A Protocol for Reliable Decentralized Conferencing

Jonathan Lennox
lennox@cs.columbia.edu

Henning Schulzrinne
hgs@cs.columbia.edu

Department of Computer Science
Columbia University
New York, NY 10027

ABSTRACT

Many approaches and topologies — including multicast and media mixing — have been proposed for distributed Internet conferencing. While existing solutions can work well for large or pre-arranged conferences, they can be less appropriate for smaller, impromptu ones. We present an alternative, *full mesh conferencing*, which allows any number of parties to communicate in a conference without a central point of control. The protocol allows parties to join and leave the conference at any time, and ensures that all members of the conference are always informed of new members. The paper gives an overview of the protocol, analyzes it, describes a simulation environment for it, and discusses its applicability to the Session Initiation Protocol (SIP) and to other forms of decentralized communication.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Applications*

General Terms

Algorithms, Design, Measurement, Security, Experimentation

Keywords

Conferencing, Internet telephony, fully-meshed peer relationship, reliability, Session Initiation Protocol (SIP)

1. INTRODUCTION

The Session Initiation Protocol [7], SIP, is the Internet Engineering Task Force's standard for setting up multimedia sessions. It provides a means by which users can establish, maintain, and terminate calls between them. To aid this, it provides sophisticated user location and media description facilities. It provides facilities to set up diverse types of media, including instant messaging, distributed notification, and presence, as well as traditional audio and video.

The basic SIP protocol is only engineered for point-to-point communications, and does not, inherently, provide any support for communications among more than two parties, other than loosely-controlled multicast conferences in which the users' media is sent to a multicast group. More tightly-controlled conferencing is useful and necessary in a number of circumstances — from simple three-way calling, in which two people on an ordinary call decide to add a third party, to large-scale conference calls.

There are a number of ways to provide conferencing with existing SIP mechanisms. However, all these mechanisms have some shortcomings, as described in Section 2. They are heavyweight or architecturally inappropriate for certain types of conferences. This paper proposes a new approach, describing a fully-distributed, decentralized protocol for conferencing which establishes a fully-connected mesh of signalling and media connections between the conference participants. We call this approach *full-mesh conferencing.*

This approach is not intended to replace the other solutions, but rather to complement them. The existing solutions are designed for certain problem domains, and are useful in those domains; however, they are over-engineered or architecturally inappropriate in some common scenarios. The new proposal addresses these scenarios.

This conferencing approach is also applicable to additional environments. Numerous scenarios require multiple networked devices to be able to communicate with each other without a single point of failure, and the topology of a full mesh is often very useful for robustness. Such topologies often need to be dynamically assembled, with end systems entering or leaving the group. Thus, the mechanism described in this paper is also useful for such environments as group text messaging, highly-reliable alerting or event systems, establishing router peering relationships, distributed simulation, distributed databases, or clusters of network servers which need to share state information.

1.1 Related Work

The 'Sticky' Conference Control Protocol [3] is an early example of decentralized conferencing. It establishes an arbitrary topology, so that not all users can necessarily hear all the others. The Mesh-enhanced Service Location Protocol [9] offers another example of a service in which fully connected meshes of devices need to be maintained as systems arrive and leave. This work establishes a protocol which

NOSSDAV'03, June 1–3, 2003, Monterey, California, USA.

lets Service Location Protocol Directory Agents exchange service registration information, so they can maintain consistent data for shared scopes. Unlike the work presented in this paper, however, this protocol has no real notion of peer discovery or invitation, except via the Service Location Protocol's normal multicast advertisement. It deals only with state synchronization.

Explicit Multicast, or Xcast [1] offers a networking technology that can be complementary to full mesh conferencing, in networks which support it. With this technique, an IP device can explicitly specify a list of destinations in the IP header for a single packet; replication then occurs in the network. In a fully meshed conference, therefore, this could allow a conference member to save significantly on its bandwidth use. The full mesh protocol could complement this technique by providing a mechanism for conference members to know the addresses of the other participants in the conference.

The full mesh conferencing model has been proposed before in the evolution of the SIP protocol [8]. The work at the time foundered on the difficulty of ensuring that all users maintained full knowledge of the other members of the conference in complex scenarios. This paper revives and completes this work.

1.2 Structure of this Paper

The rest of this paper is organized as follows. Section 2 gives an overview of existing models for SIP conferencing, and discusses their advantages and shortcomings. Section 3 describes our novel alternative, full mesh conferencing, and Section 4 describes how it can be secured. Section 5 describes how we have verified the protocol with a simulation environment, and Section 6 analyzes the protocol and explains the rationale behind some of its features. A possible realization of the protocol in SIP is given in Section 7. Future work is discussed in Section 8, and Section 9 offers some conclusions.

2. EXISTING CONFERENCING MODELS

There are several ways to support multi-party conferencing in basic SIP. Rosenberg and one of us (Schulzrinne) discuss this in an Internet-Draft [6]. To simplify somewhat, there are two primary ways to support conferencing with basic SIP: *multicast* and *mixing*.

2.1 Existing Conferencing: Multicast

Large-scale multicast conferences were the original motivation for the development of SIP. In a large-scale multicast conference, one or more multicast addresses are allocated to the conference. Each participant joins the multicast groups, and sends their media to the groups. Signalling is not sent to the multicast groups. The sole purpose of the signalling messages is to inform participants of which multicast groups to join.

Multicast conferences can work reasonably well in networks that support them. They have the advantage that they do not require tight coordination between end systems; conference members can join and leave the conference independently, and conferences can survive network trouble and reconnect themselves seamlessly. The primary disadvantage of multicast conferences, however, is that multicast can be burdensome for networks and routers. Multicast (PIM-DM, PIM-SM) requires that each router at least stores the group identity. In some cases, state is actually (S, G), i.e., you need to store sender state as well. With lots of very small groups where everyone sends, i.e., the typical 3-party phone calls, routers effectively store session state. Also, since subscription to multicast groups is usually not authenticated (since routers would need to keep the keys for users), anybody can subscribe to any group, thus directing traffic to random destinations. A single misconfigured or compromised system could fairly easily subscribe to all IPv4 dynamically-allocated multicast addresses and thus flood the network. As a result, very few Internet backbones support multicast. While multicast conferences can be useful in LANs, enterprise environments, or Internet 2, in the current commercial Internet they are largely impractical.

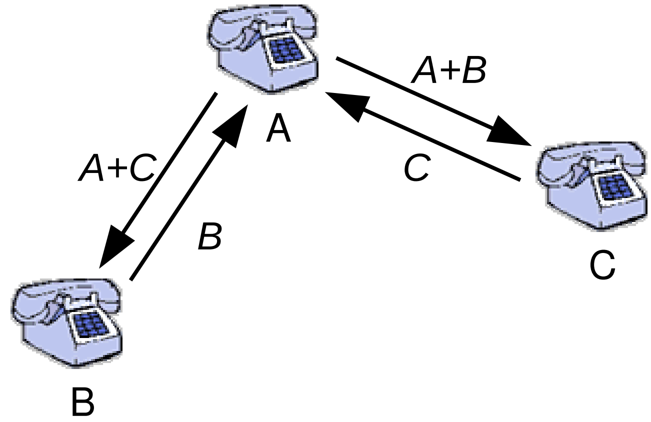

Figure 1: Conferencing: end system mixing

Multicast conferences are inherently loosely-coupled, and so they are not a good choice when tighter control of conference membership is desired. Communication of conference membership is carried out only using RTCP, so speakers may be unaware of who is currently able to hear them. They have no restriction, other than encryption, on users joining a conference, and key distribution and management can be cumbersome. Additionally, transition from a two-party to a multiparty session is awkward. Thus, while multicast can be useful for "webcasts," in networks which support it, it tends to be architecturally less applicable to the "conference call" model of group communications.

2.2 Existing Conferencing: Mixing

The other existing approach to conferencing is to have a SIP endpoint which connects the members of a conference, which mixes and forwards their media streams. There are two possible variants on this model: in *end system mixing*, shown in Figure 1, one member of the conference takes responsibility for mixing audio traffic; in *server-based mixing*, shown in Figure 2, an independent network entity performs it.

This model is probably the most common way of doing SIP conferencing. From the point of view of those end systems which are not performing mixing functions, the call can be treated as a standard SIP call. However, the model has several disadvantages. First of all, the existence of the conference is dependent on the mixer; if the mixer goes away, the call immediately ends. (This is more of a concern for end system mixing than for server-based mixing.) Secondly, the

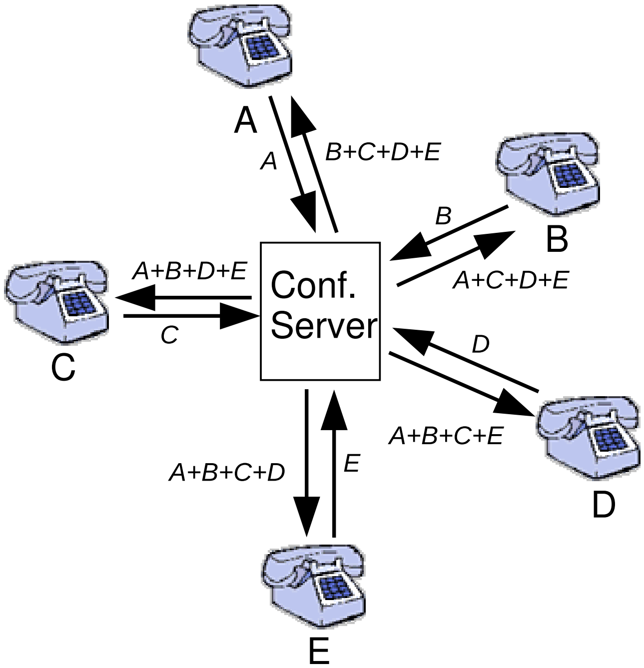

Figure 2: Conferencing: conference server mixing

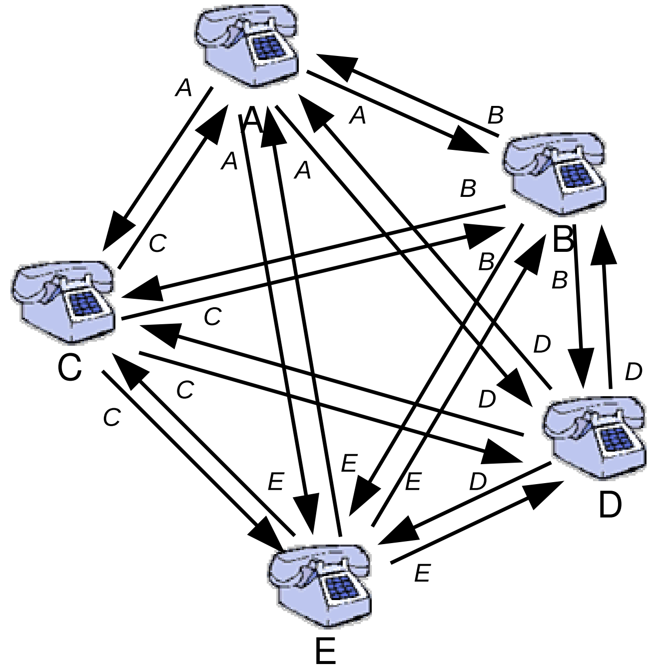

Figure 3: Conferencing: full mesh

computational load on the mixer can be high; it may need to encode up to $N-1$ audio streams for an N-party conference. (Hierarchical mixing can lessen this computational load, while makes mixer setup correspondingly more complex.) Finally, transitioning from a simple two-party call to a conference can be complex, particularly in the server-based mixing case, as the parties must locate a server, and then transition the existing call to the control of the server. Overall, of the two, server-based mixing is more reliable, and it works well for moderately large or pre-arranged conferences. However, it can be unwieldy for smaller conferences.

3. FULL MESH CONFERENCING

This paper presents a new approach to conferencing, *full mesh conferencing*. It is intended for tightly-coupled, impromptu, small-to-medium size conferences (with up to, perhaps, 10 members) — that is to say, "conference calls," not the larger "presentation" sessions for which the dedicated resources of a conference server or the loose coupling of a multicast conference are likely more appropriate.

Figure 3 illustrates this model. In the full mesh model, every endpoint directly communicates with every other one. All the parties in the conference are "equal" — no user is topologically special, or has any additional rights or abilities beyond those of the others. Any member of the conference can at any time invite a new user to the conference. If the new member accepts, it establishes connections to the other parties in the conference. Similarly, any member of the conference can drop out of it at any time, without affecting the remaining conference participants.

Audio is mixed only for playout at end points; mixed audio is never sent over the network. This has advantages and disadvantages. The primary advantage is that no end system needs to encode more than one media stream, per outgoing codec. For most voice codecs, encoding tends to be much more computationally complex than decoding. Each user will be decoding up to $N-1$ media streams in an N-member conference, but needs to encode only one. However, in an N-member conference, each user must have the bandwidth available to be able to send $N-1$ simultaneous streams. (For audio conferences, users will normally only need to be able to receive and decode one or two simultaneous streams. However, for video conferences, in most circumstances all the conference members will be sending at all times; while the user agent may, for instance, only choose to show the active speakers, the video streams will still use up bandwidth.) Thus, this mechanism is less practical for bandwidth-limited end systems such as wireless devices, users with 56 kb/s modems, or users with asymmetric DSL connections with low upstream bandwidth, and it does not scale well to large conferences. A hybrid model, illustrated in Figure 4, can ameliorate this issue; this is a matter for future research.

3.1 Example

The presentation of the protocol will begin with some examples. This presentation of the protocol uses an abstract representation of point-to-point communication between peers. These messages are inspired by SIP, but should not be interpreted as being actual SIP messages. A proposed mapping of these messages to SIP is presented later in Section 7. For the purposes of the examples, it is sufficient to understand that the abstract message JOIN invites a new member to join a conference; the message CONNECT establishes communication between two endpoints which are already members of the conference. Each of these messages can be answered with an Ok response, indicating that the request was accepted, or a Reject response, indicating that it was refused; the Ok responses are in turn acknowledged with Ack messages. This three-phase call setup procedure is needed to ensure conference security, for reasons explained in Section 6.2. These abstract messages are described in

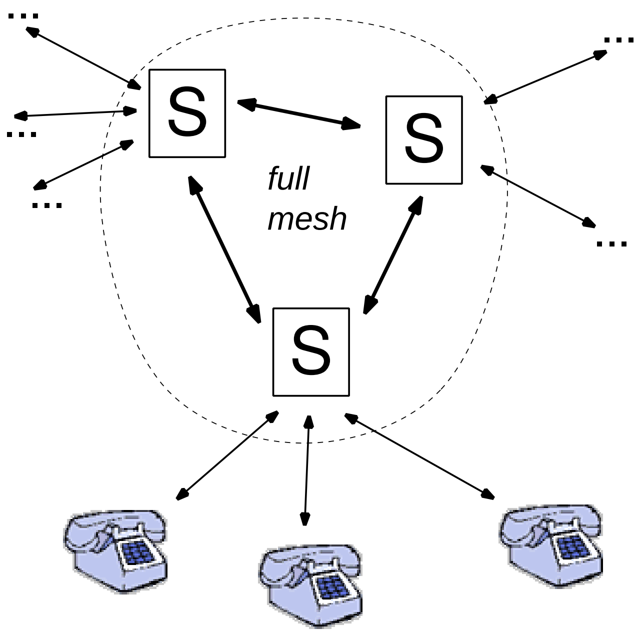

Figure 4: Conferencing: combination of conference servers (S) and full mesh

more detail in Section 3.2 below. Every end system maintains a list of the other end systems in the conference. Whenever a new member is invited, the inviter passes it a list of all the other members of the group. The new member then establishes communication with all the listed members.

Figure 5 shows the simplest case: a third endpoint being invited to a join two-party call. Initially, A and B are in a call. A then decides to ask C to join the call. To do this, A sends a JOIN message to C. C responds with a JOIN Ok message, indicating that it wishes to join the group. A then sends C a JOIN Ack message. This message lists A's view of the current membership of the group: A, B, and C. Upon receiving this message, C determines that it does not have a connection to B, and thus sends B a CONNECT message. When B receives the CONNECT message, it replies with CONNECT Ok; C responds to this with CONNECT Ack.

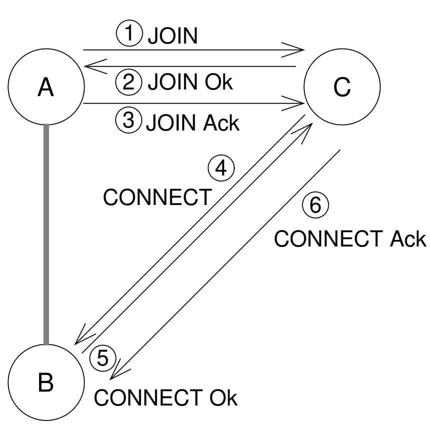

Figure 5: Example full mesh message flow: a new member is invited

At this point, every member has a communications channel established with every other.

Figure 6 illustrates what happens when both A and B invite new members, C and D, simultaneously. This illustrates that the protocol's message flow can quickly become quite complex. In the example, first A invites C, and B invites D, using the same procedure as for the simple message flow above. In the CONNECT Ok responses in messages 6 and 8, C and D are informed of each other. Since neither has yet established communications with the other, they both send each other CONNECT messages. In the specific instance illustrated in the figure, these CONNECT messages pass each other; at this point, one of the two must arbitrarily be chosen, and the other rejected, so that the only one communications dialog is set up. The mechanism of this is explored in more depth in Section 3.4.

Note that the order of these messages is not fixed. The transaction between A and C (messages 1, 2, and 3) triggers the transaction between C and B (messages 7, 8, and 9); but these are entirely independent of the transaction between B and D (messages 4, 5, and 6). The resulting behavior of each of the end points, and which systems contact which other ones, depends on the exact order in which messages are sent and received. The full mesh protocol is designed to work correctly in all these cases, and the group will always converge to a proper full mesh.

3.2 Protocol Messages

The protocol uses ten abstract messages: four initial messages, JOIN, CONNECT, LEAVE, and UPDATE, and the responses JOIN Ok, JOIN Ack, JOIN Reject, CONNECT Ok, CONNECT Ack, and CONNECT Reject. Messages are sent within the context of, and control, *dialogs*. A dialog is a communications session between two end systems. It corresponds to the existence of bidirectional media exchange between the end systems. Every dialog is identified by a globally unique *dialog identifier*. Additionally, every conference has a globally unique *conference identifier*. A conference dialog falls within exactly one conference.

The JOIN and CONNECT messages are largely similar, as are their responses. Each message requests the initiation of a dialog between two end systems. The Ok responses accept the dialog initiation, whereas the Reject responses refuse it. The Ack messages confirm the receipt of the Ok messages; these are necessary for reasons explained in Section 6.2. The difference between JOIN and CONNECT is in their semantics: the JOIN message is sent to a user not in a group, to ask it to join the group; its handling typically requires a user decision, to accept or reject the request. The CONNECT message, by contrast, is sent from an end system that has joined the group to the other pre-existing group members, to establish point-to-point dialogs. In this case, the message (if validated) will not normally require human interaction. The distinction between JOIN and CONNECT is necessary; without it, it would not be possible to distinguish the cases of an end system being re-invited to a conference, after having left it, from that of a newly-arrived end system attempting to connect with a recently-departed one, not having yet been informed of the latter system's departure.

The LEAVE message terminates a dialog, regardless of how the dialog was established. The UPDATE message does not affect the state of the dialog. It informs a party of new information about the conference membership list. This is

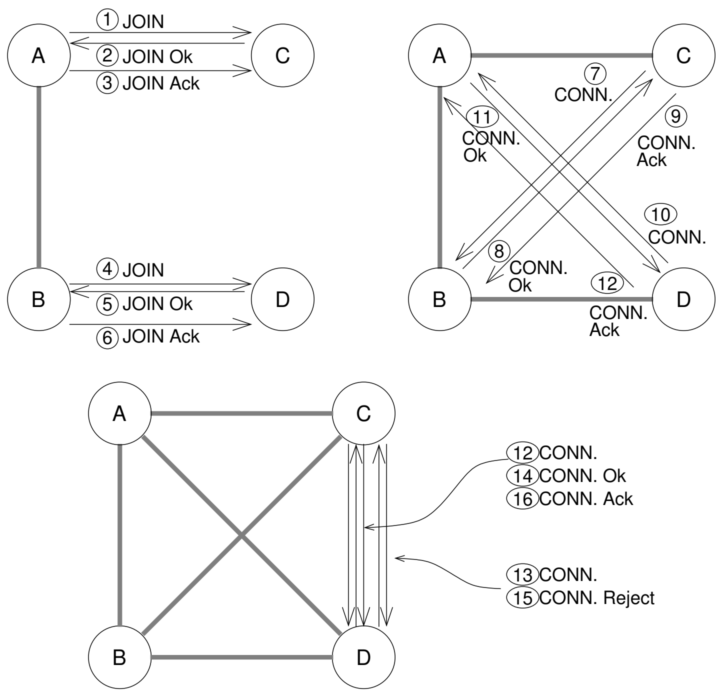

Figure 6: Example full mesh message flow: two new members are invited simultaneously

discussed further in Section 3.3. All messages are assumed to be transmitted reliably. (The Ack messages are not for reliability, but rather carry the third message of the three-phase session establishment.)

From each end system's point of view, a dialog can be in two possible states: *pending* or *established*. For the party that initiates the dialog, the dialog is pending until it receives the Ok message; for the party that answers it, it is pending until it receives the Ack message.

3.3 Membership Maintenance and State Communication

Several messages of the full mesh protocol — JOIN Ok, CONNECT Ok, JOIN Ack, CONNECT Ack, and UPDATE — carry information about the sender's current view of the conference membership. In these messages, the sender lists all the conference members with which it has a dialog and whose conference tags it knows. (Conference tags are described in Section 3.5. A sender will always know the tags of members with which it has an established dialog, but may not yet know them for pending dialogs; pending dialogs for which tags are not known are not listed.) Each member in the list is marked with the state, pending or established, of the sender's dialog with that member. Additionally, the JOIN message may carry an advisory list of conference members, so the recipient knows who is in the conference, and can use this information to decide whether to join it. However, in this case the list does not result in any protocol actions.

When the recipient B receives a message (other than JOIN) carrying a membership list from a sender A, and chooses to act on it (i.e., it does not respond to it negatively), it does two things. First of all, B consults its own membership list, and checks to see if any of the members with established dialogs listed are members it was not previously aware of. If there are, it sends new CONNECT messages to all these members.

Secondly, B prepares its own list of members in response. If the A's message was a JOIN Ok or CONNECT Ok, B always includes the list of members in its Ack response. If, however, A's message was an Ack or an UPDATE, normally B would have no response to send. However, if B has *established* connections to conference members which A did not know of (either as established or pending), B initiates a new UPDATE message within the same dialog, to inform A of all the members it knows of. Note that this response includes all established members A mentioned in its initial message, since B will now be setting up dialogs with all these members and will list them as pending. This ensures that this message will not itself trigger another UPDATE message unless A learns of further additional conference members.

The separation between established and pending members in the membership list ensures that every member's first introduction to a conference is the initial JOIN message it receives. If B were to send CONNECT messages to A's pending members, it is conceivable that the CONNECT message from B to A's pending member C could outrace a JOIN message from A to C, if, for example, A invited B and C simultaneously. This violates the definitions of JOIN and CONNECT.

3.4 The Double-Dialog Glare Problem

Because of the way the full mesh protocol floods membership information, it quite often happens that two end systems may attempt to establish dialogs with each other simultaneously for the same conference. In these cases, it is necessary to ensure that only one dialog is actually established, and the other is rejected. This is analogous to the problem of "glare" in the PSTN, in which two telephone switches simultaneously attempt to seize the same voice circuit. If both dialogs were to be set up, there would be two simultaneous connections between the end points; this is undesirable, as it is wasteful of bandwidth and causes unnecessary state complexity.

There are two possible scenarios for this. The simpler case is when a dialog establishment request (JOIN or CONNECT) arrives from A to B, when B already has an established dialog with A. In this case, B can simply always send A a Reject response to this request. The more complex case is when a dialog establishment request from A to B arrives when B has a *pending* dialog with A. The new dialog establishment request indicates that A also has a pending dialog with B; the situation is thus symmetric.

To solve this, a symmetry-breaking mechanism must be defined, so that both end systems can agree as to which dialog will be established, and which will be rejected. The simplest solution is to establish some global ordering for end systems, such that the connection from the "earlier" system to the "later" one is chosen. The exact nature and mechanism for this ordering is arbitrary, as long as it is deterministic and universally agreed-to. (A lexicographic ordering of end systems' globally-unique identifiers is one possibility, so long as these identifiers are communicated in JOIN Ok and CONNECT Ok messages.) Which dialog "wins" in this situation has little import in practice, as the direction in which a dialog was established does not matter for future communications. (The direction may matter for minor low-level details of the communications protocol, but these do not affect session and media semantics.)

3.5 Immediate Departure and Reconnection

It is possible for a user to be re-invited to a conference while in the process of leaving it. For example, a user hangs up accidentally, and is immediately invited back by the other conference members. In this situation, the system's CONNECT message for its new dialog could out-race the LEAVE message terminating the old dialog. Absent any mechanism to prevent this, the destination system for these two messages would perceive the CONNECT as setting up a double dialog, as described in Section 3.4, and would reject it.

Therefore, the protocol introduces *conference tags*. Whenever an end system joins a conference, it generates a unique identifier which will serve to identify this "instance" of its conference membership. It communicates this identifier in every message it sends. Additionally, end systems include conference tags for each member in the membership lists they send in Ok and Ack messages. Finally, whenever an end system knows the conference tag of the party to which it is sending a message, it includes the remote party's tag in the message. The only messages for which end systems do not know their counterparties' tags are the initial JOIN message, and a LEAVE message which was sent immediately following a JOIN, before JOIN Ok has arrived. (This can occur if an end system invites another party to the conference, and then immediately leaves.)

These conference tags are used in two ways to eliminate the problem of departure and reconnection. First of all, if an end system B receives a CONNECT message from another system A with which it already has a connection, but A's conference tag is different, it knows that this is not a double dialog. For example, consider the new CONNECT message to be from $A2$, and the old one to be from $A1$. In this case, B establishes a connection with $A2$, and can conclude that a LEAVE message from $A1$ will be forthcoming shortly.

Additionally, if an end system receives a message addressed to it with an unknown or outdated conference tag, it rejects the message, just as it would if it received such a message for an unknown conference. In the example, if A receives a CONNECT message from C addressed to $A1$, it knows that C has outdated information about A's state, and rejects it; it can conclude that once updated information has propagated to C, C will send a correct CONNECT message to $A2$.

4. SECURITY AND AUTHENTICATION

Security is a significant consideration for full mesh conferences. In addition to all the security requirements of point-to-point calls — authentication of the identities of callers and called parties, privacy and authentication of media traffic, and privacy of callers' and called parties' identities from third parties, for example — conferences have the additional requirement that only end systems authorized by an existing conference member are allowed to join the conference.

Under the model described in Section 3, an end system which receives a CONNECT message for an existing conference will automatically establish a dialog with the sender of the message, and then send the sender a media stream containing all the media generated by that end system. Clearly, it is very important that the end system have some way to know that this CONNECT message is coming from a legitimate conference member, i.e., one who has been invited to the conference by an actual user. Otherwise, if an adversary were able to observe or guess conference IDs, he or she would be able to barge into a conference without the consent of its members.

To resolve this, we require some way to verify that the CONNECT message was indeed triggered by a legitimate JOIN from a legitimate user. To accomplish this, we use a cryptographic public key solution. Whenever an end system joins a conference, it generates a purpose-built [2] public key which it will use for the duration of the conference.[1] All JOIN, CONNECT, JOIN Ok, and CONNECT Ok messages communicate the sender's public key to the other members of the conference. Then, whenever an end system A sends a membership list (in Ok or Ack messages) to another end system C, it includes in this invitation a "letter of invitation", signed with its private key, indicating that A has invited C to be a member of the conference. If this message's membership list informed C of the existence of B, C, as described above, sends a CONNECT message to B. In this message, it includes a copy of the letter of introduction, signed by A. B has already received a copy of A's public key, so it can verify the signature on the introduction, and so know that C is legitimately allowed to join the conference.

Because of race conditions between a session member sending a JOIN message, and its own departure from the conference — if, for instance, A invites C to a conference, and then immediately leaves, before C has contacted B — it is necessary for end systems to remember the public keys of

[1] In practice, as key generation is an expensive operation, end systems will probably use longer-lived public keys, and the signing mechanism will include a means to bind signed messages to conference IDs and tags.

members who have departed the conference. The length of time for which keys need to be remembered depends on the maximum length of time that letters of invitation could persist in ongoing transactions between conference members; roughly speaking, this will be twice the maximum duration of a transaction.

This security mechanism can be attacked by an attacker who can intercept and modify messages, by altering the public keys that members advertise to each other. Such an attacker could create its own letters of introduction at will. However, this same vulnerability exists for point-to-point communications. Communications systems need to be able to secure their point-to-point communications in order to provide user security. SIP, for example, uses mechanisms such as TLS and S/MIME for this purpose. These security mechanisms, combined with the approach described above, should be sufficient to protect against conference barge-in. This solution also does not prevent a member who has left the group from re-entering it, by replaying an existing certificate. It is possible that certificates could be set up to have limited lifetimes. This works if clocks are synchronized, but will require further investigation if there is no globally synchronized clock shared among all the conference participants.

5. VERIFICATION OF FULL MESH PROTOCOL

As mentioned earlier, there have been previous attempts [8] to describe full mesh conferencing for SIP. (The authors of this paper were involved in these previous attempts.) These attempts established the basic concept of the full mesh conference, but foundered on the difficulty of verifying manually that the protocol always converged.

The primary difficulty in verifying of the full mesh protocol is that its behavior depends strongly on the order in which events occur. For example, consider the example from Figure 6 in Section 3.1. In the example, messages 2 and 5 — the "horizontal" JOIN Ok messages from C to A and from D to B — are received before the "diagonal" CONNECT messages 7 and 10 from C to B and from D to A. Thus, at the time of the processing of the JOIN Ok messages, A and B are unaware of the existence of D and C respectively.

If, instead, for example, the CONNECT messages were to outrace the JOIN Ok messages, A and B would know already know about a new fourth member of the group when they received the JOIN Ok message. Thus, by the procedure of Section 3.3, A and B would include D and C in their JOIN Ack messages to C and D, respectively, informing them of the new member.

The number of possible orders in which events can occur is, in fact, exponential in the number of members in the group and the number of actions (JOIN messages and group departures) which will occur. Thus, two facts quickly became clear: protocol verification would need to be automated, and this automation would have to be heavily optimized in order to keep verification tractable on standard hardware.

5.1 The Verification Framework

A custom program was written in C++ to verify the protocol. The simulator maintains two items: firstly, the *state* of the system, describing which end points are in the system and each end point's knowledge of its dialogs; and secondly, a list of pending *events* which are to be executed, consisting of event actions (members inviting other end systems to the group, and members leaving the group) and sent messages. To simulate a particular scenario, the state is set up with some number of users in a fully-connected conference, and the event list contains the actions to be taken.

Recursively, the verifier picks an events from the pending event list, and applies the actions it specifies to the state. These actions may involve the addition of additional events to the event list, as when an event causes messages to be sent. The verifier is then executed on the new state and event list. Once the sub-list has completed, the verifier chooses the next event from the list, and continues until all events have been exhausted. In this way, the verifier exhaustively searches every possible event ordering.

When the verifier is executed with no pending events, it instead *validates* the resulting final state. A final state is valid if every user which believes itself to be a member of the group has exactly one, alive, dialog with every other such user. If the group is disconnected, not fully linked, or any dialogs are in the wrong state or doubled, the verifier prints an error message and the exact sequence of events and states that led to this outcome.

Because many events execute independently of one another, this simulation environment can end up repeating many scenarios. For example, in the message flow of Figure 6, if the first two events executed are the transmission of message 1 and the transmission of message 4, it is irrelevant which of these two occurs first; the state, and pending events, afterwards will be the same. Thus, the verifier maintains a *state cache*. After state has finished being executed, it is recorded in the state cache. If a future execution of the verifier for this verification run results in the same state and list events being visited, the execution is pruned as redundant. This greatly reduces the number of test cases explored by the validator, but it means that simulations can fail if the cache fills all available memory.

5.2 Test Runs Performed

Table 1 lists all the simulated mesh actions executed by the verifier. Conference members are named A, B, C, ..., in order.

In almost all cases, the verifier confirmed that every possible ordering of the events and messages of the full mesh protocol resulted in a fully-connected and self-consistent conference. There are, however, two exceptions to this. First of all, in two cases (marked with a †) the state cache grew so large as to exhaust all RAM and swap on the computer on which the simulation was executing. This exhaustion occurred after several tens of millions of orderings had been considered and pruned. As mentioned above, the number of event orderings is exponential in the size of the groups and the number of initial actions. This is why none of the simulated groups involve more than four members.

In one case (marked with a $*$ in the table), the simulation did not result in a single fully-connected conference, but instead resulted in several smaller disconnected ones, as the "bridging" members of a conference left the conference before the new members found out about each other. Specifically, in simulation 40, B and C both join what they view as three-party conferences, and then have both their peers leave. Because A and B are gone, C and D never

Run	Initial	Actions
1	A	$-A$
2	A, B	$-B$
3	A, B, C	$-C$
4	A	$A{\to}B$
5	A	$A{\to}B$, $A{\to}C$
6	A	$A{\to}B$, $-B$
7	A	$A{\to}B$, $-A$
8	A	$A{\to}B$, $-B$, $A{\to}B$
9	A	$A{\to}B$, $-A$, $B{\to}A$
10	A	$A{\to}B$, $A{\to}C$, $-A$
11	A	$A{\to}B$, $A{\to}C$, $-B$
12	A	$A{\to}B$, $A{\to}C$, $B{\to}C$
13	A	$A{\to}B$, $A{\to}C$, $B{\to}C$, $C{\to}B$
14	A	$A{\to}B$, $A{\to}C$, $-A$, $B{\to}C$
15	A	$A{\to}B$, $A{\to}C$, $-A$, $B{\to}C$, $C{\to}B$
16	A	$A{\to}B$, $B{\to}C$
17	A	$A{\to}B$, $B{\to}C$, $-A$
18	A	$A{\to}B$, $B{\to}C$, $-B$
19	A	$A{\to}B$, $B{\to}C$, $-C$
20	A	$A{\to}B$, $B{\to}C$, $A{\to}C$, $-A$
21	A	$A{\to}B$, $B{\to}C$, $A{\to}C$, $-B$
22	A	$A{\to}B$, $B{\to}C$, $A{\to}C$, $-C$
23	A	$A{\to}B$, $-B$, $A{\to}B$
24	A	$A{\to}B$, $B{\to}C$, $-B$, $A{\to}B$
25	A	$A{\to}B$, $-A$, $B{\to}A$
26	A, B	$A{\to}C$
27	A, B	$A{\to}C$, $A{\to}D$
28	A, B	$A{\to}C$, $B{\to}C$
29	A, B	$A{\to}C$, $-A$
30	A, B	$A{\to}C$, $-B$
31	A, B	$A{\to}C$, $-C$
32	A, B	$A{\to}C$, $A{\to}D$, $-A$
33	A, B	$A{\to}C$, $A{\to}D$, $-B$
34	A, B	$A{\to}C$, $A{\to}D$, $-C$
35	A, B	$A{\to}C$, $-C$, $A{\to}C$
36	A, B	$A{\to}C$, $-C$, $C{\to}A$
37	A, B	$A{\to}C$, $B{\to}D$
38	A, B	$A{\to}C$, $B{\to}D$, $-A$
39	A, B	$A{\to}C$, $B{\to}D$, $-C$
40 ∗ †	A, B	$A{\to}C$, $B{\to}D$, $-A$, $-B$
41	A, B	$A{\to}C$, $B{\to}D$, $-A$, $-C$
42	A, B	$A{\to}C$, $C{\to}D$
43	A, B	$A{\to}C$, $C{\to}D$, $-A$
44	A, B	$A{\to}C$, $C{\to}D$, $-B$
45	A, B	$A{\to}C$, $C{\to}D$, $-C$
46	A, B	$A{\to}C$, $C{\to}D$, $-D$
47	A, B	$A{\to}C$, $B{\to}C$
48	A, B	$A{\to}C$, $B{\to}C$, $-A$
49	A, B	$A{\to}C$, $B{\to}C$, $-C$
50†	A, B	$A{\to}C$, $B{\to}C$, $C{\to}D$, $-C$
51	A, B	$A{\to}C$, $-B$, $A{\to}B$
52	A, B	$A{\to}C$, $-B$, $C{\to}B$
53	A, B	$B{\to}C$, $-B$, $A{\to}B$
54	A, B	$B{\to}C$, $-B$, $C{\to}B$
55	A, B	$-A$, $-B$
56	A, B, C	$-B$, $-C$
57	A, B, C	$-A$, $-B$, $-C$

Key:

Initial The initial conference members, before any actions are executed.

Actions The actions to be executed, in some arbitrary order, during the scenario.

$X{\to}Y$ X will attempt to invite Y to the conference, if X is currently a member and does not know about Y.

$-X$ X will leave the conference, if it is currently a member.

∗ Some event orderings can lead to disconnected conferences. See the text for an explanation.

† The simulation's state cache exhausted all available memory before completing, on a Sun Fire 280R with 2.0 GB of RAM and 5.0 GB of swap, running Solaris 8.

Table 1: Full mesh conference scenarios explored with verifier

discover each other. This is not a 'bug' in the protocol; in the absence of a central repository of information about conferences, there is no way in this scenario that information about C could reach D, or vice-versa.

6. ANALYSIS AND RATIONALE

The examples of Table 1 cover a large number of the possible scenarios of full mesh operations, and each one exhaustively searches the possibilities for a particular set of operations. These simulations do not, however, fully explore the possibilities of the full mesh signalling, as the potential size of conferences is, of course, unlimited. In this section we will attempt to justify the belief that the cases considered adequately cover all the possible ways that a conference membership changes can interact. We will also give rationales for some of the more unusual features of the protocol, to illustrate how a more naïve protocol can fail.

6.1 Protocol Correctness

The first point to consider is to ensure that knowledge of a member joining the conference is always flooded to all other members of the conference. In the absence of simultaneous conference departures, this is clear. Once A invites B to the conference, B will send CONNECT messages to every member C, D, etc., that A knows about. In the responses to these CONNECT messages, B will transitively be informed of every member these members know about, and so, recursively, will eventually learn of, and be connected to, every member of the conference.

Similarly, departure from the conference is straightforward. LEAVE messages are sent to every member of the conference; even if other members of the conference subsequently attempt to connect to the new member, due to out-of-date lists of conference membership, the departing member will reject these connection attempts. Because conference tags distinguish instances of an end system's conference memberships, there is no ambiguity between near-simultaneous departures and re-connections, and so the analyses of these two scenarios can be considered independently.

The remaining case, therefore, is to consider simultaneous connections to, and departures from, the conference. It is possible in this instance for the conference to degenerate into several sub-conferences. This can happen if a "bridging" member of the conference — a conference member which alone knows about both portions of the conference — departs from the conference before propagating information between the two sides. In this case, however, both sides will still become fully-connected within themselves, by the argument above.

6.2 Rationale for Three-Phase Session Establishment

The most unusual feature of the protocol as it is described in Section 3 is likely the three-phase nature of the JOIN and CONNECT messages. This feature is necessary in order to ensure the correct behavior of the security mechanism described in Section 4. As described in that section, if B sends a membership list to C, it includes a signed "letter of introduction" certifying that C is allowed to connect to other members of the conference, say A. C can then include this in a CONNECT message to A, so A knows that C has been authorized. In order for this to work, however, A must already have received B's public key, so it can validate the signature on the letter of introduction.

In a two-phase connection model, there are some scenarios in which this would not be true. Consider, for example, a two-phase connection model in which A invites B to the conference, and then B immediately invites C. In a two-phase model, B could consider itself fully connected to A once it had sent a JOIN Ok response to a JOIN message from A. It could therefore immediately send a JOIN message to C, advertising A in its membership list, which could trigger a CONNECT message to A. However, in this scenario, the CONNECT message from C to A, with a letter of introduction signed by B, could out-race the JOIN Ok message from B to A, including B's public key. A would therefore not be able to verify the validity of the letter of introduction, and would reject the CONNECT message; a full mesh would therefore not be established.

In the actual three-phase model used by the protocol, however, B may not advertise A until B has an established connection to A. As described in Section 3.2, B does not have an established connection to A until it has received a JOIN Ack message from A. At this point, B knows that A has received a copy of its public key, so it can safely advertise A in future conference membership lists.

7. REALIZATION OF THE FULL MESH PROTOCOL IN SIP

The abstract protocol described in Section 3 was designed to be a simplified representation of SIP messaging. It was designed to be expressive enough to capture all the behavior necessary to represent point-to-point communications, yet simple enough to make the implementation and complexity of the automatic verifier, described in Section 5, tractable. In this section we will describe how this abstract protocol can be expressed in actual SIP messages.

SIP communication sessions are organized into dialogs. When SIP is being used to control multimedia communications, dialogs are initiated with the **INVITE** method. If the other side agrees to initiate the dialog, it responds with a **200 OK** response, which is acknowledged with an **ACK** request; otherwise, it sends one of a large number of potential failure responses. Once established, dialogs, and their associated multimedia communication, continue until they are terminated by either party, using the **BYE** method and its **200 OK** response. If the session initiator wants to terminate the dialog before it has received a final response to its initial **INVITE**, it can do so by sending the **CANCEL** request.

To implement the full mesh protocol in SIP, we provide a possible mapping of the full mesh protocol's abstract methods to concrete SIP methods. As both JOIN and CONNECT establish dialogs in the abstract protocol, they are both mapped to the SIP **INVITE** method. For similar reasons, LEAVE is mapped to either **BYE** or **CANCEL**, depending on the state of the dialog when it is invoked, and the UPDATE method can be mapped either to a re-**INVITE** or to a newly-defined SIP method (potentially, indeed, **UPDATE** [5]). The two subsequent phases of the connection process maps naturally: Ok becomes a **200**-class success response, Reject becomes a **400**-, **500**-, or **600**-class failure response, and Ack is **ACK**.

All of these requests must include several additional header fields, beyond those defined for a standard point-to-point call, in order to support full mesh conferences. First of all, to identify conferences, we define a header field **Conference-ID**, which uniquely identifies a full mesh conference. The value of this header field is established by the end system which initially creates the conference, and is globally unique. It is created using the same procedure as that used to create globally unique values for the existing required header field **Call-ID**. Secondly, to distinguish between JOIN and CONNECT messages, we define another new header field, **Invited-By**, which is included only for CONNECT messages. This header field carries the identity of the system which initially invited the sender of the message to the conference. This field can also have some cryptographic authentication; this was discussed further in Section 4. Finally, to provide the list of conference members, another header field, **Conference-Member**, is provided for those messages which include the member list. This lists the **Contact** addresses of all the conference members the request's sender knows about. A header field parameter, **status**, indicates whether the members are established or pending; another one, **tag**, gives the member's conference tag. Cryptographic formats for the public keys and letters of introduction remain to be determined; work for this should be developed based on ongoing work with purpose built keys [2]. Since these keys can be several kilobytes, they will most likely be carried as multi-part session bodies.

8. FUTURE WORK

As designed, the full mesh conference protocol is perfectly decentralized — no member of a conference has special privileges. While this is the proper model for many conferences, it is not universally applicable. Development is ongoing [4] on how to offer sophisticated admission and floor control for conferences. This development generally assumes centralized conferencing models, usually those involving a conference server. It would be useful to be able to use these capabilities in the decentralized environment, but this will require significant further investigation to see how well the assumptions of the centralized model can carry over to the decentralized case. In particular, there would need to be some mechanism by which the members of the mesh conference can agree about who has control over the conference and is authorized to make these decisions.

As discussed in Section 3, there are also circumstances in which combination topologies, falling somewhere between the full mesh and the centralized server, are useful. This is difficult to arrange in a decentralized manner, without prior configuration, as these topologies do not have the inherent symmetries of a star or a mesh. However, there are such environments: for example, as illustrated in Figure 4, a set of conference servers could form a mesh topology among them, and then provide a star topology to clients.

9. CONCLUSION

After reviewing a number of solutions for Internet conferencing, we concluded that they all have some limitations. We therefore presented an additional mechanism which complements these approaches, allowing conferences to be established in a reliable, decentralized manner. This mechanism is also applicable to other environments which require the decentralized establishment of a full mesh communications topology. We verified the protocol's correctness for a large number of scenarios, provided an analysis of the protocol's correctness, established a mapping to SIP, and discussed and provided solutions for some potential security issues with the protocol.

10. ACKNOWLEDGMENTS

This work was supported by a grant from Lucent Technologies.

We would like to thank Weibin Zhao for initial discussions on full mesh networking, the members of the Columbia University Internet Real-Time Laboratory for comments and suggestions, and Merav Hoffman for editing assistance.

11. REFERENCES

[1] R. Boivie et al. Explicit multicast (xcast) basic specification. Internet draft, Internet Engineering Task Force, Jan. 2003. Work in progress.

[2] S. Bradner, A. Mankin, and J. I. Schiller. A framework for purpose built keys (PBK). Internet draft, Internet Engineering Task Force, Jan. 2003. Work in progress.

[3] C. Elliott. A 'sticky' conference control protocol. *Internetworking: Research and Experience*, 5:97–119, 1994.

[4] O. Levin and R. K. Even. High level requirements for tightly coupled SIP conferencing. Internet draft, Internet Engineering Task Force, Mar. 2003. Work in progress.

[5] J. Rosenberg. The session initiation protocol (SIP) UPDATE method. RFC 3311, Internet Engineering Task Force, Oct. 2002.

[6] J. Rosenberg and H. Schulzrinne. Models for multi party conferencing in SIP. Internet draft, Internet Engineering Task Force, July 2002. Work in progress.

[7] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[8] H. Schulzrinne and J. Rosenberg. SIP call control services. Internet draft, Internet Engineering Task Force, June 1999. Work in progress.

[9] W. Zhao and H. Schulzrinne. mSLP - mesh-enhanced service location protocol. Technical Report CUCS-013-00, Columbia University, New York, May 2000.