Audio Feature Interactions in Voice-over-IP

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

ABSTRACT

In telecommunications, audio signaling is the use of the audio channel for signaling and user-interface purposes. When features use audio signaling, and are assembled in a pipes-and-filters configuration, there is a potential for undesirable feature interactions. This paper analyzes the potential feature interactions. It proposes a method for eliminating some of them, as well as directions for future work on the remaining interactions. The method can be implemented in SIP, using compositional patterns of signaling that will work correctly regardless of how many features are active.

Keywords

telecommunications, protocol verification, SIP

1. INTRODUCTION

Since the 1960s, when telecommunication systems became software-controlled, there has been a trend toward adding functionality in increments called *features*. Since the 1980s, when telecommunication software became overwhelmingly complex, there has been a trend toward encapsulating new features in software modules.

The inevitable by-product of *feature modularity* is *feature interaction*, because telecommunication features cannot be completely independent of one another. Feature interactions must be managed by a process that distinguishes between desirable and undesirable interactions, enables the good interactions, and prevents the bad ones.

Although the purpose of voice telecommunications is to enable conversation among people, throughout most of its history the audio channel to a user has also been used for signaling and user-interface purposes. Here "audio signaling" refers to progress tones, announcements, voice prompts, touch-tone detection, and voice recognition for control purposes.

Many technologists have predicted that VoIP would be the end of audio signaling, on the grounds that VoIP devices have much better signaling capabilities than old-fashioned telephones. Like audio and unlike old-fashioned telephones, the user interface of a personal computer is infinitely extensible.

For all features and services implemented outside endpoint devices, however, audio signaling is alive and well. The overwhelming reason is that audio signaling requires no assumptions about endpoint devices. Most users are still talking on ordinary telephones, and are still connected (even to VoIP users) through the Public Switched Telephone Network.

Even in a future world in which all telecommunication is IP-based and all endpoint devices are full-functioned computers, audio user interfaces will still be valued. They are user-friendly, highly portable, do not require the use of eyes, and are often hands-free. The audio user interfaces of features in endpoint devices and in the network will continue to interact, just as they do today.

This paper analyzes the large and important class of feature interactions in telecommunications due to audio signaling. In building the advanced features for a consumer broadband VoIP service [1], this was one of the major classes of interaction that we had to manage. Most of the example features in this paper were present in some version of this VoIP service. The paper also presents the foundations of a method for managing the feature interactions.

The paper assumes that feature modules are composed in a pipes-and-filters configuration, with feature modules being the filters, and instances of the call protocol being the pipes. Distributed Feature Composition (DFC) was the first pipes-and-filters architecture for VoIP systems [2, 7]. The pipes-and-filters idea has since been adopted for the SIP Servlets architecture [8, 9]. It appears that no circuit-switched telecommunication systems have a pipes-and-filters architecture, so the method presented here applies only to VoIP.

The paper begins with a brief summary of the pipes-and-filters architecture for telecommunication services (Section 2). Section 3 analyzes the possible audio feature interactions in this architecture, while Section 4 presents the rudiments of a method for managing them.

The analysis and method both use an abstract protocol that is related to both the DFC protocol and SIP [6, 13], but is not identical to either of them. Section 5 outlines how this abstract protocol might be implemented in SIP. Related and future work is discussed in Section 6.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

2. THE PIPES-AND-FILTERS ARCHITECTURE

2.1 Protocol

At the highest level of abstraction, the call protocol begins when the caller endpoint sends the callee a *request* for connection. Eventually the callee endpoint (for example, any kind of telephone) should send a response, which is *success* or *failure*. If the response is failure, then the call is over; the failure signal may include a modifier indicating the reason for the failure. If the response is success, then there is an audio connection between caller and callee. Control of the audio channel is introduced in Section 3.

The realization of this protocol in a pipes-and-filters architecture is shown in Figure 1. A request is typically routed to the next feature module that applies to it (see Section 2.2). The feature module is an object, and unless otherwise noted, it is a new instance of its class. If there are no more features that apply to the request, then the request is routed to an endpoint device.

As a request travels from one module to the next, it creates a two-way signaling channel between the sending and receiving modules. An observer of this signaling channel would see a complete and self-contained instance of the call protocol. These individual calls are linked through the feature modules to generate the end-to-end call behavior.

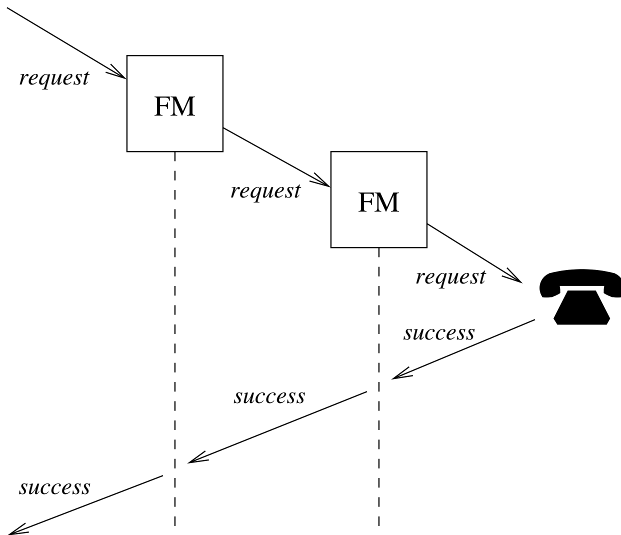

Figure 1: A chain of feature modules linked by calls. In this diagram, all features are behaving transparently.

When none of its functions is triggered, a feature module behaves transparently. Transparent signaling behavior of a module with two calls, one incoming and one outgoing, consists simply of sending each received signal out on the other call. Because all the feature modules in the figure are behaving transparently, the success response from the endpoint device is propagated backward through the chain of calls.

When a feature module is not behaving transparently, it can modify, delay, or absorb any signal that it receives. It can also generate new signals. The only constraint is that the signals exchanged on each signaling channel between two adjacent modules must form a legal and complete instance of the call protocol.

The requesting end of a call instance can send an *end* signal to end the call at any time. The accepting end can send an *end* any time after sending *success*. Additional handshaking signals are needed to set up the two-way signaling channel and to acknowledge that it has been torn down at the end of a call, but these signals are not relevant to audio feature interaction.

2.2 Routing

In a pipes-and-filters architecture, a typical caller request is handled by a chain of feature modules and calls, as shown in Figure 1. This chain is assembled dynamically by a *routing algorithm* that routes each request in the chain to the appropriate module.

Routing in a pipes-and-filters architecture is outside the scope of this paper. However, interested readers can find informal presentations in [2] or [7], and complete formal definitions in [15] or [9].

Because feature modules can be the sites of forks and joins, a configuration of feature modules can be a graph as well as a linear chain. Some examples later in the paper include forks and joins.

2.3 Feature phases and functions

The benefit of a pipes-and-filters architecture is that feature modules can be specified, implemented, and deployed as independent entities, greatly reducing software complexity.

A feature module behaves transparently except when it receives a signal or time-out that *triggers* the feature. Once triggered, the feature module can query databases, use audio-processing, or manipulate signaling. It may return to quiescence (transparency). If it does, it may be triggered again. Thus, over its lifetime, a feature module alternates between *active* and *inactive* (transparent) phases. Each active phase is triggered by a received signal or time-out.

It is often useful to refer to directions within a chain. The *downstream* direction is toward the callee; requests travel downstream. The *upstream* direction is toward the caller; success and failure signals travel upstream.

The decomposition of overall system functions into separate feature modules is arbitrary—features can be "big," with many functions, or "small," each doing a single task. This is valuable because many historical and economic forces constrain designs in the real world. The examples in this paper tend toward "small" features, because this style illustrates that a high degree of modularity is possible.

3. ANALYSIS OF AUDIO FEATURE INTERACTIONS

For descriptive and analytic purposes, first assume that a single two-way audio channel accompanies each signaling channel. Just as signals in a chain of calls go through feature modules, assume that the audio channel also passes through feature modules, which can manipulate it.

Whenever a feature module is described as performing an audio function such as generating a progress tone, playing an announcement, or detecting touch tones, assume that the function is being performed within the module itself, as opposed to being delegated to some media server. Subsequent sections will show how to turn this abstraction into a realistic implementation.

3.1 Progress tones

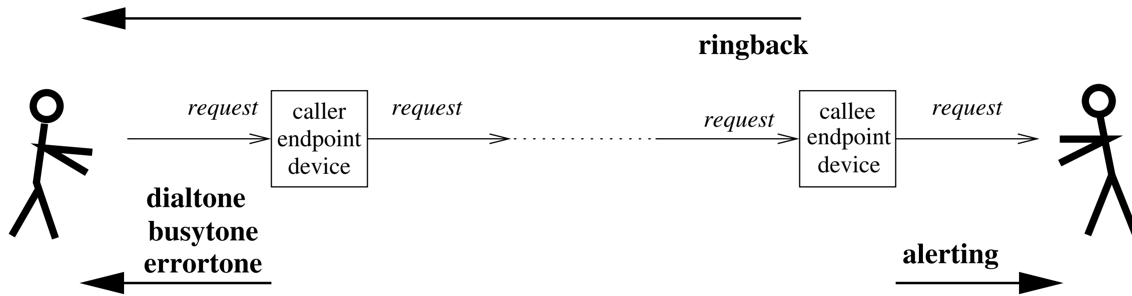

Figure 2: Endpoint devices are feature modules that may generate progress tones.

Much audio signaling is highly customized for the feature using it. For example, a Call Forwarding on Request (CFR) feature redirects a request to one of a set of addresses. Often it has a user interface in which the caller uses touch tones to answer the question, "How may we direct your call?" This user interface is typically implemented by a VoiceXML script specifying the menu of prompts.

Progress tones such as ringback and busytone are different from interactive voice-response interfaces because they are simple and standardized. We might reasonably expect them to be implemented by endpoint devices, so that features never have to do anything to generate progress tones except send signals to the endpoints.

Wherever they are generated, progress tones are an important use of the audio channel, so they must be included in our analysis. To begin, we regard endpoint devices as feature modules with hardware that may generate progress tones (Figure 2). The figure makes no distinction between requests that are user actions (such as picking up a handset) and requests that travel through the network. It makes no distinction between alerting through the handset speaker and alerting through the air, by a bell or other mechanism.

If an endpoint feature module receives a request from a user, it is currently playing the caller role. It may play dialtone upstream (toward the user), then send the request downstream (to the network). If the outcome of the request is a failure signal from downstream, it may play busytone or errortone upstream, until the user does something to stop the tone.

If an endpoint feature module receives a request from the network, it is currently playing the callee role. It may alert the user downstream and play ringback upstream, until the request is aborted or a user accepts the call. Ringback should be regarded as being played by the callee end for two reasons. For one reason, ringback is, conceptually, the echo of alerting. For another reason, Section 4.1 will show that this is the best place to *control* ringback. As stated at the beginning of Section 3, the location of an audio function in this analysis is not necessarily its location in the final implementation.

3.2 Audio contention

Features in a pipes-and-filters configuration are programmed independently and run concurrently with respect to each other. If they use audio signaling, then they have the potential to attempt to use the same audio channel simultaneously. This *audio contention* is always a bad feature interaction, because at least one of the features will not work as expected. Because it is a bad feature interaction, the only way to manage it successfully is to prevent it.

For one example of potential audio contention, many features use audio signaling to communicate with the caller before sending a request to an endpoint. CFR as described above is one such feature. Another such feature is Do Not Disturb (DND), which may play an announcement to the caller that the callee does not wish to be disturbed, and prompt to ask the caller if the call is urgent. If the call is urgent, DND will send the request to an endpoint despite the callee's default preference. When both of these features apply to a caller's request, it is important to ensure that they do not attempt to use the audio channel to the caller simultaneously.

For another example of potential audio contention, forking in SIP can reach multiple features simultaneously, some of which expect to use the audio channel to the caller. At most one of them can be connected to the caller, which means that some of them may not work as expected [4].

3.3 Reversed and sequential progress tones

The second feature interaction in the audio category is a potential bad interaction between the endpoint device feature modules and some other features, which produce *reversed and sequential progress tones.* Consider the Click-to-Dial (C2D) feature, as illustrated in Figure 3. After being triggered by a signal from a Web service, it first places a call to the clicker's phone. Once the clicker has answered, C2D places a call to the clicked address.

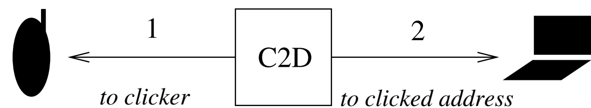

Figure 3: Click-to-Dial makes two outgoing calls in the numbered order.

Because of C2D, the user on the clicker end should hear progress tones such as ringback and busytone to indicate the status of the second call, *even though the clicker's phone was reached in the role of a callee.* These are *reversed* tones, because they reverse the expectation that ringback and busytone are heard only by callers.

Similarly, a person who is already talking to another person can try to add a third person by activating a Three-Way Calling (3WC) feature. This person (the activator) should

hear progress tones to indicate the status of the third call. If the activator was originally the caller, then these are *sequential* tones, because the caller has already made one successful call, and heard the tones for it as it was being set up. If the activator was originally the callee, then these are reversed tones.

Reversed and sequential tones are desirable from the standpoint of users, because they provide a complete and familiar user interface for many features in many different situations. Reversed and sequential tones produce a bad interaction with endpoint device feature modules, however, if the endpoint device feature modules do not support them.

For example, although the expectation of the SIP architecture is that endpoint devices will perform all tone generation, SIP signaling constraints do not allow devices to generate reversed or sequential tones. This is because the standardized SIP signals that would cause an endpoint to generate progress tones (180 for ringback, failure responses for busytone and errortone) are not allowed in reversed or sequential situations.[1]

4. A METHOD FOR PREVENTING AUDIO FEATURE INTERACTIONS

4.1 Audio contention in single chains

First, a method for eliminating audio contention in a restricted context will be presented. Then the context of applicability will be widened.

Consider the handling of a single request. In the pipes-and-filters architecture, the request stimulates assembly of a dynamic chain of feature modules. In this section we consider these chains, with two restrictions: (1) No module can have more than one continuation request at a time. (2) All continuation requests sent by a module are sent for the purpose of helping its incoming request succeed. This means that once the module has sent a success signal upstream, it cannot send any additional requests.

If all the feature modules in the chain obey a simple convention, then we can be sure that there is no audio contention. The essence of the convention is that a request signal traveling downstream, followed by a success signal traveling upstream, can be regarded as a token that is possessed by no more than one feature module at once. If a feature module performs audio signaling only when it has the token, then there can be no contention.

Figure 4 is a more precise and detailed representation of this convention. It is a nondeterministic meta-program that can be refined to a program for any feature module that follows the convention.

The letters i and o name the ports of the incoming and current outgoing call, respectively. ! and ? denote sending and receiving a signal, respectively. Each state is labeled with a letter and the calls that exist in the state, which may be *[i]* or *[i,o]*. The black dot is the initial state, while bars are final states. Commas separate independent transition labels with the same source and sink states. The label *o?end / i!end* means that the module propagates an end signal

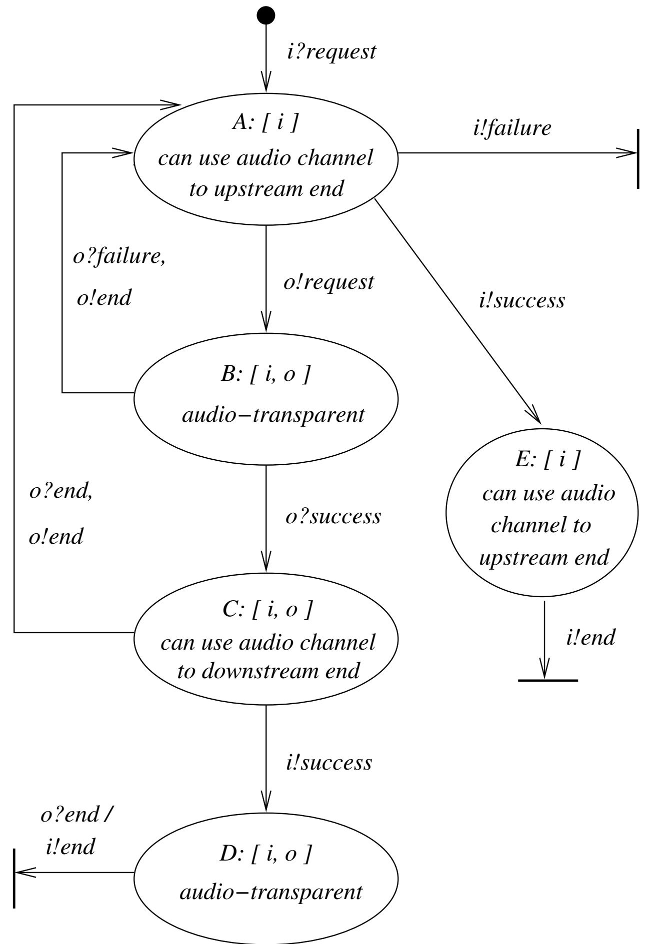

Figure 4: A meta-program for modules that receive a single request and have no more than one continuation request at a time. *i?end* **is possible in any post-initial state.**

received from downstream.

In addition to the transitions shown explicitly in the figure, in any state except the initial state, the program can receive an *end* signal from the incoming call. In response, the program must end the outgoing call (if any) and terminate.

If a feature program is in state A or E of the meta-program, its activity may include using the audio channel through i to communicate with whoever is connected to the upstream end of the audio channel. If a feature program is in state C of the meta-program, its activity may include using the audio channel through o to communicate with whoever is connected to the downstream end of the audio channel. If a feature program is in state B or D, it must be transparent with respect to audio, which means that the audio channels associated with i and o are connected to each other.

The meta-program is nondeterministic because it makes room for many possible behaviors of programs that refine it. For example, in state A, a program can choose to leave the state by sending request, success, or failure signals.

By reasoning about the meta-program and the signaling properties of chains, it is easy to prove that if a module is

[1]As an aside, the scope of this problem extends beyond audio signaling. Even if a SIP endpoint indicates progress to its user by graphical or other means, it cannot do so in reversed or sequential situations, because SIP does not allow the necessary signals.

in state A, then all modules upstream of it are in state B, and are therefore audio-transparent. This is true because all the modules upstream have sent a downstream request, and have not yet received a success signal from downstream.

Furthermore, if a module is in state C, then all modules downstream of it are in state D or E. These modules are either audio-transparent, or represent the audio endpoint of the chain. This is true because these modules have already sent a success signal upstream.

These two theorems guarantee the absence of contention when a feature module uses the audio channel in states A or C. The use of the audio channel in state E will be discussed in Section 4.3.

4.2 Refinements of the meta-program

The examples in this section illustrate the many ways in which the meta-program can be refined.

The Do Not Disturb (DND) feature is enabled by subscriber data. If enabled, it behaves as described in Section 3.2. The announcements and prompts mentioned there all occur when the DND program is in state A of the meta-program. If the call is urgent, DND continues it by sending *o!request*, entering state B, and going transparent.

Transparent behavior is a refinement of the meta-program. From state B, if a transparent program receives a success signal from downstream, it propagates the signal upstream, and enters state D. From state B, if a transparent program receives a failure signal from downstream, it propagates failure upstream, and terminates.

If the incoming call is not urgent, DND (in state A) sends *i!failure* and terminates. Because of the highly modular decomposition illustrated by the features in this paper, the failure is handled by a different feature such as Record Voice Mail.

Call Forwarding on Request (CFR), introduced in Section 3.1, also does all its work in state A, then sends the forwarding address in *o!request*. CFR could be combined in the same feature module with Call Forwarding on Failure (CFF), in which case it might have an active phase on a second visit to A after *o?failure*. Usually, this phase would use a database query rather than a caller choice to determine the forwarding address. Again, the forwarding address would be sent in the request causing a transition from state A to state B.

As a refinement of the meta-program, a Collect feature module would interact with the caller in state A, possibly to record the caller's name. In state C the module would interact with the callee, possibly playing the recorded name, and certainly asking for permission to bill the call to the callee. If the callee accepts, the module sends *i!success* and goes transparent. If the callee refuses, the module sends *o!end* and returns to state A. In state A it can inform the caller that the callee has refused, and finally send *i!failure*.

A No-Answer Time-Out (NATO) feature module generates a time-out so that a request is guaranteed to yield an outcome after a bounded amount of time. It does not use the audio channel. As a refinement of the meta-program, it does nothing in state A except to set the timer and continue the request. If there is a time-out in state B, the module sends *o!end* and *i!failure*, then terminates. If the module receives an outcome before a time-out, it becomes transparent.

As a refinement of the meta-program, an endpoint device feature module playing the caller role receives a user request *i?request* through the hardware. It can generate dialtone on its first visit to state A, and busytone or errortone on a subsequent visit to A if the request fails. If it is generating busytone or errortone, receiving *i?end* from the user through the hardware will turn off the tone.

Record Voice Mail (RVM) is a familiar feature that is triggered by the failure of its continuation request, and that provides a good substitute for reaching its subscriber by offering to record a voice message. Thus RVM turns failure into success. On receiving *o?failure* in state B, an RVM program passes instantly through A, sends *i!success*, and goes to state E.

Note that a feature module in state B can monitor the audio channel, provided that it does not interfere with end-to-end audio communication. For example, a Sequential Find Me (SFM) feature might sequence through outgoing requests to a list of addresses, attempting to reach the intended callee. It might allow the caller to abort any particular attempt because it is alerting too long or is otherwise unpromising. The feature module would do this, in state B, by monitoring the audio channel from the caller for a touch-tone signal to abort. If the signal arrives, then the feature module sends *o!end* and returns to state A to make another attempt. In the same way, a feature module can monitor the audio channel in state D.

As a refinement of the meta-program, an endpoint-device feature module playing the callee role is a slight exception. It generates both alerting and ringback in state B. To make it fit the meta-program, we must imagine that the alerting bell is just downstream of the feature module, and that ringback is heard upstream because the feature module is audio-transparent in state B, and ringback is the same sound as alerting. It is safe to bend the rules in this way because we know that there are no feature modules between the endpoint device feature module and the imaginary bell.

4.3 Audio contention in multiple chains

Parallel Ringing (PR) is one of the most interesting features we have implemented. When a PR feature module receives a request for the person who subscribes to PR, it sends simultaneous outgoing requests to several device addresses where the person might be reached. If one succeeds, then PR connects that device to the caller and ends the other requests.

To manage audio contention with the method introduced above, it is necessary to regard PR as playing two roles: as the end of one chain and as the beginning of some number of others (Figure 5). As the end of the chain initiated by the caller, PR remains in state A until it sends success or failure upstream. In state A, it generates ringback. From this viewpoint, its outgoing requests do not exist.

As the beginning of some number of independent chains, PR spends most of its time in state B for each one of them. However, in this state i is vestigial, and any downstream module that attempts to use its state A to communicate with the caller will get no response. This unfortunate feature interaction is well known; it cannot be fixed, because there is no graceful way to share the caller's audio channel among the parallel requests.[2]

[2]Forking in SIP has the same purpose as PR. Although this problem has been discussed in the forking context [4], none of the actions proposed there will help if a downstream module needs communication with the caller to make its branch

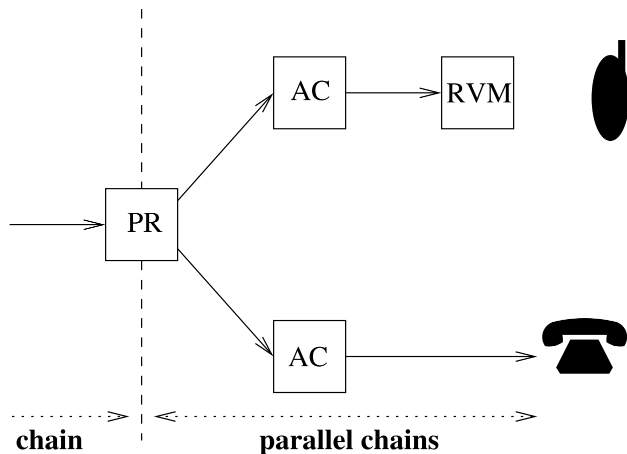

Figure 5: Multiple chains linked by Parallel Ringing.

Fortunately, it is possible for the downstream requests to make productive use of the audio channel. Our consumer VoIP service [1] has the target address of each device subscribe to Answer Confirm (AC), as shown in Figure 5. AC solves the following well-known problem: if the target of one of the requests is a cellphone with RVM (for example), and the cellphone is turned off, then the cellphone's RVM will answer the request immediately. This will cause PR to abort the other requests immediately, so that there will be no chance of reaching a person.

When AC receives success from downstream, it enters state C. In this state it uses an audio interface downstream to announce, "This is a call for John Doe. Please press 1 to accept the call." If it receives the correct touch-tone, it sends *i!success* and enters state D. Only a person will enter the tone, so AC distinguishes requests accepted by RVM, and causes them to fail or time out.

The third theorem about the behavior of the meta-program concerns use of the audio channel in state E. When a feature module is in state E, it is the permanent audio endpoint of the chain. The theorem states that if a feature module is in state E, then all feature modules upstream of it are in states B, C, or D. B and D are audio-transparent states, so they do not contend with the module in state E. The AC/RVM example shows that a C/E combination is not a case of audio contention, but rather a legitimate situation in which the module in state C is using the audio channel to communicate with the audio endpoint of the chain.

4.4 Evaluation of the meta-program

Conformance to the meta-program eliminates audio contention only among those features that conform to it. If there is an island of conforming components in a sea of non-conforming components, then the island will interoperate with the sea as well as usual, but there may be audio contention between features on the island and features in the sea.

Experience indicates that the meta-program is a valid abstraction of a large variety of feature modules. Inevitably, however, there will be feature modules that seem legitimate but do not conform exactly to the meta-program.

succeed.

In these cases the use of the meta-program, which is a very easy way to guarantee the absence of audio contention, must be augmented with additional reasoning about the special case. The reasoning that an endpoint device feature module on the callee end can safely use the audio channel in state B is an example of such additional reasoning. So is the section about PR, which shows that PR is safe in every way except that feature modules downstream of PR cannot use the audio channel to communicate with the caller.

Eventually it may be possible to design a more powerful meta-program that accommodates all the exceptions. But even if this is not possible, the current meta-program is a powerful tool for distinguishing between easy and difficult cases, taking care of the easy cases quickly, and showing exactly what work must be done on the difficult cases.

As a last resort, cases of audio contention can be eliminated by conferencing. Mixing audio sources allows several to be heard simultaneously. Consider, for example, a feature module that for some reason must use the audio channel in state B. If it forms a three-way conference with i, o and its audio-processing resource, then the audio channels of i and o can still be regarded as transparently connected.

4.5 Definition of added calls

Assuming that all feature modules adhere to the discipline of the meta-program, *added calls* are calls whose requests are sent or received by feature modules in state D. If a feature module receives a second incoming request in any state other than D, the request should be rejected.

Many feature modules can add calls, and they are the cause of reversed and sequential progress tones. Section 3.3 already introduced C2D and 3WC. Another well-known feature that adds calls is Call Waiting (CW). CW is different from the others because the added request is incoming rather than outgoing.

Another feature that adds calls is Sequential Credit-Card Calling (SCCC). This feature prompts the caller to enter credit-card information. Then the user can make a sequence of calls charged to the same account, without re-entering information. SCCC may seem very similar to SFM, which also makes a sequence of continuation requests, even though the requests of SFM are not considered to be "added calls." The difference is that all the continuation requests of SFM are made for the purpose of helping its incoming request to succeed, and once one continuation request has succeeded, it makes no others. SCCC, on the other hand, can make any number of successful continuation requests.

Figure 6 shows feature modules attempting to add a call. In the figure, endpoint W, through its Endpoint Device (ED) feature module, is already talking to X, and Y is already talking to Z. W subscribes to 3WC, and Z subscribes to CW. W is using 3WC to call Z through CW. There may be other feature modules, implicitly, in any of the paths.

Because of the presence of added calls, a feature module may have signaling channels to three or more endpoints. This introduces the possibilities of switching or conferencing their audio channels.

4.6 Supporting reversed and sequential progress tones

The structure provided so far makes it straightforward to produce reversed and sequential progress tones, which are necessitated by added calls. The basic idea, as exemplified

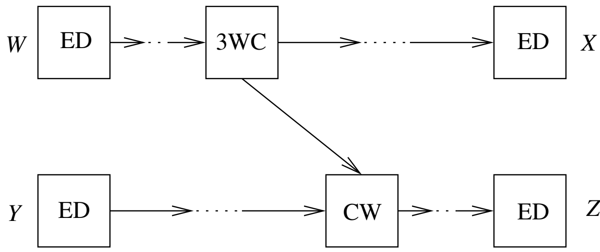

Figure 6: Adding a call from W to Z.

by Figure 6, is that the 3WC module of W acts like a caller endpoint feature module, generating any tones needed to be heard by W. It also connects the audio channel of the added call to its subscriber W when appropriate. Similarly, the CW module of Z acts like a callee endpoint feature module, generating any tones needed to be heard by Z. It also connects the audio channel of the added call to its subscriber Z when appropriate.

Tone generation can be moved from the modules where the tone is specified (3WC and CW in the example) to any point between that module and the ears of the user. All we need is a simple set of end-to-end signals indicating the beginning and ending of tones.

Ideally all tone generation would be implemented in endpoint devices, where it is most efficient. However, as noted in Section 3.3, the infrastructure does not always support reversed and sequential tones. SIP allows tone-generation signals in some circumstances and not in others.

The solution that we used in our consumer VoIP service [1] is as follows. There are two distinct sets of tone-generation signals, those standardized in SIP, and a set invented for our purposes (and sent in SIP *info* signals). Whenever possible, feature modules use the standardized SIP signals, so that tones will be generated by the endpoints. When this is not possible, feature modules send the invented signals.

Every telephone automatically subscribes to a Tone Generation (TG) feature module, which is placed in feature chains nearer to the devices than any other feature. A TG module responds to the invented tone-generating signals, playing the tones on the audio channel toward its device.

4.7 Audio contention and added calls

Added calls introduce many new opportunities for audio contention. Each adding feature module needs, during the addition phase, a clear audio path to its subscriber.[3] Three things might go wrong to interfere with the clear audio path, thus producing audio contention:

- Some feature module between the adding module and its subscriber might be in state C.
- Some feature module between the adding module and its subscriber might also be in the process of adding a call.
- Feature modules that manage multiple parties, such as 3WC and CW, often perform switching of the audio channel. The audio channel from the subscriber may have been switched away from the adding module in favor of the audio path to some other party.

[3] A possible exception to this is CW, which serves as the callee endpoint feature module. It can alert its subscriber with a very short tone or with conferencing, so that there is no contention with the ongoing use of the audio channel.

Managing audio contention for added calls seems to be quite challenging. Although it will be left to future work, there is some reason to expect that a satisfactory solution can be found.

The ray of hope is that outgoing added calls do not occur at random times, but are added because of explicit commands from a subscriber. This makes it reasonable to assume that such actions should be sequentialized, and to design enforcement into the user interface. In our commercial VoIP service [1], for example, we had to manage possible audio contention among several features that could be active mid-call. Although our techniques were *ad hoc*, they were intuitive and proved to be successful.

5. IMPLEMENTING AUDIO SIGNALING IN SIP

5.1 Implementation architecture

The meta-program is a tool for understanding how to coordinate use of audio channels. It should be viewed as a specification that can be implemented in many different ways.

In our implementation, the abstract call protocol is implemented in SIP. Most feature modules are B2BUAs, so that each signaling channel between two adjacent feature modules carries its own dialogue. For example, Figure 7 illustrates a signaling chain consisting of a UAC (caller, referred to as T), a UAS (callee, referred to as U), and two B2BUAs (feature modules F and G).

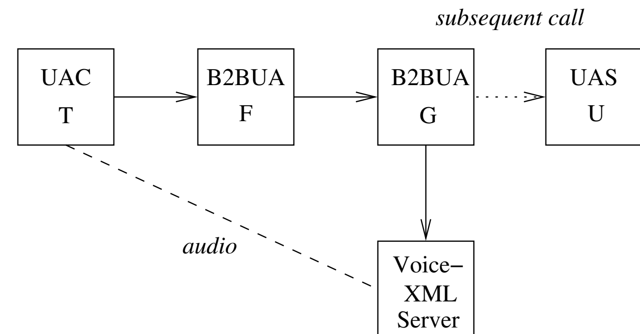

Figure 7: Architecture of a SIP implementation.

Whenever a feature module needs to perform media processing, it makes a call to a media server. In the figure, G is in state A, and is interacting with the caller through an interactive voice-response user interface. To do this, it has placed a SIP call to a VoiceXML server loaded with an appropriate script. G coordinates the signals of its two calls so that an end-to-end audio channel is set up directly between the VoiceXML server and T.

If the interaction with the caller is successful and G proceeds to state B, then G will end the call to the server and continue the incoming request to U. It will coordinate signals so that an end-to-end audio channel can be set up between U and T. Subsequent sections describe how this coordination is accomplished.

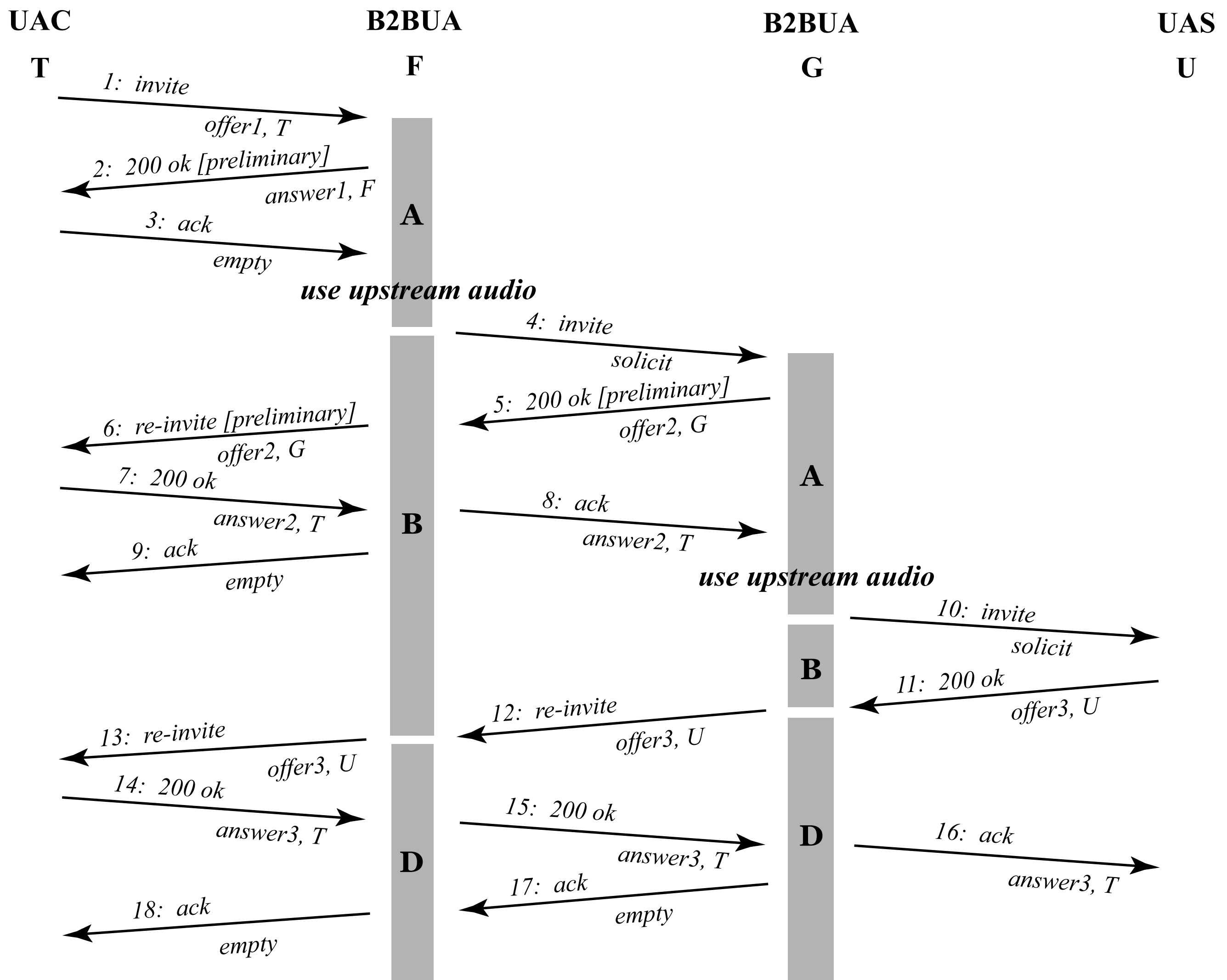

Figure 8: Implementation of upstream audio user interfaces in SIP.

5.2 Using the audio channel to the upstream end

We first consider how feature modules that are refinements of the meta-program can use the audio channel to the upstream end.

Figure 8 is a message-sequence chart showing signaling along the chain pictured in Figure 7. This configuration illustrates a feature module with an ordinary SIP client upstream, a feature module with an ordinary SIP client downstream, a feature module with a feature upstream, and a feature module with a feature downstream.

In the figure, a request eventually reaches end-to-end, as does a success response. At the end of the signal sequence, T and U are connected. However, on the way to this result, both F and G use the audio channel to T in state A of the meta-program. The figure shows the states of F and G at all times.

The label above each arrow shows the type of SIP signal and additional fields that are being used. The label below each arrow shows the session description contained in the signal, in an abbreviated form. The meaning of *offer* and *answer* are obvious [12]. Offers and answers are numbered to show their correspondence. Both *solicit* and *empty* indicate no real session description. An invite or re-invite without a session description is called *solicit* because it is soliciting an offer. Ack signals without session descriptions are merely *empty*. For offers and answers, the figure shows explicitly the media endpoint represented by the signal.

The two feature modules F and G use media servers as in Figure 7, but the media servers and the signaling with them are not shown in Figure 8. When the names F and G are used in signals to indicate media endpoints, they actually refer to the corresponding media servers. In actuality, signal 1 is forwarded by F to a media server, signal 2 comes from the media server and is forwarded by F to T, etc.

When a feature module receives a request by means of an *invite* (signals 1 and 4) and generates *200 ok* locally to prepare for using the audio channel, it adds the tag *preliminary* in an extra field (signals 2 and 5). This tag is an extension to SIP. The tag will mean nothing to a user agent, but it

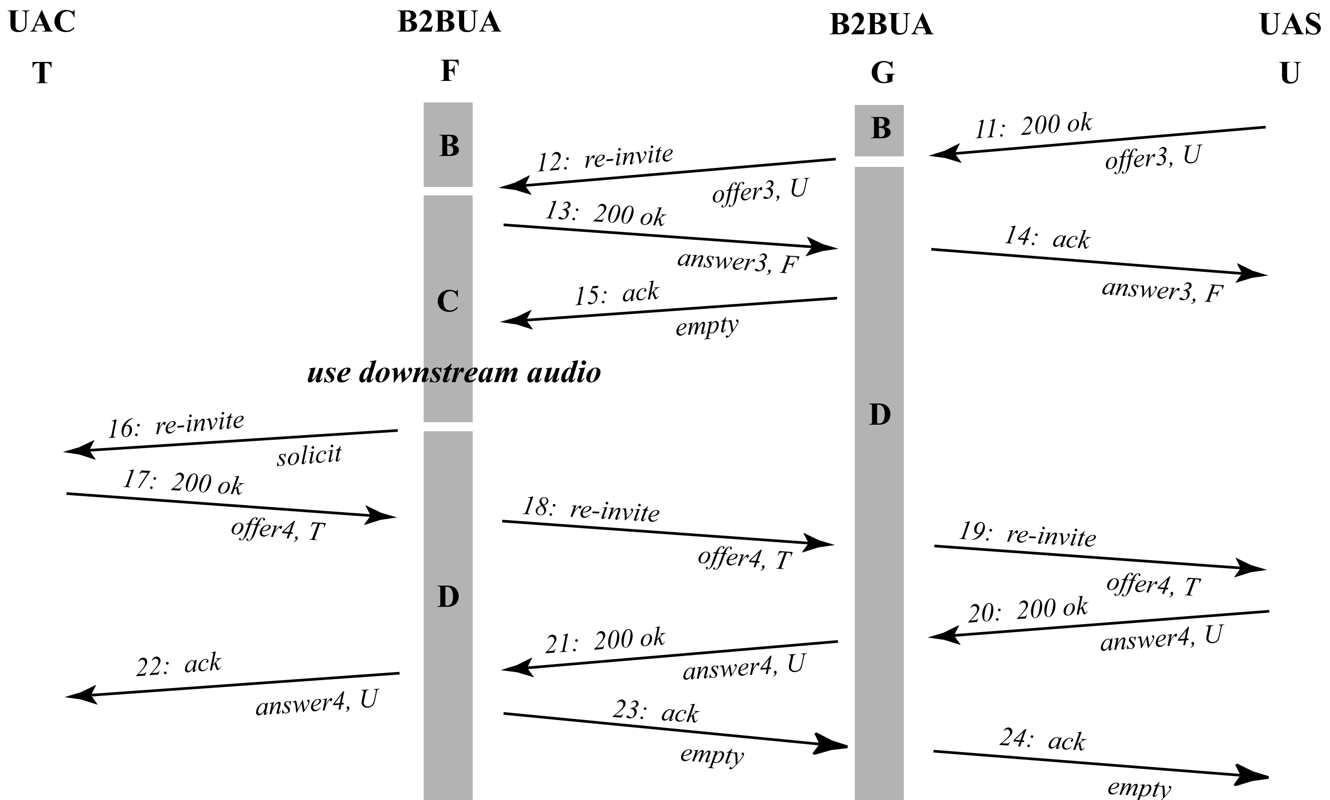

Figure 9: Implementation of a downstream audio user interface in SIP.

will indicate to another feature module that the signal is *not* an implementation of *success* in the meta-program.

A *200 ok* from a UAS will not have this tag, so it will be interpreted by a feature module as *success* in the meta-program (signal 11). Signals 12 and 13 also implement *success* in the meta-program.

The general way in which offers and answers are handled should not be surprising, as it is similar to many third-party call control scenarios [11]. When a feature module is behaving transparently, it forwards offers and answers faithfully. Offers can travel in *invite, re-invite* or *200 ok* signals. Answers can travel in *200 ok* or *ack* signals. The transparent module may need to change the type of the signals in which an offer or answer is traveling, to fit the state of the dialogue into which the signal will be sent.

For example, feature module F is logically transparent from signal 4 onward. It changes a *200 ok* with an offer to a *re-invite* with an offer (signals 5 and 6). It changes a *200 ok* with an answer to an *ack* with an answer (signals 7 and 8). Feature module G behaves similarly from signal 10 onward.

Figure 8 is a pattern that will work no matter how many feature modules in the chain wish to use the audio channel to the upstream end.

5.3 Implementing the remainder of the meta-program

Figure 9 continues after signal 12 of Figure 8 in a different way. Signal 12 causes F to enter state C of the meta-program, and the figure shows how it creates an audio channel to U, uses it to communicate with the callee, and then sends *success* to T in signal 16.

As in the previous section, this pattern will work no matter how many feature modules in the chain wish to use the audio channel to the downstream end.

If a feature module needs to generate an upstream failure after it has already sent a preliminary *200 ok*, then it simply sends *bye*. An upstream feature module that has not yet received a final *200 ok* when it receives *bye* interprets the *bye* as *failure*.

If a feature module receives *183* with a session description (early media) from downstream, then the feature module is in state B, and there is no problem with allowing a downstream module to use the audio channel. Depending on the state of the upstream dialogue, however, the transparent feature module may need to translate *183* into *re-invite*. SIP early media is not used in the basic implementation of the meta-program because it does not allow the final session description from downstream to differ from the preliminary session description. This makes it completely unsuitable for a chain of feature modules, each of which may have its own media server.

If a feature module receives *180* from downstream, it may not be able to forward the signal upstream, because of the state of the upstream dialogue. In this case, the feature module receiving *180* could generate ringback upstream without fear of audio contention.

5.4 Backward compatibility

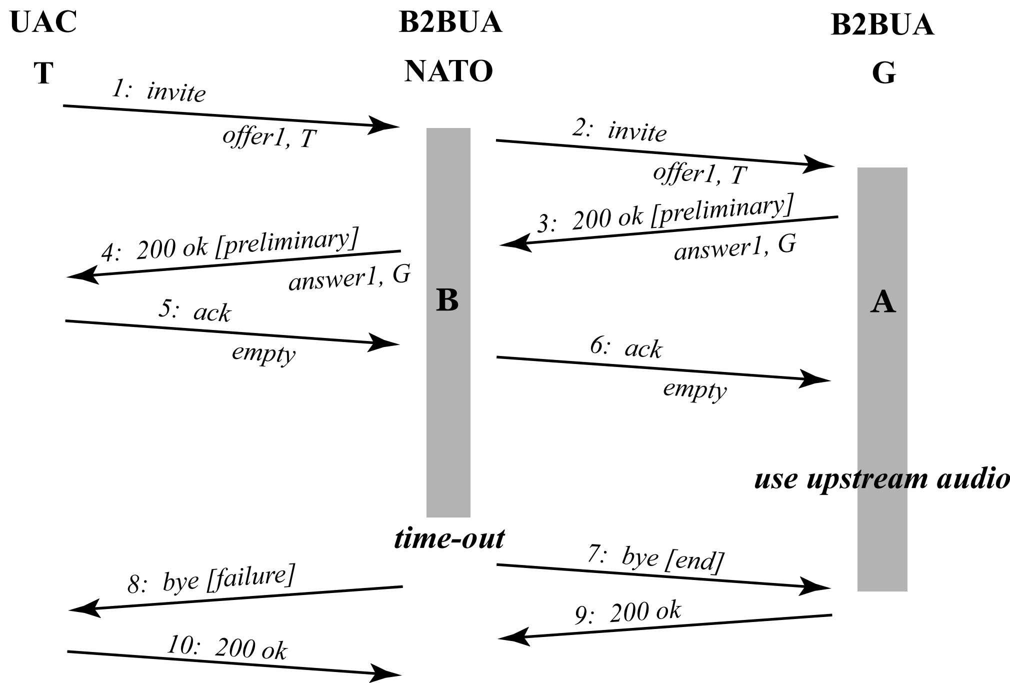

Figure 10: Implementation of No-Answer Time-Out in SIP.

Section 4.4 spoke of "an island of conforming components in a sea of non-conforming components." An island of conforming components will work successfully in a sea of SIP components that comply with [13], although, as noted, only contention among components on the island will be eliminated.

All components on the island must conform with SIP implementation of the meta-program, even if they do not use audio signaling. Figure 10 illustrates this point. The scenario in Figure 10 begins like the scenario in Figure 8, except that module F has been replaced by a No-Answer Time-Out (NATO) module.

This module sets its timer between signals 1 and 2. It does not cancel the timer on receiving signal 3, because it is only preliminary. When the time-out occurs, it sends a *bye* signal to G that corresponds in G to *i?end*, and a *bye* signal to T that corresponds in T to *o?failure*. Because of this behavior, NATO must be a B2BUA rather than a proxy.

The NATO example serves to illustrate certain protocol issues, but it is otherwise quite artificial. It is unwise to put NATO in the chain in front of G, because G uses audio signaling as a user interface to the caller, and this placement means that the time spent in audio signaling exhausts the time allowance for successful completion.

In a more practical arrangement, NATO would be placed on the shore of the island, so that the timer would not be set until audio signaling was completed. In this arrangement, the NATO module might be an ordinary SIP proxy.

6. RELATED AND FUTURE WORK

6.1 Analysis and management of audio interactions

There has been little previous work on analysis and management of audio feature interactions in VoIP. Most work on VoIP feature interactions concerns the interaction of features implemented within a single endpoint device, e.g. [14]. The endpoint device is assumed to be a personal computer, so there is no need for such features to communicate with their user by means of a audio channel. In general, the emphasis of most VoIP research is on exploiting the new opportunities provided by IP-based signaling. Not surprisingly, this de-emphasizes audio signaling.

In our laboratory, we take a broader view of VoIP. We are interested in supporting commercial voice-based services (in the network) as well as personal features (in user devices). We are committed to the interoperation of all voice networks, including the Public Switched Telephone Network and cellular networks. In this broader context, audio signaling is ubiquitous.

When overwhelmed with the difficulties of audio signaling, it is easy to miss the opportunities it provides. Consider, for example, the bad interaction of personal Parallel Ringing (PR) with cellphone Record Voice Mail (RVM), as discussed in Section 4.3.

This feature interaction is well-known in the VoIP world, and has been discussed by several authors. One proposal for resolving the problem is to require that cellphone RVM does not answer the incoming call until about 24 seconds have passed, so that a person has a chance to answer another parallel attempt [5]. A second proposal for resolving the problem is to require that the implementation of PR know which addresses have RVM, and to disallow parallel ringing of an address with RVM [10]. A third proposal is to set the SIP caller preference so that a voicemail server has the lowest priority as a callee.

In the commercial VoIP service we have worked with [1], none of these proposals would be effective.

- The first proposal is not feasible because cellphone networks are out of our service's control, so we have no power to alter the behavior of their RVM features.
- The second proposal is not feasible for roughly the same reason—we have no ability to probe the subscriber information of a cellphone network. Even if it were feasible, it would not be satisfactory, because it would force us to disallow parallel ringing to cellphones, while that capability is very popular with our customers.
- The third proposal is also infeasible because the cellphone network is not using SIP signaling. The caller preference will not be enforced in the cellphone network, and it cannot be enforced by PR in the VoIP network because PR has no way to distinguish between voicemail and human callees. Also, it is a little odd to fix a callee feature interaction with a caller capability, because the caller does not know that the callee has these features or this feature interaction.

Our solution to the problem is the use of Answer Confirm, which employs audio signaling to distinguish people from machines, and can do so while interoperating with any voice network whatsoever. This is a good example of the benefits of audio signaling.

Section 4 outlined solutions to two categories of bad feature interaction: audio contention in handling initial requests, and lack of support for reversed and sequential progress tones. The third category, that of audio contention caused by added calls, requires future work to find one or more systematic solutions.

6.2 SIP implementation of audio signaling

Because of the limitations of SIP early media as defined in [13], there have been other proposals for providing early media in SIP in a more general way [3, 4].

Although these proposals have some overlap of goals with the mechanisms presented in Section 5, they appear to be less general. They are presented as providing the final endpoint with the capability to use early media upstream. There is no discussion of how to allow an application server that is not the final endpoint the capability to use early media upstream or downstream. There is, of course, no discussion of how multiple application servers can have this capability.

Nevertheless, it is clear that Section 5 is not the only way to implement the abstract protocol of the meta-program in SIP. It may be that the techniques of [3] or [4] could be elaborated to provide alternative implementations of the abstract protocol.

Media clipping is an important issue raised in [4]. As explained there, an offer/answer exchange in which the offer travels in a *200 ok* signal and the answer travels in an *ack* signal is subject to media clipping. The endpoint sending *200 ok* is free to send media immediately afterward, the media is likely to reach the other end before the *200 ok* signal, and therefore the other end may receive media before receiving even an offer on the signaling channel.

Given the current design of SIP, it is very difficult to correct this problem. The examples in this paper, as well as many other third-party call-control scenarios [11], show that the flexibility of *invite* signals that solicit offers is necessary. The problem of media clipping arises because the *200 ok* is both a user-interface event (answering the phone) and an end-to-end signal required for media setup. The Public Switched Telephone Network, in contrast, is ready to carry audio for a call before alerting begins at the callee's device.

Of course, the design of SIP gives two distinct functions to *200 ok* so that the user who answers a call can choose the media for the call. In the normal case the choice is made from those media offered by the caller, while in the solicit case, the callee chooses first. It is worth asking the following question: How, and how often, does the callee actually make such a choice? If seldom used in practice, perhaps it should not have such a big impact on VoIP architecture.

Finally, one of the biggest practical problems with any approach to audio signaling in VoIP is that it entails a substantial burden of programming. The general problem can be characterized as that of *compositional media control*: How can multiple, independent feature modules coordinate their signaling so that end-to-end media channels are arranged correctly?

This general problem has been solved with a distributed algorithm executed by concurrent feature modules in a signaling graph [16]. The algorithm allows them to manipulate media streams independently, for their own purposes, while ensuring that the resulting end-to-end media streams will be provably correct.

Our current challenge arises from the fact that the general solution in [16] does not work in SIP; it is built on a signaling protocol that is both faster and more compositional than SIP. Section 5 indicates how we are working to achieve full compositional media control in SIP. After perfecting our techniques, we will incorporate abstractions such as the meta-program in a high-level domain-specific language for programming SIP servlets, so that the language implementation can generate the programming details automatically.

7. CONCLUSION

This paper has defined and analyzed the feature interactions caused by audio signaling in VoIP, and outlined a method for managing two out of three of the elucidated subproblems. Our experience with a commercial VoIP service indicates that both the problems and the solutions are realistic.

Two assumptions shape the perspective and approach of this work:

- Signaling in a VoIP system has a *pipes and filters architecture*, in which service is controlled by chains of alternating feature modules and signaling channels. Physically, a feature module could be an application server, a servlet within an application server, a gateway, or some other component.
- All features, modules, and functions should be viewed *compositionally*. The compositional viewpoint says that there is no such thing as a unique component. If it is plausible that a system has one instance of some type of component, then it is equally plausible that the system has multiple instances of it. If a system has multiple instances of some type of component, then their functions must be composed correctly.

These assumptions are quite different from the usual VoIP perspective. However, the research reported here illustrates two arguments that can be made in their favor.

One argument for these assumptions is that they are realistic and robust. No matter how hard designers try to avoid them, in real systems, network components are likely to be present. Among other reasons, network components

provide interoperation between heterogeneous subnetworks, and they represent the interests and functions of essential third parties. Even if designers feel that it is important to minimize network components, it is prudent to assume that some will be present, and to design systems that will work when they are present.

A second argument for making these assumptions is that generality can lead to simplicity. For example, Section 5 shows how to implement both upstream and downstream audio signaling for any number of feature modules. This appears to require less extension to SIP than the proposal in [3] for implementing upstream audio signaling for the final endpoint only. This illustrates that a solution to a more general problem need not be more complex than a solution to a more specific problem, and it will certainly last longer. This is the fundamental argument for thinking compositionally.

Acknowledgments

My colleagues Greg Bond, Eric Cheung, Hal Purdy, Tom Smith, and Venkita Subramonian have made many contributions to this work. I am grateful for our long and productive collaboration.

8. REFERENCES

[1] 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.

[2] 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.

[3] G. Camarillo. The early session disposition type for the Session Initiation Protocol (SIP). IETF Network Working Group Request for Comments 3959, 2004.

[4] G. Camarillo and H. Schulzrinne. Early media and ringing tone generation in the Session Initiation Protocol (SIP). IETF Network Working Group Request for Comments 3960, 2004.

[5] Ken Y. Chan and Gregor v. Bochmann. Methods for designing SIP services in SDL with fewer feature interactions. In D. Amyot and L. Logrippo, editors, *Feature Interactions in Telecommunications and Software Systems VII*, pages 59–76. IOS Press, Amsterdam, 2003.

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

[7] 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.

[8] JSR 116: SIP Servlet API Version 1.0. Java Community Process, `http:// www.jcp.org/ aboutJava/ communityprocess/ final/ jsr116`, 2003.

[9] JSR 289: SIP Servlet API Version 1.1. Java Community Process Early Draft Review, `http:// www.jcp.org/ en/ jsr/ detail?id=289`, 2007.

[10] Jonathan Lennox and Henning Schulzrinne. Feature interaction in Internet telephony. In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*, pages 38–50. IOS Press, Amsterdam, 2000.

[11] 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.

[12] J. Rosenberg and H. Schulzrinne. An offer/answer model with the Session Description Protocol. IETF Network Working Group Request for Comments 3264, 2002.

[13] 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.

[14] Xiaotao Wu. *Ubiquitous programming Internet telephony end system services*. PhD thesis, Columbia University, 2006.

[15] Pamela Zave. Ideal connection paths in DFC. Technical report, AT&T Research, November 2003.

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