holders>1</max-holders>
            <moderator-list>
                  <moderator>moderator@152.14.51.157</moderator>
            </moderator-list>
            <user-permissions-list>
                  <user-permissions>
                        <user>*</user>
                        <permissions>recvonly</permissions>
                  </user-permissions>
            </user-permissions-list>
      </floor-parameters>
</create-event>
```

STEP 4 - Focus -> Participants

```
v=0
o=server 0001 0002 IN IP4 152.14.51.157
t=0 0
s=A conference
c=IN IP4 152.14.51.157
a=group:FC fcchannel1 floor1
m=audio 5080 RTP/AVP 0
a=mid:floor1
a=recvonly
```

```
m=application 5070 udp wb
a=mid:nofloor
m=control 5062 TCP dc
a=mid:fcchannel1
```

Re-INVITEs were issued to the Participants with the above SDP body. The modifications to the initial SDP are shown in *red italicized* lettering.

5.3.1.2 request-floors

This sub-section presents the floor control messages and events that were observed when a Participant (emp1) requested for a floor (floor1). For a message level overview of the flows in this sub-section, please refer to Fig. 4.31.

STEP 1 - Participant -> FC Server

```
<request-floors req-id="100">
      <user-id>sip:emp1@152.14.51.157:6000</user-id>
      <floor-id-list>
            <floor-id>floor1</floor-id>
      </floor-id-list>
</request-floors>
```

STEP 2 - FC Server -> Participant

```
<participant-response req-id="100">
      <status>true</status>
</participant-response>
```

STEP 3 – FC Server -> Moderator

```
<modify-state-event>
      <floor-state floor-id="floor1">
            <request-queue-state>
                  <size>1</size>
                  <request-list>
                        <request-floors req-id="100">
                              <user-id>sip:emp1@152.14.51.157:6000</user-
id>
                              <floor-id-list>
                                    <floor-id>floor1</floor-id>
                              </floor-id-list>
                        </request-floors>
                  </request-list>
            </request-queue-state>
      </floor-state>
</modify-state-event>
```

STEP 4 - Moderator -> FC Server

```
<modify-state-floors req-id="3">
      <floor-state floor-id="floor1">
            <holding-list>
                  <holding>
                        <user-id>sip:emp1@152.14.51.157:6000</user-id>
                        <floor-id>floor1</floor-id>
                  </holding>
            </holding-list>
      </floor-state>
</modify-state-floors>
```

STEP 5 - FC Server -> Moderator

```
<moderator-response req-id="3">
      <floor-response floor-id="floor1">
            <status>true</status>
      </floor-response>
</moderator-response>
```

STEP 6 – FC Server -> Participants

```
<modify-state-event>
      <floor-state floor-id="floor1">
            <holding-list>
                  <holding>
                        <user-id>sip:emp1@152.14.51.157:6000</user-id>
                        <floor-id>floor1</floor-id>
                  </holding>
            </holding-list>
      </floor-state>
</modify-state-event>
```

STEP 7 - Focus -> Participant (emp1)

```
v=0
o=server 0001 0002 IN IP4 152.14.51.157
t=0 0
s=A conference
c=IN IP4 152.14.51.157
a=group:FC fcchannel1 floor1
m=audio 5080 RTP/AVP 0
a=mid:floor1
a=sendrecv
m=application 5070 udp wb
a=mid:nofloor
m=control 5062 TCP dc
a=mid:fcchannel1
```

Re-INVITE was issued to the Participant (emp1@host:6000). The modifications from the previous SDP are shown in *red italicized* lettering.

5.3.1.3 release-floors

This sub-section presents the floor control messages and events that were observed when the Participant (emp1) released the floor (floor1). For a message level overview of the flows in this sub-section, please refer to Fig. 4.32.

STEP 1 - Participant -> FC Server

```
<release-floors req-id="101">
      <user-id>sip:emp1@152.14.51.157:6000</user-id>
      <floor-id-list>
            <floor-id>floor1</floor-id>
      </floor-id-list>
</release-floors>
```

STEP 2 - FC Server -> Participant

```
<participant-response req-id="101">
      <status>true</status>
</participant-response>
```

STEP 3 – FC Server -> Moderator and Participants

```
<modify-state-event>
      <floor-state floor-id="floor1">
            <holding-list/>
      </floor-state>
</modify-state-event>
```

STEP 4 – Focus -> Participant (emp1)

```
v=0
o=server 0001 0002 IN IP4 152.14.51.157
t=0 0
s=A conference
c=IN IP4 152.14.51.157
a=group:FC fcchannel1 floor1
m=audio 5080 RTP/AVP 0
a=mid:floor1
a=recvonly
m=application 5070 udp wb
a=mid:nofloor
m=control 5062 TCP dc
a=mid:fcchannel1
```

Re-INVITE was issued to the Participant (emp1@host:6000). The modifications from the previous SDP are shown in *red italicized* lettering.

5.4 Software Tools

The software was developed on a Windows 2000 machine. During the implementation, many useful tools and technologies were utilized. These are listed in Table 5.1.

Table 5.1 – Software Tools

Technologies/Tools	Description
Java Development Kit (JDK)	The entire project is implemented using this freeware toolkit (jdk1.3.1). It can be found at (http://java.sun.com/)
Java API for XML Binding (JAXB)	The packages used to manipulate the XML data belong to the (JAXB v1.0 reference implementation) provided by Sun. It can be found at (http://java.sun.com/xml/jaxb/)
NIST SIP Stack	The open source project that implements a SIP Stack and provides packages to parse SDP messages. It can be found at (http:// www.antd.nist.gov/proj/iptel/)
The bonephone project	A large part of the SIPUA code has been modified from this public domain SIP phone. It can be found at (http://sourcewell.berlios.de/appbyid.php3?id=1897)
XMLSpy	A tool to create and manipulate XML including XML Schemas and documents. It has proved invaluable in the initial design of the protocol messages. An evaluation copy can be downloaded at (http://www.altova.com)
cygwin	A Unix wrapper for Windows. It provides a shell interface to windows and a number of common Unix tolls like make and vi. It can be found at (http://www.cygwin.com/)

5.5 Limitations

This sub-section lists the limitations of the current implementation.

1. A single conference is supported.
2. The full set of protocol functions is not implemented. The list of unimplemented functions is shown in Table
3. Limited UI functionality
4. There is no media support since media engines are not developed.
5. Event Notification mechanism is a simplified implementation of the SIP event notification mechanism since the NIST SIP stack used doesn’t support SUBSCRIBE/NOTIFY.

Table 5.2 – List of functions not yet implemented

Roles	Functions
Moderator	freeze-floors
	remove-floors
	modify-parameters-floors
Participant	yield-floors
	cancel-requests
	get-parameters-floors
	get-state-floors
Events	freeze-event
	remove-event
	modify-parameters-event

This concludes our discussion on the components developed while implementing the protocol. The next chapter concludes the work done in this thesis by summarizing the contributions and suggesting future work.

6 Summary and Future Work

This chapter starts off by listing some basic differences in approach between the work in this thesis and work published in [ID CONTROL-02]. These are primarily due to the fact that this work was done in parallel with [ID CONTROL-02]. The contributions made by this thesis are then summarized. Finally future directions for this work are discussed.

6.1 Differences in approach

The use of one floor per media is assumed in this work. It provides means to group these floors to denote synchronization between floors. This, the author believes, is a more extensible approach as compared to current work [ID CONTROL-02] that allows for multiple media streams per floor.

To explain more clearly, each media stream is labeled using an identifier specified in SDP. If a floor represents a media, then the same identifier can be used to access the floor. In the case of multiple media per floor, there is no concept of a “floor identifier” in [ID CONTROL-02]. This means that a request has to explicitly list each of the resources contained in the floor. So there is no advantage of this grouping, in fact this adds additional semantic overhead to the requests in some cases, as explained below.

Assume that a Participant wishes to request for three media streams, say X, Y and Z together, i.e., in an “all or none” configuration. Also assume that ModeratorA manages X and Y while ModeratorB is in charge of Z. In [ID COTROL-02], the semantics of the requests (such as request-floor) are relied on, to enforce that all media in a floor are allotted together. So in the above case, since Z is not in the same floor, the meaning of the request needs to be modified to allow for such Participant preferences. In our proposal, since the request signifies that the Participant needs all of the enclosed media streams atomically, there is no change in message semantics to handle special cases like this one.

6.2 Protocol Enhancements

The protocol proposed in this thesis has made contributions, to the problem of floor control in SIP conferences, in three major areas – new functionality, clearer definitions and support for a wider variety of floor control policies.

6.2.1 New Functionality

Participant can use the “yield-floors” request to give the floor to another Participant. This functionality is not currently addressed by [ID CONTROL-02]. It is important to achieve atomically a transfer of floor from one current holder to a participant. In the existing proposal, a participant has no control over the next holder of the floor.

Also a request, “cancel-requests”, to enable Participants to cancel an older request is added. This is most useful in situations where the Participant wants to cancel his position in the access queue. The functionality of canceling a request is not addressed in existing work.

Also, new requests to get floor parameters (“get-parameters-floors”) and floor state (get-state-floors”) have added. This is to enable a Participant to pro-actively get floor related information. This is useful for participants that would not prefer getting a notification for each change on the floor. They can periodically poll the FC Server to receive this information.

A new request for moderators, “freeze-floors”, is introduced. This marks a floor to prevent any further requests for a floor. This is useful in cases where the moderator plans to withdraw a floor from the conference and would like the current holders to complete their usage of the floor. In existing work, the floor has to abruptly withdraw by a moderator, relying on in-band means such as a voice broadcast to forewarn participants of an impending floor removal.

While creating/updating a floor, moderators can now specify user access permissions for the floor, there was not possible in [ID CONTROL-02]. The existing draft intrinsically assumes that a holder of a floor has send and

receive permissions on a stream and that all other Participants have receive only permissions. While this is intuitive for audio/video streams, there may be cases where a participant is not allowed to access the media stream even in receive only mode. Using of explicit permissions solves this problem.

6.2.2 Structural Modifications

There are several changes in the structure definitions of floors, holdings and queues. Firstly floor information is classified into two – floor parameters and floor state. A significant addition to floor parameters is the property of pre-emptibility.

In floor state, a differentiation has been made between a queue specifying the order of access and a queue representing incoming participant requests. This allows for policies such as the moderator pre-determining the order of floor access and then freezing the floor. So no new requests are entertained and the selected ones are completed.

As with the floor, queue information has been classified into queue parameters and queue state. A max-queue-size parameter has been introduced to control queue size.

6.2.3 Policies

The protocol has been designed with a variety of policies in mind, a discussion on these policies is given in Section 4.1. One special case is of the “no floor” policy for media steams. This means creating floors only for required resources and leaving the rest with free access. This reduces the overhead, of creating and requesting for floors, incurred with the approach in [ID CONTROL-02].

6.3 Future Work

In its current state, the protocol does not support primitives to manipulate the order of requests in queues on the FC Server. This may lead to large data exchanges between the FC Server and the Moderator since the entire queue needs to be sent to the Moderator, who then locally

manipulates the order and then rewrites the queue state back to the server.

New primitives are required to support the more formal meeting procedures such as the ability of a participant to request a holder to yield his current holding. Also a new primitive to allow a moderator to activate a floor after freezing it would be useful.

Currently, to mark synchronization between floor states an additional semantic with the “create-floors” message is used. All floors created within one “create-floors” request are considered synchronized and a Participant must ask for these floors together. This additional meaning to the request should be removed by providing an explicit grouping primitive.

A clearer separation between logic to implement policies with the mechanisms in place is desirable. This enables support of on-the-fly policy changing.

To be able to detect/extract and eliminate subtle protocol errors, such as deadlock situations and race conditions, a formal analysis of the protocol needs to be done using the available tools for protocol validation and correctness verification such as finite state machine analysis.

Security requirements of the protocol need to be extracted and made explicit so that they may be mapped out onto transport protocols capabilities. Currently, security of floor control messages is not handled and is left to the transport layer.

7 Bibliography

[Crowley 90] T. Crowley, P. Milazzo, E. Baker, H. Forsdick and R. Tomlinson, “MMConf: an infrastructure for building shared multimedia applications”. Proc. Conference on Computer Supported Cooperative Work, ACM SIGCHI & SIGOIS, Los Angeles, October 1990.

[Dommel 97] Hans-Peter Dommel, J.J. Garcia-Luna-Aceves, “Floor Control for multimedia conferencing and collaboration.” Multimedia Systems, 1997.

[Dommel 98] Hans-Peter Dommel, J.J. Garcia-Luna-Aceves, “A Novel Group Coordination Protocol for Collaborative Multimedia Systems”.

[Ellis 91] C.A. Ellis, S.J. Gibbs, and G.L. Rein. “Groupware – Some Issues and Experiences”. Communications of the ACM, 34(1):39-58, January 1991.

[Greenberg 91] S. Greenberg, “Personalizable Groupware: Accomodating individual roles and group differences”. In Proceedings of the European Conference of Computer Supported Cooperative Work (ECCSCW’91), pp.17-32, Amsterdam, September 24-27, Kluwer Academic Press.

[Hishii 90] H. Hishii, “TeamWorkStation : Towards a seamless shared workspace”, Proc. Conference on Computer Supported Cooperative Work, ACM SIGCHI & SIGOIS, Los Angeles, October 1990, pp 13-26.

[ID CONFARCH-03] M. Handley, J. Crowcroft, C. Bormann and J. Ott, “The Internet Multimedia Conferencing Architecture”. Internet Draft, Internet Engineering Task Force, July 2000. Expired.

[ID CONTROL-02] X. Wu, P. Koskelainen, H. Schulzrinne and C.C. Chen, ” Use SIP and SOAP for Conference Floor Control”. Internet Draft, Internet Engineering Task Force, February 2002. Work in progress.

[ID FID-06] H. Schulzrinne, G. Eriksson, G. Camarillo and J. Holler, “Grouping of media lines in SDP”. Internet Draft, Internet Engineering Task Force, February 2002. Work in progress.

[ID FRAMEWORK-00] J. Rosenberg, “A Framework for Conferencing with the Session Initiation Protocol”. Internet Draft, Internet Engineering Task Force, October 2002. Work in progress.

[ID MODELS-01] J. Rosenberg and H.Schulzrinne, “Models for Multi Party Conferencing in SIP”. Internet Draft, Internet Engineering Task Force, July 2002. Work in progress.

[ID REQ-01] H. Schulzrinne, J. Ott and P. Koskelainen, “Requirements for Floor Control”. Internet Draft, Internet Engineering Task Force, November 2002. Work in progress.

[IEC REP-01] A report titled “The Market for IP Enhanced Services”. International Engineering Consortium (www.iec.org).

[IMTC] International Multimedia Telecommunications Consortium (www.imtc.org).

[ITU H323] http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-H.323

[Kamel 93] N. Kamel, “An Integrated Approach to Shared Synchronous Groupware Workspaces”. IEEE, 1993.

[Malone 90] T. W. Malone and K. Crowston, “What is coordination theory and how can it help design cooperative work systems?”. Proc. Conference on Computer Supported Cooperative Work, ACM SIGCHI & SIGGROUP, Los Angeles, October 1990.

[RFC 768] J. Postel, “User Datagram Protocol”. Request For Comments, Internet Engineering Task Force, August 1980.

[RFC 791] J. Postel, “Internet Protocol”. Request For Comments, Internet Engineering Task Force, September 1981.

[RFC 2205] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog and S. Jamin, “Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification”. Request For Comments, Internet Engineering Task Force, September 1997.

[RFC 2208] A. Mankin, Ed., F. Baker, B. Braden, S. Bradner, M. O`Dell, A. Romanow, A. Weinrib and L. Zhang, “Resource ReSerVation Protocol (RSVP) -- Version 1 Applicability Statement Some Guidelines on Deployment”. Request For Comments, Internet Engineering Task Force, September 1997.

[RFC 2209] R. Braden and L. Zhang, “Resource ReSerVation Protocol (RSVP) -- Version 1 Message Processing Rules”. Request For Comments, Internet Engineering Task Force, September 1997.

[RFC 2821] J. Klensin, “Simple Mail Transfer Protocol”. Request For Comments, Internet Engineering Task Force, April 2001.

[RFC 1889] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications”. Request For Comments, Internet Engineering Task Force, January 1996.

[RFC 2327] M. Handley and V.Jacobson, “SDP: Session Description Protocol”. Request For Comments, Internet Engineering Task Force, April 1998. Standards Track.

[RFC 2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1”. Request For Comments, Internet Engineering Task Force, June 1999. Standards Track.

[RFC 2974] M. Handley, C. Perkins, E. Whelan, “Session Announcement Protocol”, Request for Comments, Internet Engineering Task Force, October 2000. Experimental.

[RFC 3261] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley and E. Schooler, “SIP: Session Initiation Protocol”. Request For Comments, Internet Engineering Task Force, June 2002. Standards Track.

[RFC 3265] A. B. Roach, “Session Initiation Protocol (SIP)-Specific Event Notification”. Request For Comments, Internet Engineering Task Force, June 2002. Standards Track.

[Sarin 85] S. Sarin and I. Greif, ”Computer based real-time conferences”. IEEE Computer, October 1985

[TS 23-228] “IP Multimedia Subsystem (IMS), Stage 2, Release 5”. Technical Specification, 3rd Generation Partnership Project, September 2002.

[VIDE] Video Development Initiative (www.videnet.gatech.edu/cookbook).

[W3C SOAP-00] N. Mitra, “*SOAP Version 1.2 Part 0: Primer”*, Working Draft, World Wide Web Consortium, June 2002.

[W3C SOAP-01] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, H. F. Nielsen, “*SOAP Version 1.2 Part 1: Messaging Framework”*, Working Draft, World Wide Web Consortium, June 2002.

[W3C SOAP-02] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, H. F. Nielsen, “*SOAP Version 1.2 Part 2: Adjuncts”.* Working Draft, World Wide Web Consortium, June 2002.

Appendix I – XML Schema

A Listing of the Floor Control Protocol XML Schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 2 U (http://www.xmlspy.com) by Prashant
Gupta (North Carolina State Univ) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:jxb="http://java.sun.com/xml/ns/jaxb" elementFormDefault="qualified"
attributeFormDefault="unqualified" jxb:version="1.0">
<xs:annotation>
    <xs:appinfo>
        <jxb:globalBindings collectionType="java.util.Vector"/>
    </xs:appinfo>
</xs:annotation>
<xs:simpleType name="FloorIDType">
    <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:complexType name="FloorIDsType">
    <xs:sequence>
        <xs:element name="floor-id" type="FloorIDType"
maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:simpleType name="UserIDType">
    <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="PermissionsType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="sendonly"/>
        <xs:enumeration value="recvonly"/>
        <xs:enumeration value="sendrecv"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestIDType">
    <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:complexType name="HoldingType">
    <xs:sequence>
        <xs:element name="user-id" type="UserIDType"/>
        <xs:element name="floor-id" type="FloorIDType"/>
        <xs:element name="start-time" type="xs:dateTime" minOccurs="0"/>
        <xs:element name="end-time" type="xs:dateTime" minOccurs="0"/>
        <xs:element name="permissions" type="PermissionsType"
default="sendrecv" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="HoldingsType">
    <xs:sequence>
        <xs:element name="holding" type="HoldingType" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="QueueParametersType">
    <xs:sequence>
        <xs:element name="max-size" type="xs:integer"/>
```

```
    </xs:sequence>
</xs:complexType>
<xs:complexType name="QueueStateType">
    <xs:sequence>
        <xs:element name="size" type="xs:integer"/>
        <xs:element name="request-list"
type="ParticipantRequestListType"/>
    </xs:sequence>
</xs:complexType>
<!--Modified it to support multiple FloorIDs in one request leading to the
all or nothing semantics. To achieve a per floor request semantic, a
seperate request needs to exist for each floor.-->
<xs:complexType name="FloorParametersType">
    <xs:annotation>
        <xs:documentation>to do maxholders</xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="description" type="xs:string" minOccurs="0"/>
        <xs:element name="policy" type="xs:string" minOccurs="0"/>
        <xs:element name="pre-emptible" type="xs:boolean" minOccurs="0"/>
        <xs:element name="access-queue-parameters"
type="QueueParametersType" minOccurs="0"/>
        <xs:element name="request-queue-parameters"
type="QueueParametersType" minOccurs="0"/>
        <xs:element name="max-holders" type="xs:integer" minOccurs="0"/>
        <xs:element name="moderator-list" minOccurs="0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="moderator" type="UserIDType"
maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="user-permissions-list" minOccurs="0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="user-permissions">
                        <xs:complexType>
                            <xs:sequence>
                                <xs:element name="user"
type="UserIDType"/>
                                <xs:element name="permissions"
type="PermissionsType"/>
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
    <xs:attribute name="floor-id" type="FloorIDType" use="required"/>
</xs:complexType>
<xs:complexType name="FloorStateType">
    <xs:sequence>
        <xs:element name="holding-list" type="HoldingsType"
minOccurs="0"/>
```

```
        <xs:element name="access-queue-state" type="QueueStateType"
minOccurs="0"/>
        <xs:element name="request-queue-state" type="QueueStateType"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="floor-id" type="FloorIDType" use="required"/>
</xs:complexType>
<!--User Requests-->
<xs:element name="request-floors">
    <xs:annotation>
        <xs:documentation>participant</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="user-id" type="UserIDType"/>
            <xs:element name="floor-id-list" type="FloorIDsType"
minOccurs="0"/>
            <xs:element name="permissions" type="PermissionsType"
default="sendrecv" minOccurs="0"/>
            <xs:element name="reason-phrase" type="xs:string"
minOccurs="0"/>
            <xs:element name="expected-holding-time" type="xs:duration"
minOccurs="0"/>
            <xs:element name="request-create-time" type="xs:dateTime"
minOccurs="0"/>
            <xs:element name="request-expire-time" type="xs:dateTime"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="release-floors">
    <xs:annotation>
        <xs:documentation>participant</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="user-id" type="UserIDType"/>
            <xs:element name="floor-id-list" type="FloorIDsType"
minOccurs="0"/>
            <xs:element name="reason-phrase" type="xs:string"
minOccurs="0"/>
            <xs:element name="request-create-time" type="xs:dateTime"
minOccurs="0"/>
            <xs:element name="request-expire-time" type="xs:dateTime"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="yield-floors">
    <xs:annotation>
        <xs:documentation>participant</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="user-id" type="UserIDType"/>
```

```
            <xs:element name="to-user-id" type="UserIDType"/>
            <xs:element name="floor-id-list" type="FloorIDsType"
minOccurs="0"/>
            <xs:element name="reason-phrase" type="xs:string"
minOccurs="0"/>
            <xs:element name="request-create-time" type="xs:dateTime"
minOccurs="0"/>
            <xs:element name="request-expire-time" type="xs:dateTime"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="cancel-requests">
    <xs:annotation>
        <xs:documentation>participant</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="user-id" type="UserIDType"/>
            <xs:element name="request-id-list" minOccurs="0">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="request-id" type="RequestIDType"
maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="reason-phrase" type="xs:string"
minOccurs="0"/>
            <xs:element name="request-create-time" type="xs:dateTime"
minOccurs="0"/>
            <xs:element name="request-expire-time" type="xs:dateTime"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="get-parameters-floors">
    <xs:annotation>
        <xs:documentation>participant</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-id" type="FloorIDType" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="get-state-floors">
    <xs:annotation>
        <xs:documentation>participant</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
```

```
            <xs:element name="floor-id" type="FloorIDType" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:complexType name="ParticipantRequestListType">
    <xs:sequence>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="request-floors"/>
            <xs:element ref="release-floors"/>
            <xs:element ref="yield-floors"/>
            <xs:element ref="cancel-requests"/>
            <xs:element ref="get-parameters-floors"/>
            <xs:element ref="get-state-floors"/>
        </xs:choice>
    </xs:sequence>
</xs:complexType>
<!--Moderator Requests-->
<xs:element name="create-floors">
    <xs:annotation>
        <xs:documentation>moderator</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-parameters" type="FloorParametersType"
maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="freeze-floors">
    <xs:annotation>
        <xs:documentation>moderator</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-id" type="FloorIDType"
maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="remove-floors">
    <xs:annotation>
        <xs:documentation>moderator</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-id" type="FloorIDType"
maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="modify-parameters-floors">
    <xs:annotation>
```

```
        <xs:documentation>moderator</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-parameters" type="FloorParametersType"
maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="modify-state-floors">
    <xs:annotation>
        <xs:documentation>moderator</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-state" type="FloorStateType"
maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<!--Moderator Responses-->
<xs:complexType name="ModeratorFloorResponseType">
    <xs:sequence>
        <xs:element name="status" type="xs:boolean"/>
        <xs:element name="status-phrase" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="floor-id" type="FloorIDType" use="required"/>
</xs:complexType>
<xs:element name="moderator-response">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-response"
type="ModeratorFloorResponseType" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<!--Moderator Events Generated-->
<xs:element name="create-event">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-parameters" type="FloorParametersType"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="freeze-event">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-id" type="FloorIDType" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="remove-event">
```

```
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-id" type="FloorIDType" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="modify-parameters-event">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-parameters" type="FloorParametersType"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="modify-state-event">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-state" type="FloorStateType"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<!--User Responses-->
<xs:complexType name="ParticipantFloorResponseType">
    <xs:annotation>
        <xs:documentation>WIP - In the events generated, should there be a
req-id included to inform user of which of their claims is being referred
to?</xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="status" type="xs:string"/>
        <xs:element name="status-phrase" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="req-id" type="RequestIDType" use="required"/>
</xs:complexType>
<xs:element name="participant-response"
type="ParticipantFloorResponseType"/>
<xs:element name="get-parameters-response">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-parameters" type="FloorParametersType"
minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="get-state-response">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="floor-state" type="FloorStateType"/>
        </xs:sequence>
        <xs:attribute name="req-id" type="RequestIDType" use="required"/>
    </xs:complexType>
</xs:element>
<xs:complexType name="ModeratorRequestListType">
    <xs:sequence>
```

```
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="create-floors"/>
            <xs:element ref="freeze-floors"/>
            <xs:element ref="remove-floors"/>
            <xs:element ref="modify-parameters-floors"/>
            <xs:element ref="modify-state-floors"/>
        </xs:choice>
    </xs:sequence>
</xs:complexType>
<xs:element name="moderator-requests" type="ModeratorRequestListType"/>
<xs:element name="subscribe">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="subscriber" type="UserIDType"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Appendix II – Source Code Instructions

The contents of the README file accompanying are presented in this Appendix.

Sections

1. Build Instructions
2. Execute Instructions

Section 1 - Build Instructions

Platform

This software has been compiled on a Windows 2000 machine with the JDK v1.3.1. The make utility provided by cygwin is used to compile the project. Three components have been developed for the conference - Focus, Moderator and Participants.

Directory Structure

The directory structure of the source code reflects the packages for the components developed.

```
src/
    focus/
        fc/
        sipua/
        ui/
    moderator/
        fc/
        ui/
    participant/
        fc/
        sipua/
        ui/
    XMLSchema/
        impl/
```

Pre-requisites

Please ensure that the following libraries/files are available for compiling the project successfully -

1. JAXB Reference Implementation
    a. jaxb-api.jar,
    b. jaxb-ri.jar,
    c. jaxb-xjc.jar,
    d. jaxb-libs.jar,
    e. jaxp-api.jar,
    f. xercesImpl.jar,
    g. sax.jar,
    h. dom.jar and
    i. jax-qname.jar

2. NIST SIP Stack
    a. nist-sip.jar

3. Antlr Parser
    a. antlrall.jar

Compiling

In the source directory, the Makefile to compile the entire project is available. Each of the above directory also contains a Makefile that is invoked by the main Makefile. The paths to the libraries may need modification. To make each component separately, the targets are fdir (focus), mdir (moderator), pdir (participant) and xmlschdir (XML Schema). These can be invoked using the "make <target>" command.

Output

The output of the make is 4 jar files –

focus.jar,
moderator.jar,
participant.jar and
xmlschema.jar

The focus.jar contains all the compiled classes needed to execute the Focus. The moderator.jar file is used to execute the Moderator and the participant.jar contains the classes for the Participant.

The xmlschema.jar file is required for running each of the other components. It contains all the classes that implement the XML Schema elements.

Section 2 – Execute Instructions

Pre-requisites

The libraries specified in the compilation section are required to execute each component.

Additionally configuration files are required that are machine specific. These configuration files specify properties such as the machine address, port numbers (for the SIP Stack, FCServer and Event Notification Service) and SIP address of the entity, etc. For the test setup we have defined 1 conference server, 1 moderator and 2 participants.

For the Focus the configuration files are
src/focus/Focus.txt
src/focus/FocusServerConfig.txt

For the Moderator the configuration file is
src/moderator/Moderator.txt

For the Participant the configuration files are

Participant 1
src/participant/User1.txt
src/participant/User1ServerConfig.txt

Participant 2
src/participant/User2.txt
src/participant/User2ServerConfig.txt

Although we have created these files for the test setup mentioned above, any number of instances of each component may exist. For each instance, configuration file have to be created.

Executing

The following make files are available in the src/ directory to run the test components –

run – To run the participants.
runmod – To run the moderator.
runserv – To run the focus.

These may need modification to update the paths to the library and also to update the name/location of the configuration files, if changed. To run each component the “make –f <makefilename>” command may be used.