Compositional Control of IP Media

Pamela Zave and Eric Cheung
AT&T Laboratories—Research, Florham Park, New Jersey USA
pamela,cheung@research.att.com

***Abstract*— In many IP media services, the media channels are point-to-point, dynamic, and set up with the participation of one or more application servers, even though the media packets themselves travel directly between media endpoints. The application servers must be programmed so that media behavior is globally correct, even though the servers may attempt to manipulate the same media channels concurrently and without knowledge of each other. Our proposed solution to this problem of compositional media control includes an architecture-independent descriptive model, a set of high-level programming primitives, a formal specification of their compositional semantics, a signaling protocol, an implementation, and partial verification of correctness. The paper includes performance analysis, comparison to related work, and principles for making other networked applications more compositional.**

***Index Terms*— distributed applications, domain-specific architectures, protocol design, protocol verification, software/program verification, networks, streaming media, multimedia services, telecommunications, feature interaction**

I. IP MEDIA SERVICES

IP media services use the Internet Protocol (IP) to make real-time audio and video connections. We are concerned with IP media services having two common characteristics.

First, the media channels are point-to-point and dynamic. This excludes dedicated long-term channels and multicast applications. It includes, however, a wide range of interactive applications including Internet telephony, home networks, computer-supported cooperative work (teleconferencing, telemonitoring, distance learning, and virtual reality), and computer-supported cooperative play (collaborative television, multiplayer games, and networked music performance).

Second, the dynamic setup of a media channel must sometimes involve the participation of one or more *application servers*. The presence and significance of application servers depends on several aspects of the architecture of media services, as follows.

In this paper a *media endpoint* is any source or sink of a media stream. Endpoints include original sources of media such as media synthesizers, cameras, and microphones. Endpoints include ultimate sinks of media such as user devices or clients with displays or speakers. Endpoints also include media-processing resources that perform a wide range of functions such as recording, playing, mixing, replicating, filtering, transcoding, and analyzing media streams.

In most media services, user devices act autonomously with respect to other media endpoints (even if acting as slaves to their human masters). For example, they can request connections at any time, and choose to accept or decline connections that are offered to them. For a media channel to be established between two endpoints, the endpoints must cooperate by means of a signaling protocol. The Session Initiation Protocol (SIP) is currently the best-known and most commonly used protocol for this purpose [7], [14].

It is frequently stated that IP media services can be implemented in the participating media endpoints. User devices are computers, and any necessary new software can be installed on them. As user devices become more powerful, they can even perform the high-level media processing required by some services.

Although it is certainly true that some IP media services are best implemented in the participating endpoints, it is also true that many media services are best implemented in separate application servers. Here are some of the reasons for using application servers:

- Handheld user devices are often disconnected from the network. An application server can provide a persistent network presence, such as voicemail, for handheld devices. A server can also make a user's media files accessible at all times from all devices, while files stored on an inaccessible device would not be.
- An application server can provide IP media services for closed user devices—user devices that were not designed to download new programs for built-in IP media services. The most obvious example of such a device is an ordinary telephone, connected to the Internet through the circuit-switched telephone network.
- Many user devices do not reside in trusted administrative domains. Applications with any kind of trust or security requirements cannot run on them.
- Without application servers, every multi-party service must be implemented in a completely decentralized way. This might be difficult or inefficient compared to implementations with some centralization in

servers.

- Without application servers, for every service accessed from a device, application software must be installed on that device. This is not realistic for services that are used infrequently by any one person.

We have stated that we are interested in services where the dynamic setup of a media channel sometimes involves the participation of one or more application servers. What if there are application servers, but they do not make decisions about media channels? There are such servers, but there are also many servers that make decisions about which media channels should exist when. They include servers concerned with switching and conferencing. They include servers for services that provide access to media resources as part of the service being offered. They also include servers for services that use audio signaling as their user interface. Audio signaling implements an extensible user interface on any audio device, by means of announcements, tones, touch-tone detection, and speech recognition. Audio signaling is a crucial ingredient of IP services that interoperate with circuit-switched telephones, because it is the usual way to augment the user interface of the device.

Wherever there is one application server, there might be several. A connection between two user devices might be supervised by two servers in two different administrative domains, each one serving one of the users. Often adding new functions to a system means adding new servers, because adding a new server is far easier than adding functions to an existing server. The IP Multimedia Subsystem (IMS) architecture [1], which is an emerging industry standard for media services, recognizes the necessity to route a particular signaling channel through multiple servers within the same service provider's configuration.

Whenever there are multiple application servers, it is likely that they were not programmed to coordinate with each other, beyond the common denominator of following a standardized signaling protocol such as SIP. Application servers representing different users may be in completely different administrative domains. Application servers performing different functions within one administrative domain may be produced by different vendors.

In the services we are concerned with, a typical media channel looks like Figure 1. Most importantly, there is a separation between the media channel itself and the signaling channel used to set up and control it. The signaling channel goes through one or more application servers, for all the reasons presented above. The media packets, on the other hand, travel directly between media endpoints. This is necessary because the media channel demands high bandwidth, so packets must travel by the shortest available paths. This is also necessary because the media channel demands low latency, so packets cannot incur extra delays from longer paths or server handling.

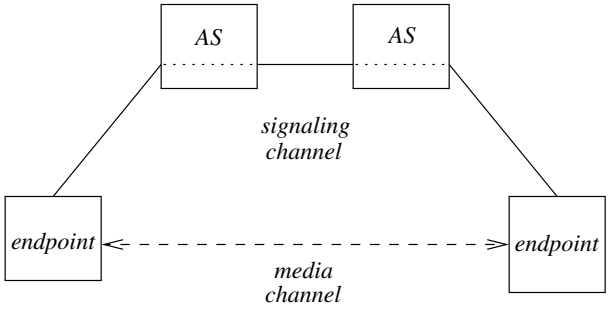

Fig. 1. In IP media services, signaling and media channels are separate.

The separation of signaling and media channels is reinforced by other factors besides the presence of application servers. Because these channels differ in both performance and reliability requirements, they use different underlying protocols. Signaling is low-bandwidth but demands reliability. It is common to use TCP for signaling, so that a signaling channel can be regarded as FIFO and reliable. Media is high-bandwidth. It is common to use RTP for media streams, because limited packet loss is preferable to delay. RTP can also be combined with quality-of-service mechanisms such as resource reservation.

II. The problem of compositional media control

Section I described a class of IP-based applications implemented in application servers. These servers must be programmed so that media behavior is globally correct, even though the servers may attempt to manipulate the same media channels concurrently and without knowledge of each other. We refer to this challenge as "compositional media control."

Before giving a more detailed description of the problem, we first show an example of why servers might act concurrently and independently, and what might happen if they are not coordinated properly.

A. An example of the problem

Figure 2 is a telephony example in four snapshots. The snapshots show the same media endpoints and servers at four different times.

Endpoint A is a telephone in an office with an IP PBX. Because of this, A has a permanent signaling channel to the PBX, and all signaling channels connecting A to other telephones radiate from the PBX. Among other features, the PBX allows A to switch between multiple

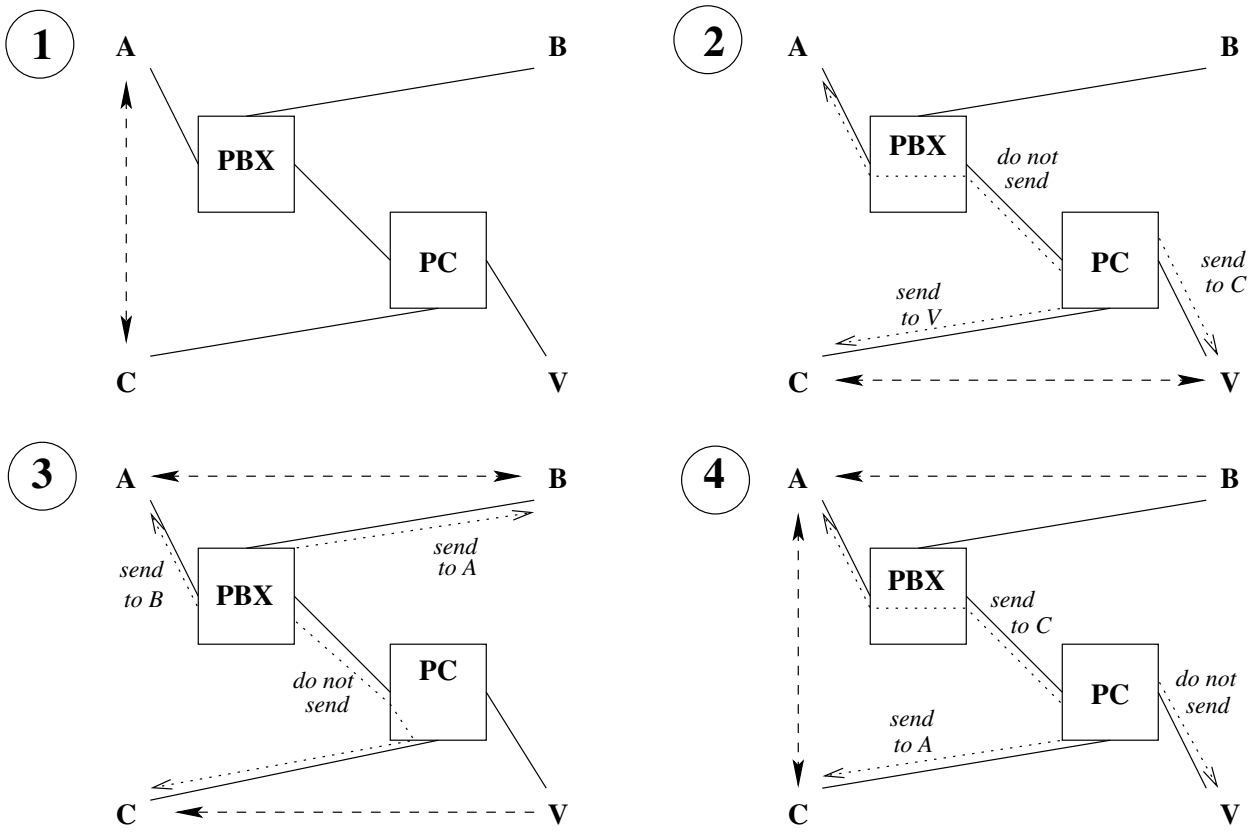

Fig. 2. An example of erroneous media control. A, B, C are telephones. PC is an application server implementing a prepaid-card feature. V is a media resource providing a user interface for PC by means of audio signaling. Solid lines are two-way signaling channels, dashed arrows are media channels, and dotted arrows are signals sent on signaling channels.

outside calls. Before any of the snapshots in the figure were taken, the user of A was talking to the user of telephone B, which is why there is a signaling channel between the PBX and B.

While A was talking to B, a user of telephone C, who is using a prepaid card, contacted the prepaid-card server PC and used the card to call A. A received notification of the incoming call and switched to C. This resulted in Snapshot 1. Note that in Snapshot 1 there is an audio channel between A and C. There is no audio channel to or from B, because A has put B on hold.

Snapshot 2 shows what happens when the funds of the prepaid card become exhausted. A timer goes off in PC and the server sends three signals. In protocol-independent terms, there is a signal to A telling it to stop sending media. There is a signal to C telling it to send media to the resource V, and a signal to V telling it to send media to C.

It is standard behavior for a server receiving a signal that does not concern itself to forward the signal untouched. In this example, because the servers are not coordinated—they are acting as if media signals concern media endpoints only—they forward all media signals that they receive. In particular, the PBX forwards the *do not send* signal to A.

After the endpoints respond to all commands in Snapshot 2, the only audio channel is between C and V. V will use it to prompt C to supply additional funds, and to receive authorization by means of touch tones.

Snapshot 3 shows what happens when A next uses the PBX to switch back to B. The PBX sends three signals appropriate to this function: a signal to A telling it to send media to B, a signal to B telling it to send media to A, and a signal to C telling it to stop sending media. This last signal passes through PC, which forwards it untouched to C.

Although the signals are appropriate from the PBX's point of view, they have the abnormal effect of leaving V without audio input from C. Note that the media arrow between C and V is now one-way.

Finally, Snapshot 4 shows what happens when V completes verification of the funds from C (presumably authorized before C was cut off) and reconnects C with A. PC sends a signal to A telling it to send to C, a signal to C telling it to send to A, and a signal to V telling it to stop sending media.

Although the signals are appropriate from PC's point of view, they have abnormal effects. Because the signal from PC is forwarded blindly by the PBX, the signal switches A from B to C without A's permission. Furthermore, B is left transmitting to an endpoint that will throw away the packets because it has been instructed to communicate with C.

B. Goals and plan for a solution

We have just shown a very simple example of what can go wrong without compositional media control. The goal of our work is to find a comprehensive approach to this problem. For an approach to be considered comprehensive, it should have the following three characteristics.

First, it should be general-purpose. It should be possible to use it to build any media service, and it should make media control relatively easy in all of them.

Second, it should be architecture-independent. It should not impose constraints on how functions are allocated to physical components, or on where physical components are located. Engineers should be free to base architectural decisions on other criteria such as cost and reliability.

Third, it should be automated and verified. Compositional media control is inherently complex; it is not something that every programmer should do afresh. Rather, application programmers should work with a set of high-level primitives. These primitives should be specified formally and implemented once. Further, the implementation should be verified as correct with respect to the specification.

Architecture-independence has another advantage beyond engineering freedom, as important as that is. If functions can be distributed arbitrarily across physical components, they can also be distributed arbitrarily across virtual components within physical components. In other words, we have support for *modularity.*

The value of modularity in developing media services has been demonstrated by the success of the Distributed Feature Composition (DFC) architecture [4], [9]. In DFC, a feature is implemented as an independent, concurrent module in a signaling pipeline. Because of this independence, each feature can be simple and comprehensible, and features are easy to add or change. In this way, DFC's goal of feature modularity has been reached.

DFC was used to build the advanced features for a commercial voice-over-IP service [3]. Using DFC, it was possible to deliver 11 features to a testing organization, two months from the inception of the project. Several of these features were very complex, requiring control of multiple parties and their audio channels. Nevertheless, the delivered software proved to be of high quality. This unprecedented development speed was due to the modularity and reuse provided by the DFC architecture. The work reported in this paper was motivated by DFC, and can be used to automate media control for DFC-like modules.

In the next three sections, we propose a comprehensive approach to achieving compositional media control, and give its formal specification. Section III provides an architecture-independent model for describing the media aspect of any service. All definitions are rooted in this model. Section IV defines a set of primitives for controlling media channels. This set of primitives is a high-level vocabulary to be used by application programmers. Section V defines the compositional semantics of these primitives in terms of temporal logic.

In the subsequent three sections, we present an implementation of the specification. Section VI describes a protocol for signaling between modules such as application servers. Section VII describes the software that must run on each module. Section VIII describes our partial verification of the protocol and implementation code. Relevant limitations are noted in each section.

This presentation of the approach is followed by a discussion of related work, of which there is relatively little, as the problem of compositional media control is not yet widely recognized (Section IX). Because compositional media control has similarities to other problems requiring modularity, distribution, and coordination, Section X makes an attempt to extract some general design principles for supporting composition.

C. An example of the solution

This paper is basically organized in a top-down fashion. Because it can be difficult to see where a top-down presentation is going, this section sketches how our approach solves the problem introduced in Section II-A. Subsequent sections fill in the details of this sketch.

At the highest level, the solution is exemplified by Figure 3. In this figure, the PBX and PC server are programmed using a fixed set of primitives for media control.

When a server program wants media flow between two media endpoints, it puts the two signaling channels that extend from the server to those endpoints under the control of a *flowLink* object, depicted by a dotted line in the figure. When a server program wants to interrupt media flow to an endpoint, it puts the signaling channel to that endpoint under the control of a *holdSlot* object, depicted by a black dot in the figure. The primitives also include an *openSlot* for opening media channels and a *closeSlot* for closing them, but these objects are not employed in the PBX/PC example.

The signaling protocol and the implementation of the media-control objects are designed to achieve the goals of the servers in which they reside, subject to the goals of

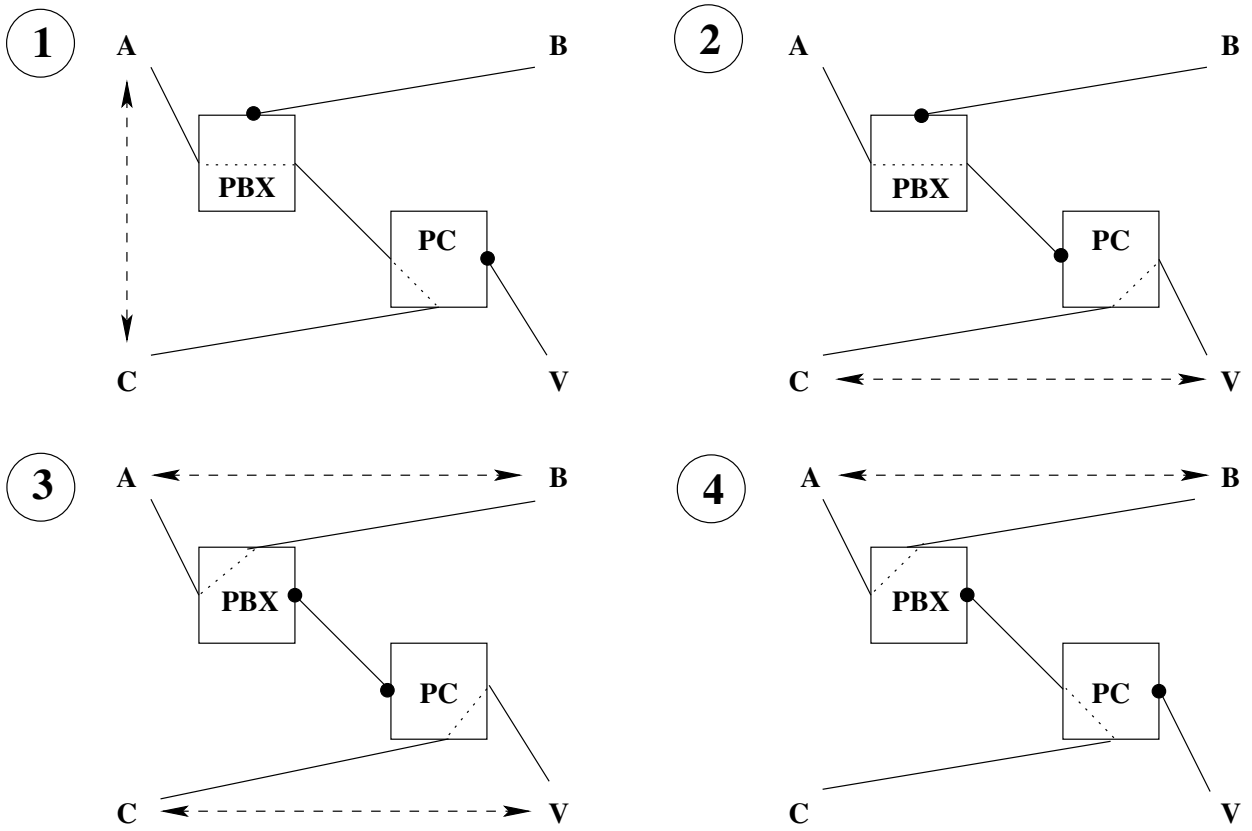

Fig. 3. Correct media control for the example, in contrast to Figure 2. Dotted lines are flowlinks, and dots are holdslots.

other servers and the rule for coordinating them. Roughly speaking, the coordination rule is that *proximity confers priority*. This means that the closer a server is to a media endpoint, the higher priority it has in controlling media flow to and from that endpoint.

Figure 3 has the same four snapshots as in Figure 2; dashed lines show the media flow that results from each goal state in the servers. In this example, the semantics of the primitives and the coordinating rule can be characterized as follows: there is media flow between two media endpoints if and only if both media endpoints desire it, and there is an unbroken chain of signaling channels and flowlinks between them.

To relate this behavior to proximity, consider the PBX. In every snapshot, A is media-connected to B if the PBX mandates it, and may be media-connected to C if the PBX allows it. Because the PBX is closest to A, it has priority over PC in controlling A. Only if the PBX has A linked to C do the actions of PC have an effect on A. Then A may actually be media-connected to C (Snapshot 1) or be silent (Snapshot 2), depending on the actions of PC.

III. A DESCRIPTIVE MODEL OF THE MEDIA ASPECT OF A SERVICE

A. Signaling paths

The modules involved in media control may be physical or virtual. A physical module is a physical component—usually a user device, application server, or media resource. A virtual module is a concurrent software process running within a physical component.

In this paper all modules are peers, whether physical or virtual. This means that if virtual modules within an application server appear in an instance of the descriptive model, then the application server as a whole does not appear in the descriptive model. We use the word *box* as a short synonym for "peer module involved in media control."

Boxes are connected by *signaling channels*. A signaling channel is two-way, FIFO, and reliable. A typical signaling channel between two physical components is implemented by TCP. A typical signaling channel within a physical component is implemented by two software queues. We do not discuss how the graph of boxes and signaling channels is configured, as this is outside the scope of this paper. Configuration is performed in varying ways by DFC, IMS, and SIP.

Each signaling channel is partitioned statically into *tunnels*, each of which provides a separate two-way signaling capability. Each tunnel can be used to control a separate media channel. The endpoint of a tunnel at a box is called a *slot*.

Media control requires a signaling protocol through which boxes communicate and coordinate their efforts. The signaling protocol (defined in Section VI) operates separately in each tunnel of each signaling channel. In other words, *each slot is a protocol endpoint.*

Within a box, two slots may be assigned to a *flowLink* object. A flowlink is a software object that reads all the signals from its two slots and controls all the signals written to them. A flowlink *coordinates the signals of its two slots*; its behavior and implementation are discussed in several sections of this paper.

A *signaling path* is a maximal chain of tunnels and flowlinks, where the tunnels and flowlinks meet at slots. Each signaling path corresponds, at any given time, to an actual or potential media channel between the path endpoints. Signals traveling through the path can create the media channel if it does not exist, and can destroy it if it does exist. Figure 4 illustrates the various pieces of a signaling path.

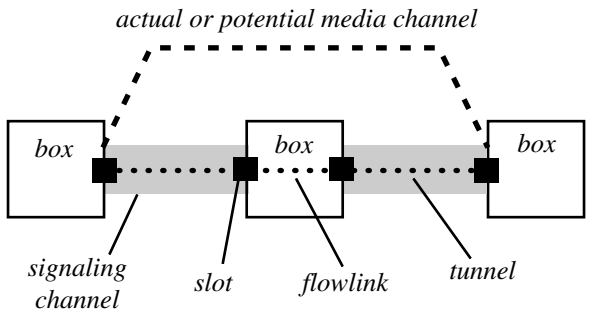

Fig. 4. A signaling path is a maximal chain of tunnels and flowlinks.

In addition to the tunnel signals that control media channels, signaling channels carry *meta-signals* that refer to the signaling channel as a whole, and can affect all the tunnels within it. Meta-signals set up and tear down signaling channels. They can indicate that the intended far endpoint is currently available or unavailable, as well as other conditions.

All of the statements about programs or protocols in this paper are made from the perspective of a particular box. For instance, a box sends a media signal or meta-signal out on a signaling channel. This signal may be intended for a user device at the far end of a signaling path. All the box program can actually do, however, is to send the signal out on the channel, to be received and processed by the next box in the chain. That box may forward it untouched toward the far endpoint, or it may not. For this reason, we usually describe the box as sending the signal "toward" the far endpoint, rather than "to" it. This tension between piecewise and end-to-end signaling is inherent to distributed applications of networks. It is at the very heart of compositional control, the goal of which is to give some behavioral guarantees to each box, even though the behavior of the overall system can be affected by every box.

The descriptive model presented here gives us a vocabulary for talking about media control, without constraining system architecture in any way. Modules can be located anywhere; any pair of modules can be co-located or not. A module can act as a media endpoint, application server, or both.

The figures in this paper provide a particular view of media services in which media endpoints are at the periphery of the system, while non-media-processing application servers are in its center. This is not the only possible view. For example, consider a media resource that is the endpoint of two separate media channels as defined here. Internally, the resource reads media packets from one channel, performs some signal processing such as transcoding on them, and writes the resulting packets to the other channel. From a user viewpoint, this resource is an application server in the middle of the system, performing some almost-transparent operation on one media stream for the benefit of two user devices at the periphery. From our viewpoint the two streams are distinguishable because they use different data encodings.

Cyclic signaling paths are not useful for controlling media channels. We assume that the configuration process prevents cycles, and do not discuss them further.

B. Media channels

Each signaling path corresponds to an actual or potential media channel between the path endpoints. If the channel exists, its global attributes include an IP address and IP port for each endpoint, which are used for sending and receiving media packets. This information is associated statically with the endpoint slot.

All other attributes of the channel are dynamic, dependent on the desires of its users, or both. Figure 5 is a nondeterministic finite-state machine specifying the user interface at one end of a media channel.

Initially the channel is *closed*, or does not exist. If the user chooses to *open* the channel, the user must choose the *medium* of the channel. Audio and video are the usual media, but there are other possibilities. For example, audio or video could be subdivided into media of different qualities. Also, text, images, or other data could be considered a medium, or one medium could encode both audio and video together.

On experiencing an *open* request, a user can *accept* or *reject* it. The channel only gets to a *flowing* state, in

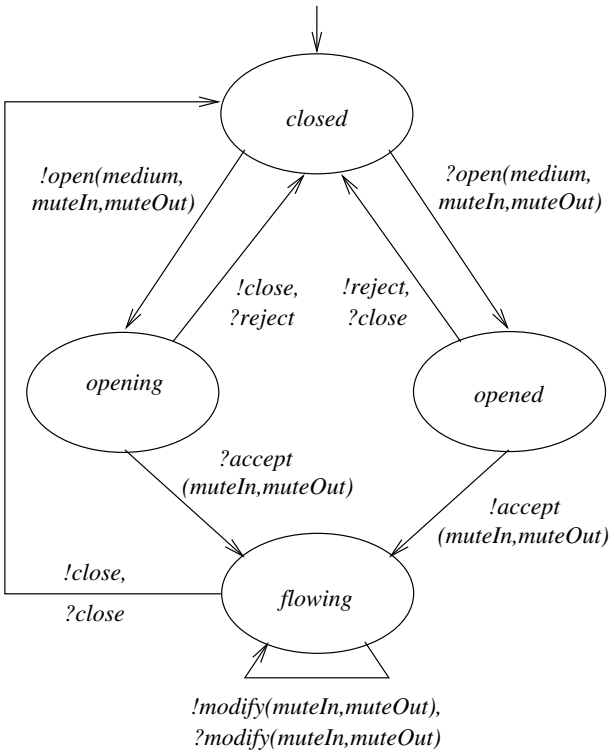

Fig. 5. The user interface at one end of a media channel. Events preceded by exclamation points are chosen by the user, while events preceded by question marks are chosen by the other end of the channel. Commas separate unrelated transition labels with the same source and sink states.

which media can flow, if one end opens it and the other accepts it. Either end can *close* the channel at any time.

A media channel always offers the potential of media flow in both directions. However, media should actually flow in a direction only if both ends desire it. Both *open* and *accept* events contain Boolean values *muteIn* and *muteOut*, indicating whether the user desires inward or outward media flow, respectively, to be temporarily suspended. Muting within a channel is dynamic, so a user can change his intention at any time, by choosing the *modify* event.[1]

It is important to note that an end of a media channel is responsible for saving and implementing the *mute* values chosen at its end only. Figure 5 shows the values chosen at one end being communicated to the other end, but the only purpose of that communication is to advise the user, whose choices might be influenced by it. If we designate the two ends of the signaling path by *left (L)* and *right (R)*, then the *muteIn* value chosen at the left end can be designated *LmuteIn*, and the value chosen at the right end can be designated *RmuteIn*. Media should flow from left to right only if $\neg$*LmuteOut*$\wedge\neg$*RmuteIn*, and media should flow from right to left only if $\neg$*RmuteOut* $\wedge$ $\neg$*LmuteIn*.

Implementation of a media channel requires that the media endpoints agree on a data format (a *coder-decoder* or *codec*) for use in each direction. Although codec choice is also dynamic, it is an implementation issue, and the user interface has no control over it.

This description of a media channel is as general as SIP's, with one exception. In SIP a media channel can be one-way only, so that media is permanently muted in the other direction. This capability could be added to our description, but it would not add any new behavior, and we omit it for simplicity.

IV. Primitives for controlling media channels

A. State-oriented goal primitives

Application programmers manipulate media channels by controlling slots. Because each slot is a protocol endpoint, the attributes of a slot are completely determined by the signaling protocol for media control (Section VI). For now it is sufficient to say that the protocol is an extension of Figure 5. Each slot has attributes *medium* and *state*, where the four states in Figure 5 are all possible slot states.

Most media services are event-driven. Many years of experience indicates that they are best programmed using finite-state machines in which the transitions are triggered by events such as received signals and timeouts. This style is almost universal in the development of telecommunication services.

To simplify media programming, we want to conceal from the programmer most of the signaling involved in media control. An application program should respond mostly to meta-signals, rather than handling each media signal individually.

This leads us to a *state-oriented* set of primitives for media control in application servers. In each state of a box program, annotations or defaults give a static description of the programmer's *goal* for each slot while the program is in that state. It is a "goal" rather than a "command" because the box must have the cooperation of other boxes and users to achieve it. If the external situation changes so that a slot should have a different goal, then the program must change to a state in which that slot is annotated differently.

Needless to say, we cannot hide slot activity from the programmer entirely, because program logic sometimes depends on it. For each slot, there are predicates *isClosed*, *isOpening*, *isOpened*, and *isFlowing* corresponding to the four states in Figure 5. These predicates can be

[1]It may seem unnecessary to have explicit control of muting, as an endpoint can simply refrain from sending media packets when it wants to "muteOut," and can simply throw away received packets when it wants to "muteIn." Nevertheless, explicit control of muting is customary in media services. It is valuable for efficiency, resource management, feature interaction, and improving the user experience.

used as guards on transitions in box programs. If a state in a box program has a transition guarded by *isClosed(s)* for some slot s, then that transition is executable as soon as the program enters the state, if s is closed at that time, or as soon as s becomes closed while the program is still in the state.

Returning to the goal primitives, there are four of them. The first three apply to one slot at a time, so they are used when a slot is acting as a path endpoint. These goals will be implemented by software objects, each of which controls a slot. Such a goal object reads all the signals received from its slot, and writes all the signals sent to its slot. The goal objects are named *openSlot*, *closeSlot*, and *holdSlot*, respectively.

The *openSlot* goal is to open a media channel and get it to the *flowing* state. The behavior of an *openSlot* goal object is a refinement of Figure 5 in which the object takes every possible opportunity to push the slot (and, by extension, the media channel) toward the *flowing* state. If an openslot sends *open* and receives *reject*, then it sends *open* again.

The medium of the channel is an argument to the *openSlot* goal object. The annotation *openSlot(s,m)* for slot s and medium m can appear in a program state only if s is in the *closed* state when the program enters the annotated state. The *openSlot* goal object sends an *open* signal with medium m. This is the only goal with a state precondition.

The *closeSlot* goal is to get its slot to the *closed* state and keep it there. Once its slot is closed, if the *closeSlot* goal object receives an *open* signal, the object sends *reject* immediately. As with an *openSlot* goal object, the behavior of a *closeSlot* object is a refinement of Figure 5 in which the object always chooses certain actions.

The *holdSlot* goal is to accept a media channel and get it to the *flowing* state, but only if the channel is requested by the other end of the signaling path. The channel will be closed if the other end closes it, and will remain closed until the other end asks to open it. This behavior is also a refinement of Figure 5.[2]

A box program can change to a state with a *closeSlot* or *holdSlot* goal object at any time in the life of the controlled slot. This means that, unlike Figure 5, these objects have no fixed initial state. When the goal object gains control the slot can be in any of its states, and the object must proceed from that point.

When any of these goal objects opens or accepts a channel, it mutes media flow on the channel in both directions. A slot in an application server may be masquerading as a media endpoint, but it is not a genuine media endpoint, and can neither send nor receive media packets fruitfully.

[2]Strictly speaking *acceptSlot* might be a better name, but we think that *holdSlot* will make more sense to service programmers.

The fourth primitive was already mentioned in Section III-A. It is a *flowLink* goal object, controlling two slots. The semantics of a *flowLink* goal are complex. Initially, its slots can be in any states. The flowlink attempts to match their states as if the slots had always been connected transparently, and to keep them matched. It has a bias toward media flow, so if a program state annotated *flowLink(s1,s2)* is entered when *s1* is in the *flowing* state and *s2* is in the *closed* state, it will attempt to get *s2* to flowing rather than closing *s1*. The semantics of a flowlink will be further elucidated by the formal specification in Section V and the implementation design in Section VII.

It is a precondition on the use of the *flowLink* goal that if both slots have the *medium* attribute defined, which means that both slots are not closed, then their *medium* attributes are the same.

B. Programming examples

Using the primitives, it is easy to program the PC behavior shown in Figure 3. In Snapshots 1 and 4, the program is in a state annotated *flowLink(c,a), holdSlot(v)*, where the slot names correspond to the endpoints they are intended to reach. A timeout event (expiration of the prepaid talk time) causes a transition to the PC state of Snapshots 2 and 3, which is annotated *flowLink(c,v), holdSlot(a)*. A signal from V that the user has paid causes a transition from this state to the other one.

Figure 6 shows a program for a Click-to-Dial box. Most guards and actions on the transitions are written informally, because they are implemented by meta-signals.

The program takes its initial transition when a user 1, who is browsing a Web site, clicks on a "click-to-dial" link. This box is configured with the address of user 1's IP telephone, and responds to the click by creating a signaling channel 1 toward the IP telephone, with a slot *1a*. The program sets a timer and enters state *oneCall*, annotated *openSlot(1a,audio)* so that the implementation will attempt to open an audio channel to the telephone. At the telephone, the *open* signal will initiate some change in the user interface, usually ringing.

The transition from state *oneCall* to state *twoCalls* is triggered by the entrance of slot *1a* into the *flowing* state, which in turn is caused by an accept action from user 1. If user 1 does not accept, the timer will eventually cause a timeout, causing the program to destroy channel 1 and terminate. Destroying channel 1 is a meta-action that of course destroys all its tunnels and slots. It is typical of single-medium applications such as Click-to-Dial that when a media channel is no longer needed, the entire signaling channel is destroyed, so that the *closeSlot* goal is seldom used.

If *isFlowing(1a)*, the program has successfully reached its first audio device, and is now ready to attempt

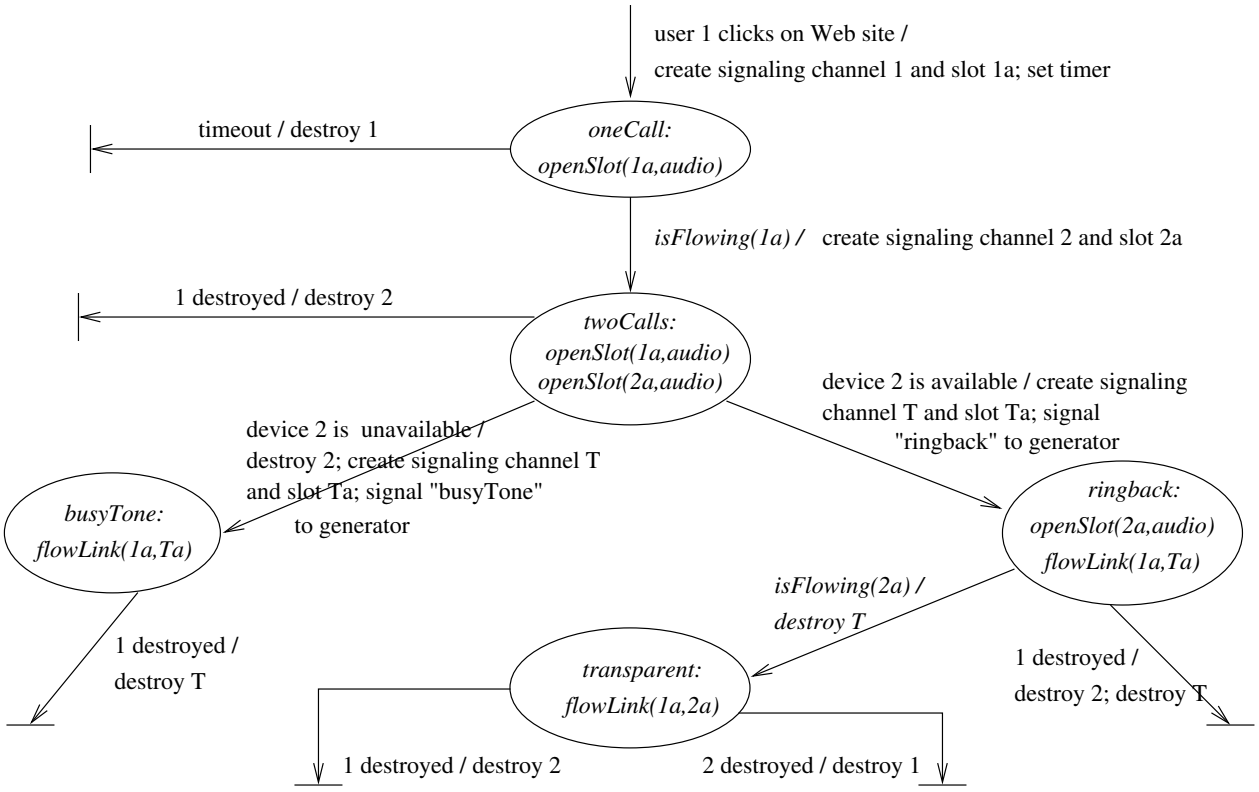

Fig. 6. A program for a Click-to-Dial box.

reaching the second one. It creates a signaling channel 2 toward the IP telephone at the clicked address on the Web site, with slot *2a*. In state *twoCalls* the goal for slot *1a* remains the same (which means control of the slot is implemented by the same object), while the program also puts slot *2a* under the control of an openslot.

In state *twoCalls* the program is waiting for a meta-signal that the telephone at the end of channel 2 is available or unavailable. If user 1 gives up in the meantime, his action will destroy channel 1, which will result in the program's destroying channel 2 and terminating.

If the device is unavailable, then the program destroys channel 2, and creates a signaling channel T to a tone-generator resource. In state *busyTone* slots *1a* and *Ta* are flowlinked. On entrance to this program state, the slot state of *1a* is *flowing*, and the slot state of *Ta* is *closed*. The *flowLink* implementation will match the states of these two slots by opening *Ta*; once the resource accepts the audio channel, it will generate a busy tone, and user 1 will be able to hear it. Eventually user 1 will abandon the call, and the program can terminate.[3]

[3]It may seem strange to implement audio tones this way, and not in user 1's telephone. The fact is that tone generation in the device is often not feasible, because the device will not generate tones when it believes it is playing the role of the called party. The implementation technique shown here is commonly used [15].

If the device is available, then the program uses the same technique as above to play a ringback tone for user 1 in state *ringback*. At the same time, it continues to try to open an audio channel to user 2. Because the annotation controlling slot *2a* is the same in both states *twoCalls* and *ringback*, the *openLink* object controlling *2a* is also the same.

Finally, if and when *isFlowing(2a)*, the program will destroy channel T and flowlink slots *1a* and *2a*. The *flowLink* implementation will automatically reconfigure IP addresses, ports, and codecs so that user 1 and user 2 can talk to each other.

Although conferencing is used by many applications for many different purposes, the implementation of conferencing always looks approximately the same. The signaling graph of a three-way audio conference is shown in Figure 7. The conference server is an application server, while the conference bridge is a media resource that performs audio mixing.

As can be seen in the figure, during the conference the conference server flowlinks the tunnel for each user device to a tunnel leading to the bridge. Each tunnel corresponds to a two-way audio channel. In the direction toward the bridge, an audio channel carries the voice of a single user. In the direction away from the bridge, an

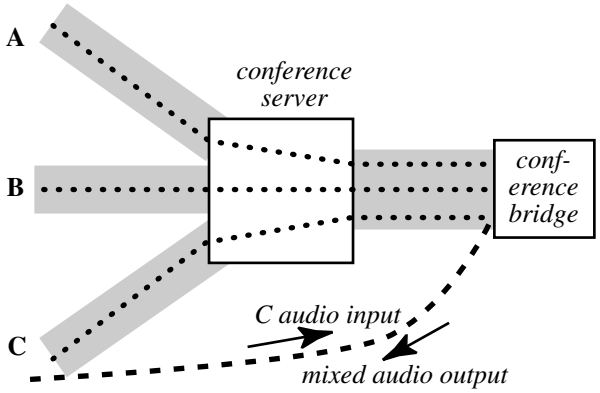

Fig. 7. The signaling model of an audio conference. Only the media flow for endpoint C is shown.

audio channel carries the mixed voices of all the users except the user the channel goes to.

Various conference applications require different kinds of muting. Full muting separates one user from the conference entirely. The conference server can accomplish this by temporarily replacing a flowlink by two holdslots.

Partial muting is more interesting and more varied. If the conference is a large business meeting, it may be desirable to mute the audio input from nonspeaking participants, so that they can hear the meeting, but background noise at their locations does not degrade overall audio quality. If the conference is part of IP-based emergency services, on the other hand, A may be a call-taker, B may be a person who has called emergency services, and C may be an emergency responder in the police or fire department. In this case it must be possible to retain the audio input from B while muting the conference output to B, so that B cannot hear what the emergency personnel are saying [2]. This is the opposite of business muting.

For a final example of partial muting, let A be a new customer-service agent, B be a customer calling for service, and C be the supervisor of A. In a training situation, the requirement is that A and B can hear each other clearly, C can hear both of them clearly, B cannot hear C, and A hears a whispered version of what C is saying. This enables C to advise A without being apparent to B.

The four primitives cannot achieve any of these partial-muting scenarios directly. They can be achieved easily by the conference bridge, however, because they are just different mixes of the three audio inputs. The application server simply connects all the user devices to a media server (conference bridge), and uses standardized meta-signals to tell the media server how to mix them [10].

Our final example shows a scenario in the use of collaborative television (Figure 8). The scenario is taken from [11], although our approach to the application is more distributed and compositional than the architecture proposed there.

In this scenario, endpoint A is a large television in a family room. C is a laptop in a daughter's bedroom. They are sharing a particular movie, which means that both are seeing the same movie at the same time point in the movie. The signaling channel from the collaborative-control box for A to the movie server is associated in the server with this movie and time pointer.

This signaling channel has five active tunnels controlling five media channels. Because they are all in the same signaling channel, the media is all from the same movie at the same time point. There are video and English audio channels for the two video devices, which differ because the two devices have different media quality and use different codecs. There is also a French audio channel to the headphones of a French-speaking friend in the family room (endpoint B).

The control box for A has control of the movie, so that commands to pause or play the movie are mediated by it, and affect all five media channels. The signaling paths from all three devices to the movie server go through this box so that they are watching the movie collaboratively.

In the scenario from [11], the daughter decides to leave the collaboration and fast-forward to the end of the movie. After this change is completed, the collaboration box of C would have its own signaling channel to the movie server, associated with the same movie but a different time pointer. There would no longer be a signaling channel between the two collaboration boxes. Because C has its own collaboration box, other devices could now join and share this new view of the movie.

We believe that the four primitives are sufficient for all media programming in application servers, and we have tested them for completeness on numerous small examples. The only way to know for sure is to gain extensive experience with programming compositional IP media services. At present no one has much experience, primarily because such applications are so difficult to build.

V. Specification of compositional semantics

To specify correctness, we assume that media endpoints are programmed using the *openSlot, closeSlot,* and *holdSlot* goal primitives as presented above, with the exception that users at media endpoints have full freedom to choose the values of the *mute* flags. Programming endpoints in this state-oriented way would be far clumsier than implementing the events of Figure 5 directly, but it is no less complete. With this assumption, all box behavior reduces to the behavior of the *openSlot, closeSlot, holdSlot,* and *flowLink* primitives.

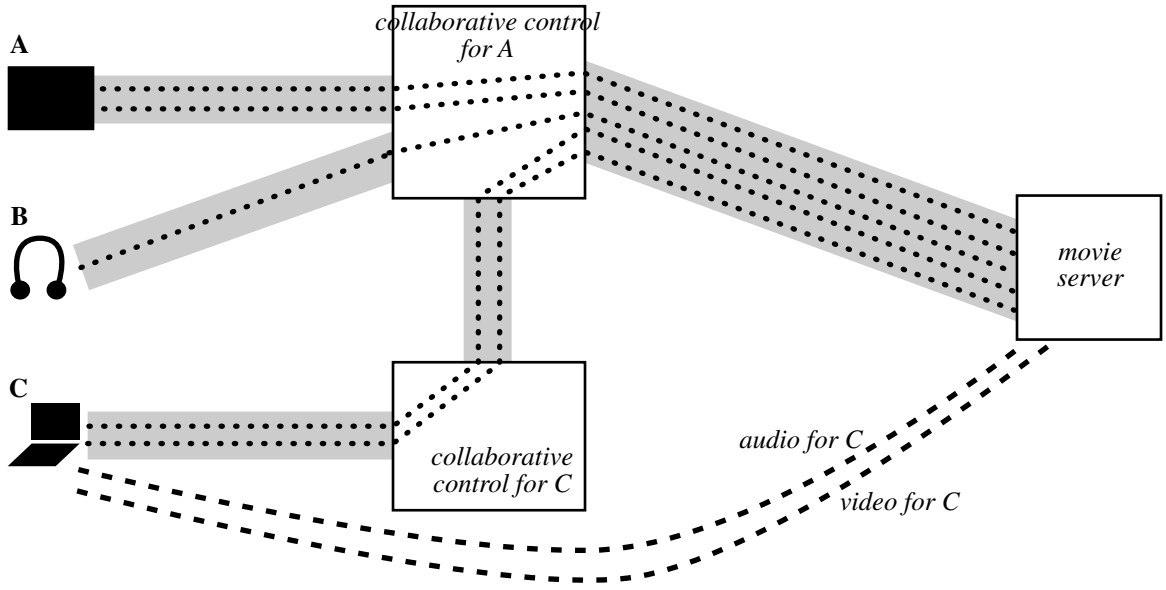

Fig. 8. A scenario in collaborative television. Only the media flow for endpoint C is shown.

The other factor that affects media behavior is the graph of signaling channels and boxes. We take this graph into account by specifying correctness in terms of individual signaling paths. Signaling paths depend on the graph of signaling channels and boxes. Signaling paths also depend on the configuration of flowlinks within boxes, determining which slots and tunnels form a path.

Signaling paths are an indirect encoding of the rule of *proximity confers priority.* This is because, from the perspective of a media endpoint, each box in a signaling path leading away from the endpoint has complete control over where the far side of the path is going. The rule of *proximity confers priority* has been used to govern media-control feature interactions in many applications built using the IP-based implementation of DFC [3], [4]. It is convenient, intuitive, and sufficient for a wide range of applications, provided that there is enough control of the configuration graph in which proximity is measured.

For each signaling path, we specify correct behavior in terms of stability or recurrence properties in temporal logic. This is necessary because a set of signaling paths is a snapshot of a system, and can change at any time as the flowlinks change. Stability properties express the idea that if a particular path is allowed to persist long enough, the goal primitives and protocol will do their work, and eventually achieve a desired path state. Recurrence properties express the same idea, plus the additional idea that if something is perturbed while the path persists, the path state will eventually adjust to the perturbation. If the system is thrashing and paths do not persist long enough to stabilize, then this specification of correctness does not say anything about their behavior.

For convenience, we identify the two ends of a signaling path as *left (L)* and *right (R).* For each path, there are two stable states that we might wish to achieve. The first is the *bothClosed* state, in which both endpoints are in the *closed* state and there is no possibility of media flow. This path state is defined as

$$Lclosed \wedge Rclosed$$

where the predicates refer to the endpoint states as defined in Figure 5. The second stable path state is the *bothFlowing* state, in which both endpoints are in the *flowing* state. To specify a correct state completely, we need to ensure that the *medium* attribute of both endpoints is the same. We also need to ensure that the implementation state correctly reflects the *mute* attributes of the endpoints. The implementation state of the endpoints is captured by the new history variables *Lenabled* and *Renabled*. If *Lenabled* is true, both endpoints are ready for packets in the right-to-left direction. They have agreed on a codec, they have agreed to enable transmission, and they have each other's IP address and port number. If *Renabled* is true, both endpoints are ready for packets in the left-to-right direction. The complete definition of the *bothFlowing* path state is

$$\begin{aligned} &Lflowing \wedge Rflowing \wedge (Lmedium = Rmedium) \wedge \\ &\quad (Lenabled = \neg LmuteIn \wedge \neg RmuteOut) \wedge \\ &\quad (Renabled = \neg RmuteIn \wedge \neg LmuteOut) \end{aligned}$$

The implementation of the new history variables *Lenabled* and *Renabled* is described in Section VI-C.

Each signaling path has two ends, each of which is controlled by an openslot, closeslot, or holdslot. Taking symmetry into account, there are six possible path types based on classification of their end slots. A path of a given type can have any number of tunnels and flowlinks, as these should be transparent with respect to observable behavior.

If one end of a path is controlled by a closeslot and one end is controlled by a closeslot or holdslot, then correctness is:

$$\Diamond \quad \Box \quad \textit{bothClosed}$$

This stability property in linear temporal logic says that eventually the path will reach a state in which both end slots are closed, and will remain there forever. In practice, of course, the slots are only required to remain closed until the environment changes the path in some way, at which time a different specification may apply.

The specification of a path with one end controlled by a closeslot and one end controlled by an openslot is weaker, because the path will not stabilize—the openslot will continue trying to open it. All we can be sure of is that once the objects have had a chance to do their work, there will be no media flow in either direction. This is expressed by the stability property:

$$\Diamond \quad \Box \quad \neg \textit{bothFlowing}$$

The specification for a path with one end controlled by an openslot and one end controlled by an openslot or holdslot requires that the path reach a *bothFlowing* state. However, once this state is reached, the path may leave it temporarily because a *modify* event in a user interface changes a *mute* flag. It will take time for the implementation to send the signals to restore the *bothFlowing* state.[4] Unlike the previous path specifications, this is a recurrence property, saying that the signaling path will always eventually return to the *bothFlowing* state:

$$\Box \quad \Diamond \quad \textit{bothFlowing}$$

Finally, the specification of a path with both ends controlled by holdslots is more complex because either *bothClosed* or *bothFlowing* is acceptable. (What actually happens depends on the state of the path when it was formed.) Thus the specification is a disjunction of a stability property and a recurrence property:

$$(\Diamond\Box \textit{bothClosed}) \vee (\Box\Diamond \textit{bothFlowing})$$

[4]At the implementation level, an endpoint can also change its IP address, port number, or codec choice without changing its muting. Because the implementation uses the same mechanism for all such modifications in the *flowing* state, we do not consider these other modifications separately.

All of these formulas are idealized specifications that will not be satisfied in the face of network or hardware failures. They are reasonable for our purposes, however, because there should be no defects in the software of application servers.

Bandwidth limitations would not prevent an implementation from satisfying the specification, because the specification is based on the software state at the ends of a signaling path, not on actual packet transmission. More relevantly, if there are bandwidth constraints on which media channels should be opened or accepted, then these constraints should be enforced by the endpoints and applications. Application servers should make decisions wisely, then rely on our primitives to carry out their decisions.

VI. Signaling protocol

A. Practical requirements

To set up media flow between two endpoints, as explained in Section III-B, each endpoint must know the IP address and port number that the other will be using. The endpoints must also agree on a *codec* for the media stream in each direction.

A codec is a data format for a medium. For example, G.726 is a lower-fidelity and lower-bandwidth codec for audio, while G.711 is a higher-fidelity and higher-bandwidth codec for audio. G.711 is approximately equivalent in fidelity to circuit-switched telephony. Note that it is not necessary for the two directions of a channel to use the same codec.

Although many endpoints can interpret more than one codec, it is still important for them to know which codec they are expected to interpret at a given time. This is because they allocate resources dynamically to whichever codec they are using, and need to reconfigure before changing codecs. Surprising as it may seem, media sources may wish to send using different codecs even within the same media episode. For example, a resource that plays recorded speech may have speech files that were stored in several different codecs.

We use *noMedia* as the name of a distinguished pseudo-codec indicating no media transmission. For simplicity of presentation, we assume that any two devices supporting the same medium have at least one real codec in common.

The separation of signaling and media channels can cause synchronization problems. Media *clipping* results when media packets are lost because they arrive at an endpoint before the endpoint is set up to receive them. Clipping should be minimized, although it is not always cost-effective to eliminate it entirely.

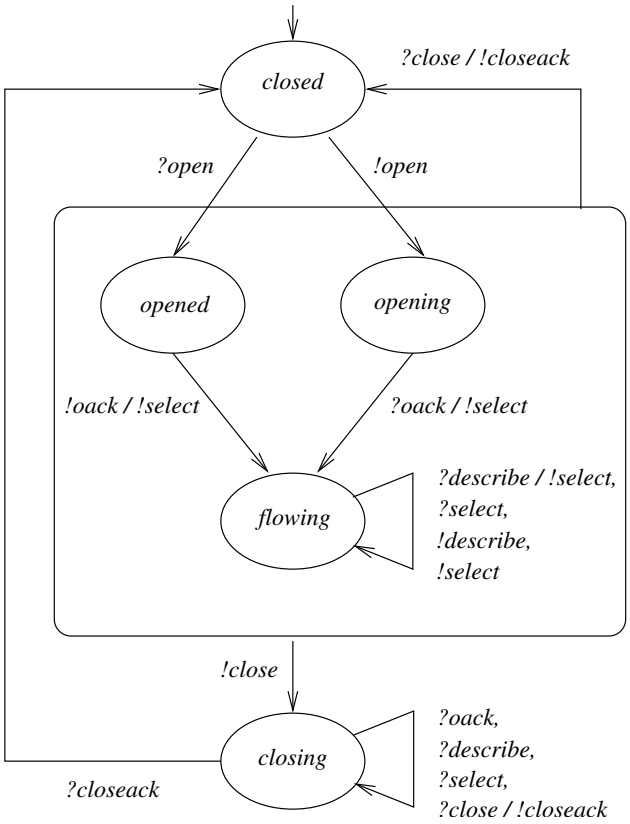

Fig. 9. Specification of the protocol at each protocol endpoint. *?* means *received*, *!* means *sent*. *?oack / !select* means send *select* if and when *oack* is received. *!oack / !select* means send the two signals in sequence. Commas separate unrelated transition labels with the same source and sink states.

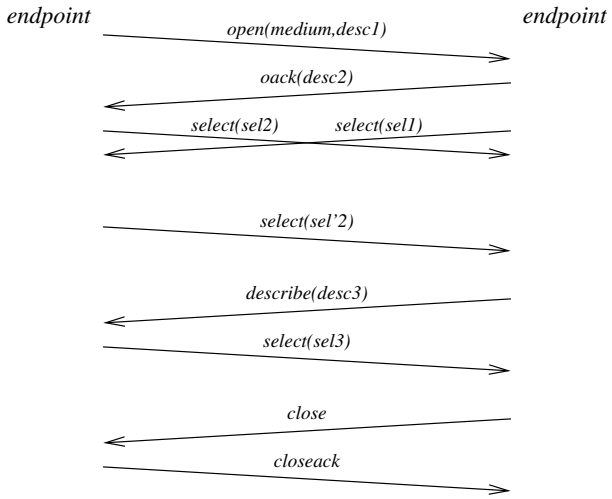

Fig. 10. Use of the protocol.

B. Protocol definition

Recall from Section III-A that the actual scope of the protocol is one tunnel in one signaling channel. For reference as the protocol description proceeds, Figure 9 shows a finite-state machine specification of the protocol at each protocol endpoint, i.e., slot.

At the same time, the use and meaning of the protocol is best described as if the protocol endpoints were media endpoints, which is what we will do. Figure 10 is a scenario in which the protocol is used to open, modify, and close a media channel between two media endpoints. Figure 10 represents a signaling path in which there are no flowlinks. The conceptual gap between the piecewise and end-to-end views must be bridged by the correct operation of application servers.

Either end of a tunnel can attempt to open a media channel by sending an *open* signal. The other end can respond affirmatively with *oack* or negatively with *close*. Either end can close the media channel at any time by sending *close*, which must be acknowledged by the other end with a *closeack*. Note that *close* now plays the role of both *close* and *reject* in Figure 5.

Each *open* signal carries the medium being requested, and a *descriptor*. A descriptor is a record in which an endpoint describes itself as a receiver of media. A descriptor contains an IP address, port number, and priority-ordered list of codecs that it can handle. If the endpoint does not wish to receive media, i.e., *muteIn* is true, then the only offered codec is *noMedia*.

Each *oack* signal also carries a descriptor, describing the channel acceptor as a receiver of media. These descriptors are shown in Figure 10 but not in Figure 9.

A *selector* is a record in which an endpoint declares its intention to send to the endpoint described by a descriptor, and indicates the codec it will be using. A selector contains identification of the descriptor it is responding to, the IP address of the sender, and the port number of the sender. If the selecting endpoint does not wish to send media, i.e., *muteOut* is true, then the selector contains *noMedia*; otherwise, it contains a single codec selected from the list in the descriptor. For optimal codec choice, the sender should choose the highest-priority codec that it is able and willing to send. The only legal response to a descriptor *noMedia* is a selector *noMedia*.

When a channel is first being established, the opened end sends an *oack* signal and then a *select* signal carrying a selector. The selector is a response to the descriptor in the *open* signal. The initiator's response to the descriptor in the *oack* signal is carried in another *select* signal. In Figure 10, descriptors and selectors have numbers to indicate which selector is responding to which descriptor.

Either endpoint can send media as soon as it has sent a selector with a real (not *noMedia*) codec. An endpoint should be ready to receive media as soon as it has received a selector with a real codec. This is the most

relaxed approach to synchronization of the signaling and media channels.[5]

At any time after sending the first selector in response to a descriptor, an endpoint can choose a new codec from the list in the descriptor, send it as a selector in a *select* signal, and begin to send media in the new codec. In Figure 10, *select(sel'2)* shows this possibility.

At any time after sending or receiving *oack*, an endpoint can send a new descriptor for itself in a *describe* signal. The endpoint that receives the new descriptor must begin to act according to the new descriptor. This might mean sending to a new address or choosing a new codec. In any case, the receiver of the descriptor must respond with a new selector in a *select* signal, if only to show that it has received the descriptor. In Figure 10, *descriptor3* and *selector3* illustrate this interaction.

It is possible that a race condition, with two *open* signals traveling in opposite directions, could occur within a tunnel. The race is easily detectable by both slots, because each sends an open and receives an open in return. In this case the winner of the race is always the end of the tunnel that initiated setup of the signaling channel, which is fixed and unambiguous. The losing *open* signal is simply ignored. This aspect of protocol behavior is not illustrated in Figure 9.

C. Properties of the protocol

At each end of a signaling path, the user interface (Figure 5) translates straightforwardly to its protocol implementation (Figure 9). There is an extra protocol state *closing* not observable in the user interface. *Accept* events are replaced by *oack* signals. *Modify* events are replaced by *describe* and *select* signals. The values of the *mute* variables are communicated through descriptors, as presented in the previous section.

The history variable *Lenabled (Renabled)* is initially false. It becomes true when the left (right) endpoint of the signaling path sends a selector with a real (not *noMedia*) codec. It becomes false again when the left (right) endpoint leaves the *flowing* state or sends a selector with *noMedia* as the codec. As required by Section V, when *Lenabled (Renabled)* is true both endpoints are ready for packets in a left-to-right (right-to-left) direction: they have agreed on a real codec and have each other's IP address and port number.

A *describe* signal makes it possible for a media endpoint to change its characteristics as a receiver of media. This is sometimes useful, but—because the protocol is used piecewise, and every box is a protocol endpoint—most *describe* signals are sent by application servers.

For example, consider the transition from Snapshot 1 to Snapshot 2 in Figure 3. To implement this transition, PC sends a *describe* signal with *noMedia* to A, a *describe* signal with the descriptor of C to V, and a *describe* signal with the descriptor of V to C. (PC has these descriptors available because it has recorded them as they passed through in previous signals.) The answering *select* signal from A is absorbed by PC, and the answering *select* signals from C and V are sent to each other. These signals will cause the actual media paths to change as indicated in the figure.

To make media control as easy as possible, *describe* signals (and their answering *selects*) going in opposite directions of the same tunnel do not constrain each other. This means that changes initiated in both directions can proceed concurrently. There is no need to introduce the complexity and overhead of serializing them.

Another simplifying design decison is that the protocol has no enforced pairing of *describe/select* signals relevant to media transmission in one direction. A *describe* can be sent at any time, even if no *select* has been received in response to the last *describe*. A *select* can be sent at any time, even if no *describe* has been received since the last *select* was sent. This makes box state simpler and eliminates unnecessary constraints.

This protocol was designed specifically to facilitate composition. It is radically different from SIP [7], [14], which is the dominant protocol in use for media control. Section IX compares the two and explores the consequences of their differences.

VII. Implementation software

Implementation of compositional media control requires Java code resident in each application server. *Box* objects contain the high-level code that calls on *Goal* and *Slot* objects when necessary. Figure 11 shows the hierarchical structure of method invocations among *Box, Goal,* and *Slot* objects.

A *Slot* object sees all signals received from a slot and sends all signals to the slot. Because of this complete view, it is able to maintain the complete implementation-level state of the slot, consisting of protocol state, *medium,* and *descriptor.* The descriptor of a slot in an application server is the most recent descriptor received in an *open, oack,* or *describe* signal.

The first action of a goal object is to query its slots, using *slotState* and *slotDesc,* to get their protocol states and descriptors. Then, having completed this initialization, the goal object proceeds to control its slot or slots until its slots are moved elsewhere and this goal object becomes garbage.

[5]To make absolutely sure that no media is lost, even if media packets travel through the network faster than signals, an endpoint must begin "listening" for media in accordance with a descriptor as soon as it has sent the descriptor, and must be able to accept packets in any allowed codec at any time. This is possible because codecs are self-describing. It is easier, however, for an endpoint to wait for *select* signals and risk the loss of a few packets that arrive before their corresponding selectors.

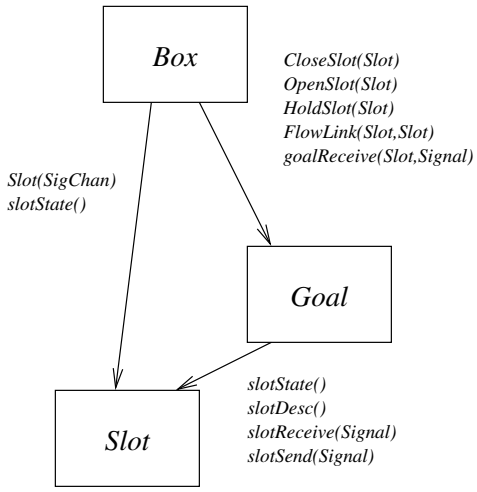

Fig. 11. The hierarchy of method invocations among Java objects.

There is also a *Maps* object that maintains the dynamic association between slots and goal objects. When a box receives a signal, the box uses these associations to find the goal object to which it should show the signal via *goalReceive*.

The *openSlot*, *closeSlot*, and *holdSlot* programs are all reasonably straightforward, because each controls a single slot. The code of each is structured as a finite-state machine that follows the structure of Figure 9. The design of the *flowLink* code, as described below, is considerably more complex.

The primary organization of the *flowLink* code is based on slot states. There is a flowlink state for each pair of slot states; any combination of slot states is possible because a flowlink can be instantiated to control two slots that were previously independent. Based on its pair of slot states, the flowlink performs *state matching* as shown in Figure 12. The state labels use the shorthands *live* and *dead*, as defined in the caption.

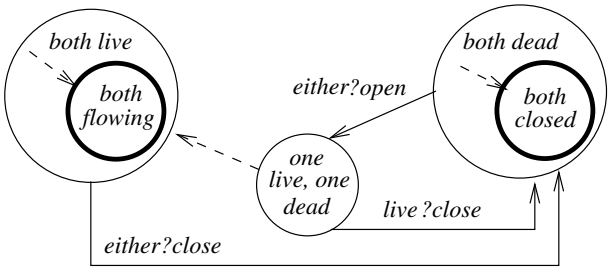

Fig. 12. State matching in a flowlink. The *live* states are *opening, opened* and *flowing.* The *dead* states are *closed* and *closing*.

The dashed arrows indicate the work of state matching, which consists of sending signals and waiting for signals, as needed, to push toward a goal. Which of the three superstates the flowlink is in, at any time, is chosen by the flowlink's environment. This is because the superstate depends on the *open* and *close* signals that the flowlink receives. The dashed arrows show that the flowlink works from whichever of the three superstates it is currently in to one of the two heavily outlined substates. These are the two goal states *both flowing* and *both closed*.

There is a close relationship between Figure 12 and the formal specification of path semantics. To see this relationship, it is necessary to realize that:

- a *closeSlot* object emits *close* signals, and never *open* or *oack* signals;
- an *openSlot* object emits *open* and *accept/oack* signals, and never *close* signals;[6]
- a *holdSlot* object emits *accept/oack* signals, and never *open* or *close* signals.

For each type of path, the objects at its endpoints determine which signals will be coming toward the flowlinks. For each type of path, these signals lead to one or two goal states in Figure 12, and these are the same goals as found in the temporal formula for that type of path.

The secondary organization of the flowlink is based on descriptors. A flowlink caches the most recently received descriptor of each of its slots. The code design is built around two concepts:

- A slot is *described* if the object has received a current descriptor for it. Slots in the *opened* and *flowing* states are *described*, while slots in other states are not.
- Each slot has a Boolean variable *up-to-date (utd)* that is true if and only if the other slot is described and this slot has been sent its most recent descriptor.

In any *live* state, the flowlink is working to make the *utd* variables true. This depends on the slot states, because an *utd* variable can only be true if the other slot is described, and if its slot is in a state allowing sending of a new descriptor if necessary.

To see how these structures make a bewildering array of cases comprehensible, consider a flowlink state in which slot 1 is *flowing* and slot 2 is *opening*. This state could not have arisen if the two slots were always flowlinked to each other; it could only have arisen if one or both slots were previously linked elsewhere.

From the perspective of this flowlink, either (1) slot 2 was *opening* when it entered the flowlink, or (2) slot 2 was originally dead and the flowlink made it live by sending an *open*. Both *utd* variables in a flowlink are initialized as false. In Case 1, when an *oack* is received from slot 2, slot 2 will be in the *flowing* state but with

[6]Usually it emits only *open* signals, but in the case of a race between two opens in a tunnel, it may back off and be the acceptor instead.

utd2 = false. This makes sense because the descriptor carried by the *open* signal that opened slot 2 had nothing to do with this flowlink. To set *utd2 = true*, the flowlink must send *describe* with the descriptor of slot 1.

In Case 2, when an *oack* is received from slot 2, it will reach the *flowing* state. Is slot 2 up-to-date? Variable *utd2* became true when the flowlink sent it an *open* signal with the then-current descriptor of slot 1. If it is still true, there is no more work to do.

However, between sending *open* to slot 2 and receiving *oack* from it, slot 1 may have received a new descriptor for some reason. If it has, then *utd2* has been set to false again. It can be made true again by sending slot 2 a *describe* with the new descriptor of slot 1. This shows how the relevant history of three different cases is maintained concisely by the *utd2* variable.

Interestingly, handling of *select* signals is much simpler. In all cases in which a flowlink succeeds in reaching its goal state of *both flowing*, it sends a descriptor of the other slot to each slot. A selector always responds to a descriptor, and in the *both flowing* state both slots have received fresh descriptors. This means that only fresh selectors matter, so the flowlink need not keep any history of them.

When a flowlink receives a selector and is in a state to forward it to the other slot, it checks before forwarding that the selector is a response to the other slot's descriptor. If it is not a proper response, then the selector is obsolete and is discarded.

VIII. Verification and performance

A. Partial verification

Partial verification has been performed by modeling the Java code in Promela and checking the Promela models with the Spin model checker [8].

The scope of each model is a signaling path. We modeled and checked 12 signaling paths: six paths with no flowlinks and every possible combination of closeslots, openslots, and holdslots at their ends, and six paths similar to the first six paths but with one flowlink each.

In every Promela model, every slot is controlled by a goal object. Each goal object has two phases. In a goal object's initial phase, the behavior of the slot or slots it controls is allowed to be completely nondeterministic, and to have nothing to do with the goal. At some nondeterministically chosen point, the goal object switches permanently to a second phase in which it behaves according to the specified goal. Because of the existence of the nondeterministic initial phases, model checking covers traces in which the goal objects begin their real work in all possible initial states of the slots and of the signaling tunnels that connect the slots.

We performed two checks on each model. First, a safety check was run to make sure that the path model had no deadlocks or other abnormal terminations. The check ensured that in any final state, each slot is *closed* or *flowing*, and all signaling channels are empty. Second, we verified that each model satisfies its specification as given in Section V.

The definition of the *bothFlowing* path state in Section V is:

$$\begin{aligned} &\mathit{Lflowing} \wedge \mathit{Rflowing} \wedge (\mathit{Lmedium} = \mathit{Rmedium}) \wedge \\ &\quad (\mathit{Lenabled} = \neg \mathit{LmuteIn} \wedge \neg \mathit{RmuteOut}) \wedge \\ &\quad (\mathit{Renabled} = \neg \mathit{RmuteIn} \wedge \neg \mathit{LmuteOut}) \end{aligned}$$

The definition that we used in model checking is somewhat different:

$$\begin{aligned} &\mathit{Lflowing} \wedge \mathit{Rflowing} \wedge \\ &(\mathit{LdescRcvd} = \mathit{RdescSent}) \wedge (\mathit{RdescRcvd} = \mathit{LdescSent}) \wedge \\ &\quad (\mathit{LselRcvd} = \mathit{LdescSent}) \wedge (\mathit{RselRcvd} = \mathit{RdescSent}) \wedge \end{aligned}$$

This definition uses history variables that store the descriptors and selectors most recently sent and received at path endpoints. It abuses notation slightly in assuming that a selector responding to a descriptor is "equal" to it. The definition says that in the *bothFlowing* state, each end has most recently received the descriptor most recently sent by the other end, and each end has most recently received a selector responding to its own most recent descriptor. From this definition and the rules about protocol behavior in Section VI, it is easy to derive the original definition of *bothFlowing*.

It may not be feasible to model-check signaling paths with more than one flowlink. Even with partial order reduction, compression, and a few simplifying assumptions, a typical check of a signaling path with a flowlink takes 20 minutes and 3 Gb of memory on a Sun Solaris M9000 SMP machine with dual-core 2.4 GHz SPARC processors.[7] More importantly, when we compare similar checks of two paths, varying only in that one has a flowlink and the other does not, adding a flowlink causes the memory to grow by a factor of 300 on the average, and the time to grow by a factor of 1000 on the average. This suggests that checking a path with two flowlinks might take something like 900 Gb of memory and 300 hours. Even if these numbers over-estimate the impact of another flowlink by an order of magnitude, they are still forbidding.

B. Toward complete verification

Even if it were possible to model-check signaling paths with several flowlinks, this would still not yield

[7]The variance among checks is considerable. The biggest check took 161 minutes and 19 Gb of memory.

a proof of correctness for signaling paths of any length. Such a proof can only be constructed inductively.

Although we leave this inductive proof for future work, it seems that the most promising approach would be to construct the proof in terms of lemmas that could be verified by model-checking. For example, imagine a lemma whose scope is an arbitrary contiguous segment of a signaling path, no larger than two tunnels and three boxes (in other words, a segment with no more than one internal flowlink). It is plausible that such a lemma could be verified by model-checking, and could be used inductively to prove a theorem over whole signaling paths of any length.

In the meantime, we give an informal argument that a signaling path with openlinks on both endpoints always converges to the correct state. This argument does not focus on getting slots to the *flowing* protocol state, but rather on the exchange of descriptors and selectors. The argument is illustrated by the message-sequence chart in Figure 13. It shows a scenario in which, starting from Snapshot 3 of Figure 3, PC completes authorization and the PBX switches back to C at about the same time. This means that the transition from Snapshots 3 to 4 (in which PC changes linkages) and the transition from Snapshot 4 back to Snapshot 1 (in which the PBX changes linkages) are proceeding concurrently.

Control of media flow in each direction is independent and symmetric. Thus, without loss of generality, we consider only media flow from A to C.

The new flowlink in the PBX begins by sending to the left its most recent descriptor from the right, which is *noMedia*. When A receives the *describe* signal, it responds with *select(noMedia)*.

Concurrently, the new flowlink in PC begins by sending to the left its most recent descriptor from the right, which is that of C. When the flowlink in the PBX receives it, to remain up-to-date, it forwards *describe(C)* to the left. This new descriptor supersedes *noMedia* at A, so A replies with *select(C)*.

Along the entire signaling path from left to right, *select(C)* matches the most recent descriptor from the right, and is therefore forwarded all the way to C. Now media flow from A to C is fully established.

To generalize from this example, after a signaling path stabilizes, eventually the descriptor of an endpoint will propagate along the entire signaling path as the most recent descriptor from that end. When it reaches the other end, the other end will respond with a new selector. Because the descriptor has now stabilized along the path, the selector will be accepted and forwarded by each box in the path.

To reiterate a point made in Section V, if signaling paths do not remain stable long enough for the distributed algorithm to converge, then nothing can be expected of the system.

C. Performance

We have tested our Java code with a suite of test drivers, as a check on aspects of the code not modeled in Promela. Unfortunately, it is not possible to test the code on live IP media. The reason is that testing on live media would require a great deal of hardware and software infrastructure, all of it using this protocol. Although we have developed such an infrastructure in our laboratory and are accustomed to using it for live media [3], [4], the infrastructure is all based on SIP. The significance of this situation is discussed in Section IX.

Fortunately, the behavior of our protocol is simple enough so that its performance can be studied analytically. The most important performance measure for a signaling protocol is its latency. In our protocol, an endpoint can transmit media as soon as it has received a descriptor and sent a corresponding selector.

We now consider the latency of the protocol in the compositional scenario of Figure 13. Let c be the average time it takes for a server to read a new stimulus from an input queue and compute the next signal to send. Let n be the average time it takes for the network or server infrastructure to accept a signal and deliver it to its destination box. In Figure 13 both endpoints can transmit after an average delay of $2n + 3c$.

The actual values of n and c can vary greatly. However, to make the analysis more concrete, a typical value of c might be 20 ms. In a few experiments on a typical carrier network with multiple geographic sites, n averaged 34 ms. With these numbers the latency of Figure 13 is 128 ms.

This latency is not directly affected by other activity in the system, such as changes to other media channels controlled by the same signaling path (needless to say, it can be affected indirectly by processor overload). Because its behavior is fundamentally simple, the analysis can be generalized as follows.

The latency of providing media flow from a signaling path should be measured from the moment that the last flowlink in the path is initialized. Before this moment the path did not exist, and all its other flowlinks were elements of other signaling paths. From the pattern of Figure 13, it is easy to see that the average signaling delay after that moment will be

$$pn + (p + 1)c$$

where p is the number of hops between the last flowlink and its farther endpoint. In Figure 13 p is the path length minus 1, which is the maximum for any path.

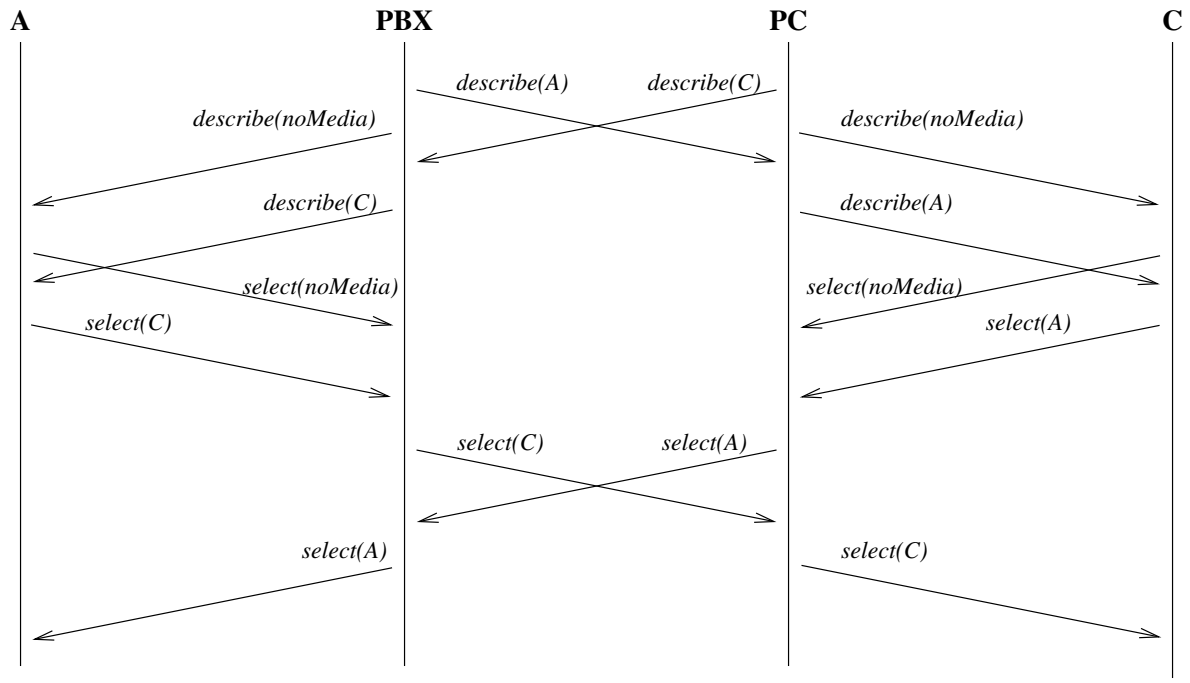

Fig. 13. Behavior of the solution when the PBX and PC change state concurrently.

IX. Related work

A. Research on media control

Very little work has been done on compositional media control, because the problem is not yet widely recognized. In the public circuit-switched telephone network compositional media control is easy, because there is no signaling/media separation. There is only one path between communicating endpoints, carrying both signals and media; that path can be switched by any network element it goes through, using stricly local manipulations.

As explained in Section I, compositional media control is often needed when different administrative domains, each with their own application servers, interoperate. In IP-based telecommunications ("voice-over-IP" or "VoIP") there are many administrative domains, but there are almost no instances in which these domains interoperate directly. Rather, all VoIP domains interoperate with the circuit-switched network, so that the VoIP domains only interoperate with each other through the circuit-switched network! A telephone call from one VoIP domain to another may have separate signaling and media paths in each VoIP domain, but those paths will join where the call passes through the circuit-switched network. Clearly compositional media control will be needed for VoIP to reach its full potential.

The problem of feature interaction in telecommunication systems is well known [5]. Controlling media is one of the things that features do, and affecting the same media channels is one of the ways that features interact. Nevertheless, there has been little previous work on media-related feature interactions, because most research on feature interactions does not consider features of sufficient complexity.

In the first IP-based implementation of DFC [4], the problem of compositional media control is addressed in a limited way. In a particular application server or cluster of cooperating application servers, all of the boxes report all changes to the graph of signaling channels and boxes, and all changes to the links within the boxes, to a central media-control module [6]. The media-control module maintains an explicit graph of the media state; this graph has the same information as graphs like Figure 3. Based on the current graph, the module computes what the media state should be and acts as a central controller to achieve it.

This approach has two obvious deficiencies: (1) the central computation is a performance bottleneck, and (2) the approach does nothing to coordinate the actions of servers that are administratively independent. Less obvious but also important is a third deficiency: (3) codec choice is managed in an *ad hoc* manner that does not guarantee an optimal result.

Despite these deficiencies, our first approach was successful enough to support compositional development and deployment of a feature-rich, nation-wide, consumer voice-over-IP service [3].

The solution reported in this paper, first presented

in [16], is different in almost every detail. The media-control protocol has been improved to support optimal codec choice and easier composition. The API has been refined and formally specified. Most important of all, we have discovered a way to implement compositional media control in a distributed, rather than centralized, fashion. And we have taken significant steps toward verifying our implementation.

Today most IP-based media applications are being built using SIP [7], [14]. SIP was designed for communication between endpoints, with little or no logical processing in the network between them. This anti-compositional design philosophy was typical in the early days of the Internet. The problem for industry today is that prospects and plans for IP media have moved beyond what was envisioned in those days. As a result, vendors are finding it increasingly difficult to deploy services and ensure interoperation of system components.

In an attempt to bridge the gap, a document on best current practices for SIP [13] explains how a SIP application server, acting as an intermediary, can control the media streams of endpoints. However, the document does not have any mention of the problem of composition. All of the examples assume that the server being discussed is the only server in the signaling path. All of the signaling techniques are presented only as orderly scenarios in which everything proceeds in the best way, with no events occurring at inconvenient times.

In the next section, we compare SIP to our protocol with respect to some of the relevant criteria.

B. Protocol comparison

First we compare our protocol and SIP using an end-to-end perspective. From this perspective, media channels are opened, closed, and modified only by their endpoints. There are three major differences between the protocols.

The first difference concerns basic synchronization. With respect to basic synchronization, the design of SIP was based on HTTP, and is therefore *transactional*. When SIP signals travel in unreliable UDP datagrams, which is allowed but not recommended, SIP agents use the transactional structure to recover from lost signals.

In SIP, a media channel is opened or modified by a three-signal transaction composed of an *invite* from an endpoint, a *success* from the other end, and an *ack* from the initiator. Such an invite transaction cannot overlap with any other invite transaction on the same signaling path. If a race between two invite transactions is detected, both fail immediately. The initiator of each failed *invite* is supposed to wait for a randomly chosen period and then retry.

Our protocol is not transactional. It depends on an underlying mechanism such as TCP for reliability. Once a channel has been opened, an agent can send new *describe* or *select* signals at any time. The signals that modify media flow in one direction are completely independent of the signals that pertain to media flow in the other direction. Our protocol can be described as *idempotent*, because *describe* and *select* signals provide updated information without changing the fundamental state, which is why they can be so unconstrained.

In terms of both programming complexity and performance, there is a great difference between these two protocol designs. In our protocol, if an endpoint needs to modify a media channel, it simply sends a new *describe* immediately. When the other endpoint receives the descriptor, it can send a corresponding *select* immediately.

In SIP, on the other hand, initiating a media change can be arduous. First the endpoint must wait for any ongoing transaction that it knows about to complete. Next the initiator sends an *invite* signal. If it next receives a corresponding *success*, it can send *ack* and be finished. If it receives an *invite*, however, it has detected a race. The initiator must receive and acknowledge *failure*, wait for some period, check that there is no ongoing transaction, and start over.

The second difference concerns codec choice. For codec choice, SIP uses a *negotiation* model. To open a media channel or modify an existing one, an endpoint sends in its *invite* signal an *offer* containing a set of possible codecs that it can handle. The responder sends in its *success* signal an *answer* that is a subset of the offer codecs, all of which the responder can handle. Henceforth any of the codecs in the answer subset can be used.

Our protocol decouples codec choice in the two directions, as there is no technical necessity to use the same codec in both directions. In each direction, one endpoint sends a set of possible choices in a descriptor, and the other end chooses one in a selector. This has the advantage of guaranteeing that, once an endpoint has received a selector, it knows exactly which codec it is expected to interpret.

In our protocol, a descriptor is a *unilateral* description of an endpoint. In SIP, an answer is a *relative* description of an endpoint, i.e., a description of one endpoint with respect to (in negotiation with) another. The decision to make a protocol transactional or idempotent, and the decision to choose codecs by negotiation or unilaterally, are not independent. Negotiation requires a transaction, while unilateral codec choice does not.

Negotiation takes a toll on performance, because the description of the responder to the initiator (in an answer) must follow the description of the initiator to the responder (in an offer). With unilateral codec choice, descriptors in both directions can travel concurrently, and

selectors in both directions can travel concurrently. This is illustrated by Figure 13. When we consider the case of application servers, we will see that negotiation has additional costs in increased latency and programming complexity.

The third difference between our protocol and SIP concerns media bundling. We consider every tunnel within a signaling path, controlling one media channel, to be completely independent of every other tunnel. Each SIP signal for controlling media, in contrast, refers to all media channels of the path simultaneously. It contains a list with an entry for each potential media channel, so that a list entry has the same purpose as a tunnel.

SIP's media bundling makes it more difficult to program an application server that splits or joins media channels controlled by a signaling path, such as a collaborative control server in Figure 8. Using SIP, the server must reconstruct every media signal that passes through it, because the bundles are different on each side.

Another problem with media bundling is that it increases the probability of race conditions between transactions, with their significant performance penalty. Because of media bundling, a transaction to control a video channel contends with a transaction to control an audio channel on the same signaling path. If the channels were controlled by signals in separate tunnels, as in our protocol, this contention could not occur.

Sometimes use of our protocol will entail sending several signals simultaneously on the same signaling path, for example to open audio and video channels simultaneously. As an optimization, these signals can be bundled into one packet.

This completes the comparison from an end-to-end perspective. Now we consider the functions of application servers. An application server running an openslot, closeslot, or holdslot is acting as an endpoint, so it is really the flowlink case that concerns us. Typical behavior of flowlinks with our protocol is illustrated by Figure 13.

Using SIP, if a box in the middle of a signaling path wishes to function as a new flowlink and create media flow between its slots, it must first send to one end of the path a signal soliciting a fresh offer. This takes the form of an *invite* with no offer in it. The endpoint responds with *success* containing an offer (instead of an answer, which is what a *success* signal usually contains). When the other endpoint receives this signal, it responds with an *ack* signal containing an answer (instead of nothing, which is what an *ack* usually contains) [13].

This variation causes yet another increase in programming complexity. It has greater latency than our protocol for yet another reason, which is that our unilateral descriptors can be cached and re-used by the boxes that receive them. In SIP, answers can never be re-used because they are relative, and offers are not supposed to be re-used. Hence there is additional delay while the server solicits fresh information.

Figure 14 shows a SIP solution to the same control problem as in Figure 13. Flowlinks in both servers solicit fresh offers. When each receives its solicited offer in a *success* signal, it forwards it in an *invite*, because the signaling channel on the server's other side is in a different state.

In this scenario the two *invites* are in a race, which each flowlink knows as soon as it receives an *invite* after sending an *invite*. Because both transactions fail, both servers send dummy answers on their other sides to finish off the related transactions. Then there is a random delay to allow both servers to clean up their failed transactions, and one server (here the PBX) to retry its transaction. After this delay PC retries its entire operation, this time successfully. Thus what the PBX initiates during the delay is the mirror image of what PC does after the delay, and is actually redundant.

Using the same units as Section VIII-C, the latency of this solution is

$$10n + 11c + d$$

where d is a random variable with expected value 3 seconds. With the same time estimates as before, this comes to 3560 ms. This compares unfavorably to the 128 ms latency of Figure 13 for three reasons: (1) there is extra delay to solicit a fresh offer rather than using a cached descriptor ($2n+2c$); (2) there is extra delay to fail and retry because of contention ($3n+4c+d$); (3) there is extra delay to describe each end to the other sequentially rather than in parallel ($3n + 2c$). The situation causing delay (2) is relatively rare, but delays (1) and (3) occur whenever media is controlled by an application server. Thus, in the common situation, the comparison is 378 ms versus 128 ms.

Despite the difficulties presented by SIP, we are currently working on implementing the formal specification in SIP. Even if it is not used in current applications, the implementation in this paper is a precursor to implementing the specification with less-suitable protocols, and hopefully an inspiration for future designs.

X. Compositional design principles

Most Internet applications are not designed compositionally. Yet if we listen to the press releases, we have expectations that Internet services will work together synergistically, or at least interoperate without breaking each other. To meet these expectations of the public, we have to stop thinking of each application server as the center of its own universe, and start thinking of each application server as a member of an *ad hoc* team.

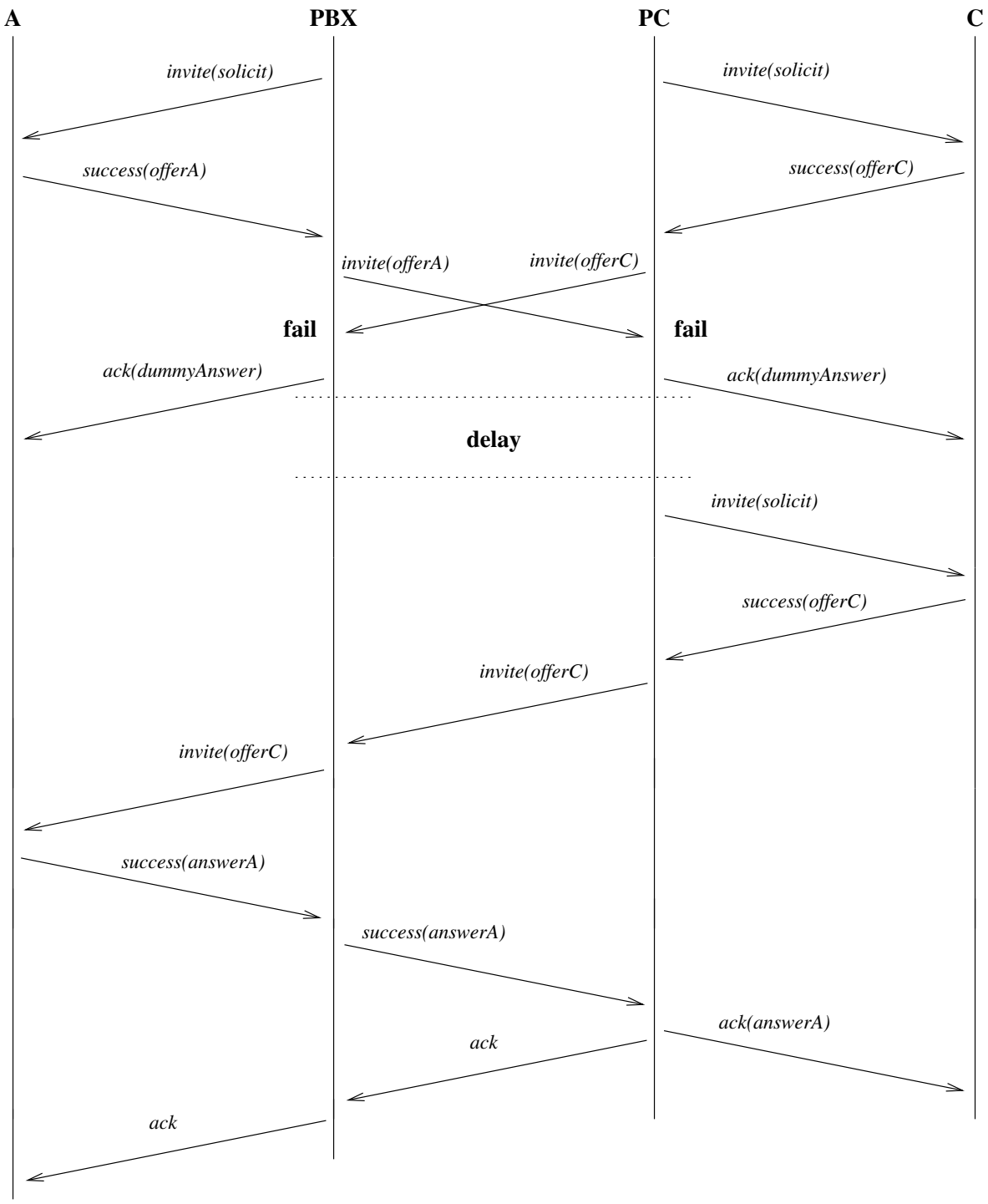

Fig. 14. Behavior of a SIP solution when the PBX and PC change state concurrently.

Our solution to compositional media control has several characteristics that might be useful in making other protocols and application programs friendlier to composition.

A. Piecewise protocol

The protocol is designed to be used in a *piecewise* fashion, so that there is no externally observable difference between a tunnel and two tunnels connected by a module acting transparently (Section III-A). It should be

obvious that this facilitates the use of application servers. Because SIP is not designed for piecewise use, it is an open problem how to build a SIP application server that is truly transparent with respect to the protocol.

B. Idempotent and unilateral protocol

Transactions are widely used in distributed programming, and they are very successful for client/server interactions. The comparison in the previous section shows, however, that real-time communication is not an asymmetrical client/server application, but something fundamentally different. In real-time communications the ends of a signaling path are symmetric, and either end (or a server in the middle) can produce a stimulus that reverberates throughout the path.

We have designed our protocol to be idempotent and unilateral rather than transactional and based on negotiation. The comparison in Section IX-B suggests that idempotent signaling and unilateral description may be superior to transactions and negotiation for control of real-time communications. They are faster and require less protocol state, both of which are important in a compositional context. There may also be other distributed applications where these opposing design principles should be compared before making a final choice between them.

C. Goal-oriented programming

State-based, declarative, goal-oriented programming primitives are much more abstract for this purpose than event-oriented primitives would be. The application's current goal for a media channel can be independent of the actual state of the media channel, for example because there is a timeout in the application, or because the environment changes the state of the channel in some unexpected way. Thus there is a wide variety of event sequences that might be needed to match the slot states in a flowlink, and these are best determined by the implementation of the programming primitives. The same observations might be true of other programming tasks.

D. Path specification

Attaching temporal-logic specifications to signaling paths, where the interior of each path is an arbitrary-length sequence of identical elements (flowlinks), was a breakthrough in our understanding of this problem.

A path is a small part of the system, measured in both space and time. In space, it is one of a large number of signaling paths that can exist, even among a small set of network nodes. In time, it is a narrow window during which no box on the path changes the goal for any of its slots. Because of this limited scope, paths are straightforward formal objects, simple enough to reason about.

At the same time, a path-based specification completely captures the requirements for compositional media control. It includes, whether explicitly or implicitly, user intentions, network topology, and feature priority.

In Section III-A we characterized compositional media control as needing to give some behavioral guarantees to each box, even though the behavior of the overall system can be affected by every box. What guarantees can we give to each box, particularly boxes in application servers? We can answer this question in terms of paths.

If a box in an application server owns a slot, then the box has the power to treat that slot as the endpoint of its signaling path (by not assigning it to a flowlink). If the box chooses to make it a path endpoint, then the other end of the signaling path may be a box in another application server, or it may be a media endpoint. If the other end of the signaling path is a media endpoint, then the box is guaranteed that the media endpoint has no media flow associated with that signaling path.

Not surprisingly, a box gets no "positive" guarantees, i.e., guarantees that it has the power to allow media flow. Media flow is always allowed by a consensus of boxes.

E. Implementation design

The last principle is the most difficult to generalize. Our flowlink implementation went through many iterations, because the earlier versions were extremely difficult to understand and debug. The key to successful code design was the combination of cached descriptors, the concept of *described* (which says that a slot has a good cached descriptor), and the concept of *up-to-date* (which says that the good cached descriptor from the other side of the flowlink has been sent to this slot). These concepts are likely to be useful for working with other idempotent, unilateral protocols.

F. Application to mobility

As an example of another application where these principles might be applied, consider the problem of providing persistent IP connections to mobile endpoints. The difficult trade-off affecting this much-studied problem concerns the number of special-purpose mobile routers that know the current location of a mobile host. If there are many such routers, the path of a packet to a mobile host can be quite direct. Unfortunately, many routers in many different subnets are required, and they must all have access to location information. If there is only one locating router for each mobile host, however, mobile routers need not be everywhere, and each location update is easy. Unfortunately, if the unique

locating router for a host is far from the sender and the current location of the host, the path of each packet can be triangular and very long [12].

In addition to this unresolved trade-off, all proposed solutions to this problem seem to favor—if not require—isolation. It can be difficult to see how they would accommodate multiple layers of mobility, or compose with a variety of other applications.

In the cases where signaling and data streams are separable, and where other applications operate directly only on the signals, it might be possible to find a better solution that is similar to compositional media control. Unique locating routers could be interspersed on signaling paths with servers for other applications. Triangular routing of data packets would be avoided by signaling/data separation, and data packets could travel between endpoints by the most direct routes.

XI. Conclusion

This paper has defined the problem of compositional control of IP media, and explained its importance in providing desired network services. We have presented a comprehensive solution in the form of an architecture-independent descriptive model, a set of high-level programming primitives, a formal specification of their compositional semantics, a signaling protocol, an implementation, and partial verification of correctness. The performance of the implementation compares favorably to the performance of the best comparable implementation in SIP. The overall solution illustrates several principles that may be useful for making other networked applications more compositional.

Although our implementation simulates IP media control, it cannot be tested with live IP media. The reason is that use of IP media in practical services requires a great deal of hardware and software infrastructure. Although we have developed such an infrastructure in our laboratory and are accustomed to using it [3], [4], it is all based on SIP rather than the protocol defined in Section VI.

Despite this limitation, the architecture-independent descriptive model, set of high-level programming primitives, and formal specification of their compositional semantics are all protocol-independent. We are currently working on implementing the specification in SIP.

Work on the protocol, implementation, and verification have not been wasted, despite the fact that they cannot be used in current applications. Without them, we would have no idea where to begin with SIP. The protocol and implementation provide compositionality in a straightforward, relatively comprehensible form that illustrates some potentially useful design principles. They are patterns for thinking about how the Internet application environment can be made more compositional.

Acknowledgments

Michael Jackson made important contributions to the earlier phases of our work on media control. We have benefited greatly from ongoing discussions with our colleagues Greg Bond, Hal Purdy, and Tom Smith. Laurie Dillon and the referees made many helpful comments on the presentation.

References

[1] 3GPP. Service requirements for the IP multimedia core network subsystem. 3GPP Technical Specification 23.228 Stage 2.

[2] National Emergency Number Association. NENA IP-capable Public Safety Access Point features and capabilities standard. Document 58-001, 2005.

[3] Gregory W. Bond, Eric Cheung, Healfdene H. Goguen, Karrie J. Hanson, Don Henderson, Gerald M. Karam, K. Hal Purdy, Thomas M. Smith, and Pamela Zave. Experience with component-based development of a telecommunication service. In *Proceedings of the Eighth International Symposium on Component-Based Software Engineering*, pages 298–305. Springer-Verlag LNCS 3489, May 2005.

[4] Gregory W. Bond, Eric Cheung, K. Hal Purdy, Pamela Zave, and J. Christopher Ramming. An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technology*, 4(1):83–123, February 2004.

[5] E. Jane Cameron, Nancy D. Griffeth, Yow-Jian Lin, Margaret E. Nilson, William K. Schnure, and Hugo Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications*, 31(3):64–69, March 1993.

[6] Eric Cheung, Michael Jackson, and Pamela Zave. Distributed media control for multimedia communications services. In *Proceedings of the 2002 IEEE International Conference on Communications: Symposium on Multimedia and VoIP—Services and Technologies*. IEEE Communications Society, 2002.

[7] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. IETF Network Working Group Request for Comments 2543, 1999.

[8] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[9] Michael Jackson and Pamela Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[10] JSR 309: Java media server control. Java Community Process, `http:// jcp.org/ aboutJava/ communityprocess/ edr/ jsr309`.

[11] Verena Kahmann, Jens Brandt, and Lars Wolf. Collaborative streaming in hetergeneous and dynamic scenarios. *Communications of the ACM*, 49(11):58–63, November 2006.

[12] Jayanth Mysore and Vaduvur Bharghavan. A new multicasting-based architecture for Internet host mobility. In *Proceedings of the Third Annual ACM/IEEE International conference on Mobile Computing and Networking*. ACM, 1997.

[13] J. Rosenberg, J. Peterson, H. Schulzrinne, and G. Camarillo. Best current practices for third party call control in the Session Initiation Protocol (SIP). IETF Network Working Group Request for Comments 3725, 2004.

[14] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. IETF Network Working Group Request for Comments 3261, 2002.

[15] Pamela Zave. Audio feature interactions in voice-over-IP. In *Proceedings of the First International Conference on Principles, Systems and Applicatons of IP Telecommunications*, pages 67–78. ACM SIGCOMM, 2007.

[16] Pamela Zave and Eric Cheung. Compositional control of IP media. In *Proceedings of the Second Conference on Future Networking Technologies*. ACM SIGCOMM, 2006.