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

When there is more than one application server in the signaling path between IP media endpoints, and the servers manipulate media flow, media flow must be controlled compositionally. This paper presents a protocol, signaling architecture, API, and API implementation for distributed, compositional media control. The semantics of the API is specified in linear temporal logic, and the implementation has been partially verified by model checking. The principles developed to solve this problem may be useful for making other network applications compositional.

Keywords
protocol architecture, protocol verification, software/program verification

1. INTRODUCTION TO IP MEDIA

Many IP (Internet Protocol) applications are concerned with transmission of media such as voice, music, and video. These applications include Internet telephony, teleconferencing, entertainment distribution, telemonitoring, multiplayer games, and distance learning.

In this paper a media endpoint (or just endpoint) is any source or sink of a media stream. Media endpoints include user I/O devices, content servers, and media-processing resources. Media-processing resources perform a wide range of functions including mixing, replicating, recording, playback, digital signal processing of many kinds, and higher-level functions such as speech recognition.

An application server (or just server) is any server whose primary function is application control rather than media processing. Programming application servers is really the subject of this paper, so it will contain many examples of their roles and functions.

For IP media applications based on point-to-point communication rather than multicast, there is broad consensus on the best architecture; this architecture is shown in Figure 1. For each IP media channel there is an IP signaling channel. The signaling channel is used to set up, control, and tear down the media channel. For example, one of its functions is to communicate to each media endpoint the IP address and port number of the other media endpoint.

Figure 1: Signaling and media channels are separate in IP media.

Once the media channel has been set up, and until it is modified or torn down, the two endpoints transmit media packets to each other directly, without the participation of the signaling channel. The reasons for this separation of signaling and media are:

- Signal and media packets travel different paths. Signals may need to go through various application servers. The high-bandwidth media packets, on the other hand, must travel end-to-end by the most direct Internet routes.
- Signaling and media channels use different underlying protocols, because their reliability and performance requirements are different. Signaling is low-bandwidth. 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.
- One signaling channel can be used to control multiple media channels, often carrying different media. The signaling protocol is the same for channels of any medium.

This complete separation of signaling and media channels is the primary characteristic of IP media. Although there is
also signaling/media separation in circuit-switched telecommunications, the separation is not as complete. In circuit-switched networks, signaling and media may have different transport protocols, but signals for a media channel travel the same path, through the same switches, as the media packets.

In general, to set up media flow in each direction between two endpoints, the sender in that direction must know the receiver’s IP address and port number. Both endpoints must know the codec (coder-decoder) that will be employed for the media stream in that direction. A codec is a data format for a medium. For example, G.726 is a lower-fidelity and lower-bandwidth codec for voice, while G.711 is a higher-fidelity and higher-bandwidth codec for voice. G.711 is approximately equivalent in fidelity to circuit-switched telephony.

2. THE PROBLEM OF COMPOSITIONAL MEDIA CONTROL

2.1 What is compositional media control?

Among other functions, application servers must control the media channels whose signaling channels they participate in. The most common media-control functions are opening, closing, switching, and holding (temporarily interrupting) media channels.

Such functions are very common. In telecommunication applications, for example, they serve more purposes than just the obvious ones of switching or conferencing users. Any feature that employs the voice channel for signaling (by means of announcements, prompts, touch-tone recognition, speech recognition, etc.) must switch user voice channels to and from voice-processing resources.

The problem of compositional control arises when the same signaling channel passes through multiple application servers. Each server has jurisdiction over the media channels controlled by the signaling channel, and each server may be attempting to perform media control. The challenge is to ensure that the overall behavior of the system is a correct composition of the actions of all the relevant servers. This correct composition must be achieved even though the servers may be acting independently, without knowledge of each others’ presence.

Figure 2 is a telecommunication example that illustrates what can go wrong when the actions of multiple servers are not coordinated. The snapshots show the same endpoints and servers at four different times.

Snapshot 1 arose in the following way. User A is a subscriber to a call-waiting feature. Because of this, A has a signaling channel to the CW server, which implements this feature. All signaling channels connecting A to other users radiate from the CW server. For example, originally A was talking to user B, so there is a signaling channel between CW and B.

While A was talking to B, a third user 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. In Snapshot 1 there is a voice channel between A and C. Because B is on hold, there is no voice channel between B and any other endpoint.

For simplicity of presentation, this paper assumes that all media channels are intended to be two-way.

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 voice-processing 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 endpoints only—they forward all media signals that they receive. In particular, CW forwards the do not send signal to A.

After the endpoints respond to all commands in Snapshot 2, the only voice 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 call waiting to switch back to B. CW 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 CW’s point of view, they have the abnormal effect of leaving V without voice 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 CW, it 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.

To put the problem of compositional media control in different words, different servers are implementing different features or sets of features. The problem of feature interaction is well known [4]. Controlling media is one of the things that features do, and affecting the same media channels is one of the ways that features interact. The purpose of compositional media control is to manage these feature interactions so that desirable interactions are enabled and undesirable ones are prevented. Thus, a distinction between desirable and undesirable interactions is a necessary part of defining correct overall behavior.

It is quite easy to see that, to control media in conformance to some standard of correctness, the servers must pay attention to signals from other servers. It is quite difficult to see, however, exactly how the servers should behave. Defining and justifying that behavior is the subject of this paper.

2.2 How real is the problem?

In the early days of voice-over-IP (VoIP), application servers were used mainly to route connection requests from one endpoint to another. Once an end-to-end signaling channel was established, endpoints were expected to do all their own media control. The only “composition” required was composi-
Figure 2: An example of erroneous media control in telecommunications. A, B, C, and V are endpoints. CW and PC are servers implementing call-waiting and prepaid-card features, respectively. Solid lines are signaling channels, dashed lines are media channels, and dotted lines are signals sent on signaling channels.

of the intentions of the communicating endpoints. This composition was achieved directly by the signaling protocol, as all protocols are designed to make agreements between the endpoints using them.

IP media is becoming more widely used, and there are increasingly many reasons why the early viewpoint is not sufficient. First we consider reasons why it might be necessary or desirable to have media-control features that are not implemented in endpoints:

- It might not always be possible to add arbitrary new software to endpoints. Consumer devices are diverse, and small consumer devices do not usually have updatable software.

- Features requiring media resources such as conference bridges or speech processors cannot be implemented in endpoints if the endpoints do not incorporate such resources or have access to them.

- Features that make personal data such as voice recordings reliably accessible to customers from all locations and endpoints cannot be implemented in endpoints unless the endpoints are reliably accessible at all times.

- A multimedia feature integrating the media capabilities of two or more devices is not easily implemented on one device.

- Functions such as providing security or billing for usage cannot be implemented in user endpoints because the endpoints do not reside in a trusted administrative domain.

- Service providers may wish to offer value-added communication services. They can only do this by implementing them in network servers and arranging for consumer access to the servers.

If there are media-control features in endpoints, even one application server in the signaling path may be enough to cause unanticipated interactions between media-control features.

Once we understand why servers might perform media control, it is easy to see why there might be multiple independent application servers in a signaling path. In the composition example, CW is associated with user A, PC is associated with user C, and the two have nothing to do with each other except that C decided to call A. The two servers are in different administrative domains, and neither has any way to know that the other is present in the signaling path.
Another important reason for multiple independent application servers in a signaling path is the requirements of large service providers, who operate clusters of servers. These clusters typically include servers developed independently by different vendors. Many of these servers have specialized purposes such as PBX functionality or voice mail. The IP Multimedia Subsystem (IMS) architecture [1], which is an emerging industry standard, recognizes the necessity to route a particular signaling channel through multiple servers within the same service provider’s configuration.

The final reason for studying multiple independent application servers in a signaling path concerns software engineering. The media application with the longest history is telecommunications. It has long been recognized by its practitioners that their software is continually changing because of the addition of features, and that their software is very difficult to develop because of feature interactions [4].

One successful approach to managing feature interactions is the Distributed Feature Composition (DFC) architecture [3, 7], in which features behave as independent modules in signaling paths. In other words, a feature in the DFC architecture plays the same role as an independent server, even though it is packaged inside an application server acting as a platform for many features. By making feature modules as independent as possible, the architecture allows each feature to be simple and comprehensible, and makes it easy to add or change features.

As an example of both the second and third reasons for multiple servers in a path, a typical call to or from a user of a commercial VoIP service [2] goes through two application servers owned by the service provider. One is supplied by an equipment vendor, and supports basic, PBX-like features. The other application server provides up to 15 advanced features, and is programmed using the DFC architecture. An ordinary call with only one party subscribing to the service could easily have ten DFC feature modules in its signaling path.

It is important to note that the modularity argument applies to features regardless of where they are implemented. Even if all media-control features reside in endpoints, if the features are complex enough, it may be better to program them as if they were independent servers.

3. OVERVIEW OF A SOLUTION

This paper presents a comprehensive solution to the problem of compositional control of IP media. In this section we describe the solution at the highest possible level; subsequent sections provide the details.

This paper does not discuss how the endpoints, servers, and signaling channels of a media application are configured and assembled, nor how they evolve over time. Everything is described within the context of a fixed “current” graph such
as the one found in Figure 2. Configuration and assembly are performed in varying ways by IMS, DFC, and the Session Initiation Protocol (SIP) for multimedia applications [10].

At the highest level, the solution is exemplified by Figure 3. In this figure, the CW and PC servers are programmed using an application programming interface (API) for media control.

When a server program wants media flow between two endpoints, it puts the two signaling channels that extend from the server to those endpoints under 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 control of a holdlink object, depicted by a black dot in the figure. The API also offers an openlink for opening media channels and a closelink for closing them, but these objects are not employed in the CW/PC example.

The signaling protocol and the implementation of the link objects are designed to achieve the goals of the servers in which they reside, subject to the goals of 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 an 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 link state in the servers. In this example, the semantics of the API and the coordinating rule can be characterized as follows: there is media flow between two endpoints if and only if there is an unbroken chain of signaling channels and flowlins between them.

To relate this behavior to proximity, consider the CW server. In every snapshot, A is media-connected to B if the CW server mandates it, and may be media-connected to C if the CW server allows it. Because the CW server is closest to A, it has priority over PC in controlling A. Only if CW 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.

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]. 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.

The new contributions reported in this paper include the API, a signaling protocol for media control, a formal specification of correctness in terms of the API and protocol, an implementation of the link objects, and a partial verification that the implementation satisfies the specification. These are described in subsequent sections, with related and future work at the end of the paper.

4. SIGNALING PROTOCOL

4.1 Signaling architecture

We use the word box for any independent module in a signaling path, whether it is a whole application server or a DFC-like feature module within an application server. As shown in Figure 4, a graph representing a current configuration consists of boxes and signaling channels. In Figures 1 through 4, all the boxes are peers.

In general, a signaling channel can control more than one media channel. Within each signaling channel, a different tunnel is used for each media channel controlled by the signaling channel.

In the presence of boxes whose media-control functions must be composed, the media-control protocol is used piecewise rather than end-to-end. This means that each box acts as a protocol endpoint, and each tunnel of each signaling channel supports a separate and independent instance of the protocol.

A signaling path is defined as a maximal chain of tunnels and flowlins, as shown by a dotted line in the figure. Each signaling path corresponds to an actual or potential media channel.

It is important to note that Figures 1 through 4 represent a particular view of IP media systems in which media sources, sinks, and processors are at the periphery of the system, while application servers are in the center. This view is convenient for understanding IP media control because it emphasizes the signaling/media separation, and isolates the media-control actions of servers.

This view does not restrict actual system architecture; nor does it preclude other views. For one example, any subset of the boxes in any of these diagrams can be implemented in the same physical component. For another example, a box that is an endpoint of two media streams can actually be mixing or transcoding those streams, so that a user view of the system would place that media-processing box in the middle, and user devices at the periphery.

4.2 Protocol definition

The media-control protocol is separate from the protocol for setting up signaling channels. Setting up signaling channels is well-understood and is not considered here; for example, signaling channels are often set up using TCP.

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.

For reference as the protocol description proceeds, Figure 5 shows a finite-state machine specification of the protocol, and Figure 6 is a scenario in which the protocol is used to open, modify, and close a media channel.

Each open signal carries the type of media channel 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.
For every medium there is a common-denominator codec that all endpoints of that medium can handle. This common codec is an entry in the codec list of every descriptor for the medium.

Each oack signal also carries a descriptor, describing the channel acceptor as a receiver of media. These descriptors are shown in Figure 6 but not in Figure 5.

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 and port number of the sender, and 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.

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 6, 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. An endpoint should be ready to receive media as soon as it has received a selector. 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 6, 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 6, descriptor3 and selector3 illustrate this interaction.

There is often a need to interrupt the media flow, temporarily, of an established media channel. Both interruption and resumption of media flow are signaled with descriptors and selectors. There are many possible ways to encode this information. We assume that there is a distinguished descriptor noMedia that means “do not send to me” and a distinguished selector noMedia that means “I am not sending to you.” The only legal response to a descriptor noMedia is a selector noMedia.

4.3 Use and properties of the protocol

Using this protocol, any number of channels of any media can be set up along a signaling path. They can be opened and closed independently, from either end. There are no unnecessary constraints to get in the way, such as, “media channels controlled by a signaling channel must be opened by the same endpoint that initiated setup of the signaling channel.”

A describe signal makes it possible for a true 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 de-

---

Figure 5: Specification of the protocol. ? 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 distinct transition labels with the same source and sink states.

Figure 6: Use of the protocol.
scribe 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 descriptor 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 in 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 decision 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.

In addition to being designed to facilitate composition, our protocol is also designed to yield an optimal choice of codec. In each direction of a media channel, the sender is sending the best codec it is currently willing to send, from the viewpoint of the receiver. The protocol does not force the two directions of a media channel to use the same codec; such a restriction would have no intrinsic value, and would sometimes make the code choice suboptimal.

A selector identifies a single codec because many media endpoints must allocate resources to whatever codec they are receiving. When they receive a new selector with a new codec, this triggers them to reconfigure themselves. Media sources may wish to send 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.

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.

5. APPLICATION PROGRAMMING INTERFACE (API)

5.1 Slots

A box is an endpoint of some set of signaling channels. Within each signaling channel, the box is a protocol endpoint of some set of tunnels. The programmer must have some way to identify these tunnel endpoints so that they can be manipulated separately.

The software abstraction that serves this purpose in the API is the slot. A slot is a software object that corresponds to an actual or potential tunnel endpoint, and is associated with the role that the actual or potential tunnel endpoint plays within the box.

In this paper we do not go into the details of how Slot objects and tunnel endpoints are associated, as that is somewhat application-dependent. The important point is that tunnel endpoints are dynamic and slots are static. A slot can be vacant, meaning that no tunnel endpoint is filling it. Or it can be filled by a tunnel endpoint. During its lifetime a slot can be repeatedly vacant and filled again, as the corresponding media channel is repeatedly lost and then replaced or restored.

The Java signature of a Slot object is:

```java
public class Slot {
    private SigChan sigChan;
    private int state, tunnel;
    private Descriptor descriptor;
    ...
}
```

where the sigChan and tunnel identify the tunnel endpoint that is currently occupying the slot, if the slot is filled.

The five Slot states are vacant, open, opening, closing, and closing, which are the same as the protocol states in Figure 5. The state of a slot is its state as a tunnel endpoint.

The descriptor of a slot is the most recent descriptor received from the other endpoint in an open, oack, or describe signal.

5.2 Programming with links

A Link is an object that controls one or two slots. As mentioned in Section 3, there are four link types, each with a different goal. These goals are discussed in detail in Section 5.3.

Every slot must be under the control of a Link object at all times. The Link object sees every signal received by the box through its slots, and mandates the sending of every signal sent by the box through its slots.

Although a slot must be under control of a link at all times, it does not live its entire life within one link. Rather, the program moves slots from link to link to achieve its current goals. A move may take a slot from one type of link to another, or it may take it from a flowlink with one slot partner to a flowlink with a different slot partner.

When a Link object is created, it receives its slot or slots as arguments. Because slots are moved from link to link, a slot can enter a link in any state. Thus the first action of a link is to query each slot to find out what state it is in. Then, having completed this initialization, the Link object proceeds to control its slot or slots until its slots are moved elsewhere and this Link object becomes garbage.

Slots are a powerful programming abstraction because they represent the roles of virtual segments of media channels within boxes, regardless of whether the media channels exist at the moment or not. This is important because whether a media channel exists at a particular moment depends primarily on the media endpoints, and a box program has limited control over it.

In the same way, links are a powerful programming abstraction because they represent goals for slots regardless of the states of the slots. This is important because many state changes within media servers are caused by timeouts,
user commands, or other stimuli that are independent of slot states. With links, a program can simply react to such stimuli with a re-assignment of slots to links. The link implementation automatically takes care of the actual slot states, and of managing them so that the link goal is achieved.

Needless to say, some state changes within application servers do depend on slot states. The API provides a way for an application program to detect slot states, so that state changes can also act as stimuli.

5.3 Link goals

Figure 7 has finite-state machines specifying the goals of the four link types. The state of a Link object, as shown in Figure 7, is derived from the state of its slot or slots. The state label dead means closing or vacant, and the state label live means opening, opened, or flowing. The figure shows only received signals that change the states of the machines, i.e., it does not show signals sent or self-transitions. Dashed arrows show the work that each link does to reach its goal.

Beginning with the simplest type, the goal of a CloseLink is to make its slot vacant. When a filled slot enters the control of the link, the link immediately forces the slot to a closing state by sending a close signal. Once the slot has received a closeack signal, the link has achieved its goal. If afterwards the slot receives a fresh open signal, the link again forces it to a closing state.

The goal of an OpenLink is to get its slot to the flowing state. When a vacant slot enters the control of an OpenLink, the link immediately forces the slot to an opening state by sending an open signal. If the open signal is rejected or the channel is subsequently closed, the OpenLink will try again.

Usually media channels are opened by media endpoints, but occasionally they must be opened by boxes, creating the need for OpenLinks. Consider, for example, a box program for a Click-to-Dial feature. When a user of a Web site clicks a Click-to-Dial number, the feature first creates a signaling channel to the telephone of the clicker, and opens a voice channel to it. This is necessary because the standard practice in telephony is to open a voice channel to a telephone before starting its ringing. If and when the user answers, the box creates a signaling channel to the clicked number, and does its best to create media flow between the two telephones. This Java code might be part of the box program:

```java
// initialization
Slot slot1, slot2; Link open, flow;
// on activation
slot1 = new Slot(toClicker);
open = new OpenLink(voice,slot1);
```

The API allows the box program to sense the state of a slot at any time. If the OpenLink fails because the open is rejected, then the program can detect this and abort. It is more likely, however, that success and failure will be signaled at the level of the signaling channel as a whole. If the entire signaling channel is torn down as a result of failure, then all its tunnels and slots will disappear.

If and when the call to the clicker is answered, the box program creates a signaling channel toward the clicked number. To complete its job, the box must open a voice channel toward the clicked number, and ensure that both endpoints have the right descriptors and selectors. It can do all of the media control simply by putting the two voice slots in a FlowLink, as follows:

```java
// when the clicker answers
slot2 = new Slot(toClicked);
flow = new FlowLink(slot1,slot2);
```

The goal of a FlowLink is to make a media connection between its two slots.

The FlowLink is by far the most complex link type, because it has two slots that can enter the link in any states, and the link must match both their states and descriptors. Most importantly, if two live states enter it, it will work toward making both slots flowing with correct descriptors and selectors. This may require sending each slot the most recent descriptor from the other slot, as described in Section 4.3.

If both slots are live and it receives a close from one of them, it will work to make both slots vacant. If two dead states enter it, it will work toward making both slots vacant. If both slots are dead and it receives an open from one of them, it will work to make both slots flowing with correct descriptors and selectors.

If one slot is live and one slot is dead, the FlowLink works to make both slots flowing. This behavior makes sense if one live, one dead is an intermediate state. For example, after receiving an open signal when both slots were dead, the FlowLink may have one slot opened (live) and one slot closing (dead). In this state, the link is blocked. As soon as it receives closeack from the dead slot, it can send an open through that slot, and both slots will be live.

The behavior also makes sense if one live, one dead is an initial state, on the grounds that the bias of media control should be toward creating media flow rather than destroying it. For more about FlowLinks, see Section 6.

The final link type is a HoldLink, which puts a slot “on hold.” If the slot is dead, the link tries to make it vacant. If the slot is live, the link tries to make it flowing, but with no actual media flow. This is done by sending it a noMedia descriptor.

These four link types are sufficient for all applications not based on multicast. They might seem to be insufficient because they cannot be used to construct multi-point media paths. Without multicast, however, multi-point media connections are always implemented within a “bridge” or “mixer” resource. All the participating media endpoints have point-to-point media channels to this resource.

5.4 Specification of correctness

To specify correctness, we assume that media endpoints are also programmed using slots and links. The goals of media endpoints are indicated by whether their slots are controlled by openlinks, closeLinks, or holdLinks. These goals are one of the factors that combine to determine correct behavior.
The other factor that determines correct behavior is the graph of tunnels through signaling channels and links within boxes. This graph represents both the intentions of individual boxes and the precedence relations on boxes.

Correctness is defined over individual signaling paths within the current graph, as in Figure 4. Each such signaling path has two ends, each of which is controlled by an openlink, closelink, or holdlink. Taking symmetry into account, there are six possible path types based on their end links. A path of a given type can have any number of tunnels and flowlinks, as these are supposed to be transparent with respect to observable behavior.

For each path type, we specify correct behavior in terms of stability or recurrence properties in temporal logic. This is necessary because a graph of signaling paths is a snapshot of a system, and can change at any time. Stability properties express the idea that if a particular path is allowed to persist long enough, the links and protocol will do their work, and eventually achieve a desired media state. Recurrence properties express the same idea, plus the additional idea that if something is perturbed while the path persists, the media 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.

To formalize the details, we visualize each path horizontally, and identify its ends as left and right. For purposes of specification each end slot is augmented with the following history variables:

• **Boolean:** vac is true if and only if the slot is vacant.
• **Boolean:** flow is true if and only if the slot is in the flowing state.
• **Descriptor:** descSent is the descriptor most recently sent by this slot.
• **Descriptor:** descRcvd is the descriptor most recently received by this slot.
• **Selector:** selRcvd is the selector most recently received by this slot.

The name of each history variable will be prefixed with L or R to indicate which end of the path it belongs to.

If one end of a path is controlled by a closelink and the other end is controlled by a closelink or holdlink, then correctness is:

\[ \square \ (L\text{vac} \land R\text{vac}) \]

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

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

\[ \square \ (\neg L\text{flow} \land \neg R\text{flow}) \]

The specification of a path with both ends controlled by holdlinks is a more complex stability property. For one reason, there are two stable goal states: both endpoint slots can be vacant or both can be flowing. For another reason, in the goal state in which both endpoints are in the flowing state, the absence of media flow must be established by looking at the selectors. Thus the stability property is:

\[ \square \ (L\text{vac} \land R\text{vac}) \land (L\text{flow} \land R\text{flow}) \land \]

\[ L\text{selRcvd} = \text{noMedia} \land R\text{selRcvd} = \text{noMedia} \]

The specification for a path with one openlink and one holdlink is exactly the same as the specification for two holdlinks except that a stable state in which both end slots
are \textit{vacant} is not acceptable:

\[ L_{\text{flow}} \land R_{\text{flow}} \land \]
\[ L_{\text{selRcvd}} = \text{noMedia} \land \]
\[ R_{\text{selRcvd}} = \text{noMedia} \]

The specification for a path with both ends controlled by openlinks requires that both ends reach a \textit{flowing} state, that each has received the most recent descriptor sent by the other, and that descriptors and selectors match locally. We abuse notation slightly by assuming that a selector derived from a descriptor is equal to it.

Unlike the previous specifications, this is a recurrence property, saying that the signaling path will always eventually return to its goal state. This is necessary because even when the path is in its goal state, it can be knocked out of it if an endpoint decides to send a new descriptor. The path returns to its goal state only after the descriptor and response to it propagate through the path.

\[ \square \diamond L_{\text{descRcvd}} \land R_{\text{descRcvd}} \land \]
\[ L_{\text{descSent}} = R_{\text{descSent}} \land \]
\[ R_{\text{descSent}} = L_{\text{descSent}} \land L_{\text{selRcvd}} = R_{\text{selRcvd}} = R_{\text{descSent}} \]

6. IMPLEMENTATION

Implementation of compositional media control requires Java code resident in each application server. Box objects contain the high-level code that calls on Link and Slot objects when necessary. Figure 8 shows the hierarchical structure of method invocations among Box, Link, and Slot objects.

![Figure 8: The hierarchy of method invocations among Java objects.](image)

There is also a Maps object that maintains the dynamic association among signaling channels, tunnels, slots, and links. When a box receives a signal, the box uses these associations to find the link to which it should show the signal. The code fragments in Section 5.3 do not include invocations of Maps methods.

Both boxes and links use \texttt{slotState} to get the current state of a slot; links also use \texttt{slotDesc} to get a current descriptor. When the box receives a signal, it passes the signal to the controlling link with \texttt{linkReceive}. In processing the signal, the link passes the signal to the slot with \texttt{slotReceive}, so that the slot can keep its local variables up-to-date. A link sends a signal by passing it to the slot with \texttt{slotSend}.

The \texttt{OpenLink}, \texttt{CloseLink}, and \texttt{HoldLink} 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 5. Most of the complexity of the implementation lies in the \texttt{FlowLink} code, which we now describe briefly.

A \texttt{FlowLink} object caches the current state and most recently received descriptor of both its slots. Its design is built around two concepts:

- A slot is \textit{described} if the link has received a current descriptor for it. Slots in the \textit{opened} and \textit{flowing} states are \textit{described}, while slots in other states are not.

- Each slot has a Boolean variable \textit{up-to-date} (\texttt{utd}) that is \textit{true} if and only if the other slot is described and this slot has been sent its most recent descriptor.

In any state, the link is working toward its goal state, which is relatively straightforward. More subtly, it is always working to maintain the \texttt{utd} variables and to make them \textit{true}.

The intuitive argument that this achieves compositional media control is quite simple. Consider a signaling path in which the right endpoint has stabilized in a \textit{flowing} state. In the rightmost flowlink of the path, the right slot will be \textit{described}. If the left slot of the flowlink is \textit{up-to-date}, then it will have sent out, to its left, the current descriptor of the right endpoint. If all flowlinks in the path have made their left slots \textit{up-to-date}, the argument continues inductively through all the flowlinks. It concludes that the leftmost flowlink has sent out, to its left, the current descriptor of the right endpoint. It will be received by the left endpoint, which now has the correct descriptor for end-to-end media flow.

These concepts are good for implementation because they cover a bewildering variety of cases in a comprehensible way. For example, consider a flowlink state in which \texttt{slot1} is \textit{flowing} and \texttt{slot2} is \textit{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 flowlinked elsewhere.

From the perspective of this flowlink, either \texttt{slot2} was originally dead and the flowlink made it live by sending an \texttt{open}, or \texttt{slot2} was \textit{opening} when it entered the flowlink. In the first case, when an \texttt{oack} is received from \texttt{slot2}, that slot needs no more immediate attention. In the second case, when an \texttt{oack} is received, the flowlink must immediately send \texttt{describe} with the descriptor of \texttt{slot1}.

Actually, the first case has been oversimplified. Even if \texttt{slot2} was sent an \texttt{open} by the flowlink with the descriptor of \texttt{slot1}, while the flowlink was waiting for \texttt{oack} from \texttt{slot2}, it may have received a new descriptor from \texttt{slot1}. In this subcase of the first case above, the flowlink also needs to send \texttt{slot2} a \texttt{describe} with the latest descriptor of \texttt{slot1}.

Fortunately, the \texttt{utd} variables collapse all the relevant history into two Boolean values, so the flowlink can always do the correct thing just by consulting them.

Interestingly, handling of \textit{select} signals is much simpler. In all cases in which a flowlink succeeds in reaching its goal state of \textit{both flowing}, it sends a descriptor of the other slot to each slot. A selector always responds to a descriptor, and in the \textit{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.

7. VERIFICATION

The implementation maintains dynamic associations among signaling channels, tunnels, slots, and links. These associations are found in the Maps object and in local variables of other objects. We have stated and proved manually a number of straightforward invariants concerning these associations. We have also tested the code with test drivers rather than real media endpoints, for reasons discussed in the final sections of the paper.

Except for the invariants, all of the verification has been performed by modeling the Java code in Promela and checking the Promela models with the Spin model checker [6].

The scope of each model is a static signaling path. This means that dynamic associations among signaling channels, tunnels, slots, and links are not included in the Promela models.

In every Promela model of a link, there is an initial phase in which the behavior of each slot is completely nondeterministic and has nothing to do with the goal of the link. At some nondeterministically chosen point, the process switches permanently to a linked phase it which it behaves according to the link program. Because of the initial phase, model checking covers traces in which the links begin their work in every possible initial state of the slots and of the signaling channels that connect the slots.

Several modifications reduce the state space of the Promela models. The number of unsolicited describe and select signals that a slot can send is limited to one each. Descriptors and selectors are represented by one bit.

Despite a straightforward high-level design, the flowlink program is very complex in detail. Prior to verification, model-checking was essential as a debugging tool.

During verification, for each signaling path, we performed two checks. 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 vacant or flowing, and all signaling channels are empty. Second, we verified that each signaling path satisfies its specification as given in Section 5.4.

We modeled and checked 12 signaling paths: six paths with no flowlinks and every possible combination of closelinks, openlinks, and holdlinks at their ends, and six paths similar to the first six paths but with one flowlink each.

It is not feasible to model-check longer signaling paths. Each flowlink has many states, and each tunnel is modeled by two FIFO channels that need to hold at least three signals each to prevent artificial deadlocks. Thus, adding even one additional flowlink to a path causes a state explosion.

In summary, the step that is still needed for a complete proof of correctness is a proof that two adjacent flowlinks are behaviorally equivalent to a single flowlink in a signaling path. With such a proof, we could extend our results inductively to signaling paths of any length. Finding such a proof is an interesting and significant challenge.

8. RELATED WORK

In the first IP-based implementation of DFC [3], 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 [5]. The media-control module maintains an explicit graph of the media state; this graph has the same information as the graphs shown in 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 [2].

The solution reported in this paper 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.

There is no other work on compositional control of IP media that we know of. For example, SIP [10] is the dominant signaling protocol for control of IP media. It is being used in most new service development, and is an integral part of IMS [1].

The design philosophy of SIP has been clearly stated by its designers: it is designed for communication between endpoints, with minimal intervention between them. Ideally there would be nothing between endpoints except proxy servers with no media functions. The actual design of SIP is true to this philosophy. As a result, media control in SIP differs radically from media control as proposed here.

The design philosophy of SIP was typical in the early days of the Internet. The problem for industry today is that, as explained in Section 2.2, prospects and plans for IP media have moved beyond what was envisioned in those days.

In an attempt to bridge the gap, a document on best current practices for SIP [9] 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.

Although there is so little directly related work, the themes of this paper may be related in a broader sense to many other forms of networking.

For example, the goal of the Mobile IP standard [8] is to support persistent TCP connections between mobile endpoints. Consider a Mobile IP connection between an ordinary endpoint and a mobile endpoint. Packets from the ordi-
nary endpoint to the mobile endpoint pass through a server (the “home agent”) with Mobile IP functionality, while packets from the mobile endpoint to the ordinary endpoint do not pass through any Mobile IP server. This is reminiscent of signaling/media separation as in Figure 1, except that the two different paths correspond to different directions of data transmission.

Mobile IP has not been notably successful, and its goals might be achieved equally well or better by an architecture that separates signaling and data completely. Just as signaling/media separation is preferred for high-bandwidth media streams, signaling/data separation might be efficient for high-bandwidth bulk data streams. Servers in the signaling path could provide mobility and other functions.

In the same spirit of analogy, it is possible that other end-to-end protocols, in addition to SIP, may be under pressure from the need to put servers in signaling paths. This need can arise from security concerns as well as functional requirements. If so, the approach to compositionality in this paper may provide useful principles and mechanisms.

9. CONCLUSIONS AND FUTURE WORK

Compositional control of IP media has proven to be a very difficult problem. We have been working on it since 1999, and the solution in this paper is the first fully satisfactory one.

We found this solution by relentless stripping away of inessential complexity in the problem domain. Because all aspects of the problem are closely intertwined, eliminating complexity required redesigning the protocol and API as well as the implementation algorithm.

Our current challenge is to embed this solution into our research platform for creating IP multimedia services. The goal of this platform is to implement the Distributed Feature Composition (DFC) architecture on top of SIP-based application servers and media resources. The reason for this hybrid goal is that DFC has proven to be a successful technology for modular feature development and management of feature interactions, while SIP is the dominant industry standard. The challenge is not a small one, because of the many differences between SIP and our preferred protocol.

On the brighter side, the principles we have used to solve this problem may be applicable to a variety of related networking problems. If so, evidence of their usefulness may create leverage for improving existing protocols.

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.

10. REFERENCES


