Internet Engineering Task Force   SIP WG
Internet Draft   Rosenberg/Mataga/Schulzrinne
draft-rosenberg-sip-app-components-01.txt   dynamicsoft/Columbia U.
March 2, 2001
Expires: September 2001

An Application Server Component Architecture for SIP

STATUS OF THIS MEMO

Abstract

An application server is defined as an entity that is capable of providing advanced features to users. Examples of features include call forwarding, call screening, debit card calling, web interactive voice response, etc. However, the set of functions needed to enable a broad range of such applications is quite large - it includes speech recognition, DTMF recognition and digit collection, text-to-speech synthesis, database interfacing, audio and video coding and decoding, audio and video bridging and mixing, and signaling, to name a few. Supporting such a large set of functions on the same box presents a major challenge. To solve this problem, the industry is proposing a decomposition of the application server into two components - a media server that handles the media component, and an application server that handles the call control, data, and signaling. The interface that has been proposed between these two elements is a control mechanism along the lines of MGCP or Megaco. In this paper, we propose an orthogonal decomposition, which breaks an application

server into application server components. Each component represents a application server in its own right, but it provides a well defined component that by itself may be a complete, but simpler, application.

1 Introduction

An observable trend in VoIP systems is the continuing decomposition of monolithic elements into component subparts, with the corresponding development of standardized interfaces between components. This kind of decomposition can be observed in the MGCP/megaco [1] gateway decomposition of a large gateway into a signaling gateway (SG), media gateway (MG) and media gateway controller (MGC), often referred to as a softswitch. Following that decomposition, the softswitch was further decomposed into a pure call control component (still referred to as a softswitch) and an application server (AS), which provides features and services. The AS was then decomposed, breaking it into a signaling piece (still referred to as an application server), and a media server (MS), which provides the media components of applications. Protocols like MGCP [2] and Megaco [3] have been proposed as the interface between an AS and MS.

This paper proposes an additional decomposition of an application server into application server components (ASCs). This decomposition is orthogonal to the MS/AS decomposition, and differs significantly in its goals and benefits. The primary motivation is the recognition that most complex (and interesting) applications require a common set of core pieces - speech recognition and text-to-speech, translation services, conference servers, messaging servers, etc. Each of these components is complex and a full-fledged application in its own right. In most cases, a complex application really doesn't care about the details of the operation of the component. In many cases, these components run on separate servers, and often, would be provided by separate providers. What is needed, then, is a well-defined, distributed interface to these application server components. Here, we motivate a distributed decomposition of applications into components, and then show why, for many of these, the interface is ideally suited for a distributed, session establishment and termination interface that follows a standardized pattern of addressing and parameter passing. We believe the Session Initiation Protocol (SIP) [4] is ideally suited for such an interface.

2 Why Decompose

The first question to address is "why decompose an application server".

Decomposition is the act of breaking a large, monolithic system into a number of smaller compoents that interact according to specified behaviors. Decomposition of large components offers a number of benefits:

Scale. As systems need to serve more and more users, there are two approaches to scaling up. One is to buy increasingly faster hardware, so that the monolithic servers can keep up with increasing use. The second is to distribute the work across components, so that multiple servers perform the work. Distribution is fundamentally cheaper, since the cost of large monolithic systems increases exponentially with capacity, compared to the linear increase in cost with multiple, smaller units. Distribution of work can be done through load balancing, where each server remains homogeneous, but the work is spread across numerous servers, or it can be done through specialization, where the work is split into separate functions, and each function placed on a separate server. Specialization is ideal in cases where the work has different requirements for it to be completed. As an example, a component of an application may require special purpose hardware. This component can distributed to a specialized processor, with a normal off the shelf processor handling the more generic software tasks. Several of the components that we are describing fit into this category (such as the TTS server).

Sharing of resources. By decomposing a server into components, a many-to-many interaction between them becomes possible. This means that one component can provide services to many other components. This provides for sharing of resources, which ultimately results in capital cost reduction.

Expertise. Building a complex application requires expertise in call control, media services, compression, web, speech recognition, etc. It is highly unlikely that one organization will have enough expertise in all of these to build them all. By decomposing an application server into subpieces, organizations with expertise in one particular piece can build that one. The result is that the complete system can be composed of best in breed components.

Speed of deployment. By decomposing, upgrading existing applications and deploying new ones becomes simpler. The decomposition provides isolation. This isolation means that

one component can be changed or improved without affecting others. That makes it easy to add new features to an application, or to deploy a new one by using components already deployed.

Decomposition does have its drawbacks. Primary amongst them is security. In general, the more boxes in a system, and the more they interact with each other, the more complex the security is. As a result, any distributed system has inherently more complex security issues. Another drawback is reliability. A system with multiple boxes, where the system requires all boxes to work in order to function, is less reliable than a system with a single box which must work.

3 Tightly Coupled Decomposition

As an example of decomposition, it has been proposed to break the application server into a signaling and control component (the AS), plus a media server component (the MS). This decomposition is shown in Figure 1.

Calls arrive at the AS component over SIP. The AS then accesses the MS using MGCP, and learns the IP address and port where the media for the call can be sent. This is returned in the 200 OK response by the AS. The AS then begins to instruct the MS to perform specific functions - collect digits, play tones and announcements, and to report the digits and tones back to the AS for further processing. Typically, the MGCP interface between the two devices is fairly "busy"; there is a lot of messaging for complex applications.

In this model, there is a tightly coupled relationship between the MS and AS. The MS cannot function without the AS, and the AS needs to perform tight, low-level controls over the detailed operation of the media server.

To some degree, breaking of an application server into these two components represents an implementation detail of how one builds a large, monolithic application server. It is not generally practical for the two components to be owned by separate providers, due to the master/slave relationship between the two.

This decomposition also does not provide a true separation of function. Most applications that require media interaction (IVR, credit card and debit card, etc.) have very cleanly separated media phases and signaling phases. The details of the media interactions are usually not important to the signaling component, and vice a versa. As an example, consider a debit card application. The

```
                         ....................
                         .                  .
                         . +-------------+  .
                         . |             |  .
               SIP       . |             |  .
              -------------+      AS     |  .
                         . |             |  .
                         . |             |  .
                         . |             |  .
                         . +-------------+  .
                         .        |         .
                         .        |         .
                         .        |         .
                         .        |MGCP     .
                         .        |         .
                         .        |         .
                         .        |         .
                         . +-------------+  .
                         . |             |  .
                         . |             |  .
               RTP       . |             |  .
              -------------+     MS      |  .
                         . |             |  .
                         . |             |  .
                         . +-------------+  .
                         .                  .
                         ....................
                          Complete Application
                          Server
```

Figure 1: MGCP-based decomposition

application starts with the user making a call. As part of the call processing, interaction is needed with the user via the media stream to determine the debit card number. The precise set of menu operations and interactions used to obtain this number aren't important to the call/signaling processing piece; only the result (the number), is important. Once the number is returned, media processing ceases, and data and call processing commence. The debit card is looked up in a subscriber database, and if enough time remains, the call is completed. The signaling component monitors the call, and when the card has run out of minutes, the call is

terminated.

Consider the case where the application provider decides that the menus presented for debit card collection are confusing, and they need to be changed. This change really affects the media processing only; ideally, we would like to have no change whatsoever in the data processing and signaling part of the application. However, in the decomposition afforded by MGCP, the AS component contains both the signaling and call control, in addition to the control of the IVR menus and and processing. Thus, the AS needs to be updated, even though what has changed is really an IVR component.

The MGCP decomposition also presents a burden for software developers on the AS. They need to understand, and program, the detailed interactions with the MS that are provided by MGCP, in addition to the detailed signaling and data processing operations. The developers will also need to build and manage the low level state representing the controlled entity, which can be painful. The result is longer development times, less code reuse, and slower innovation.

It has been argued that one of the benefits of the MGCP decomposition is that it offloads the "burden" of call control from the media server. However, from a complexity standpoint, the MGCP processing required is probably on par with (if not more than), the simple amount of call control and event processing needed if SIP and VoiceXML were used.

From a reliability perspective, an MGCP style decomposition is less desirable. Since the components are strongly coupled, the system will fail so long as any of the pieces fail. Failure can also be introduced because of additional network resources needed for communications between the boxes. The result is that the MGCP decomposition may actually increase the probability of failure, as compared to no decomposition at all.

Another decomposition that has been proposed is to break a proxy into a routing and call control component, plus a services component. The interface between the two is then a transactional interface for services, similar in concept to INAP, based upon state transitions within a call model. This is another form of tight coupling, since it requires the services component to have detailed knowledge of the operational model of the call control component. We believe that this decomposition is limiting, for the same reasons the AS/MS decomposition is limiting.

4 The Decoupled Model

4.1 Architecture

As a result of this, we see the master/slave decomposition as being ideal for a single vendor to build a large system. However, this decomposition does not solve the other distribution needs we have motivated above. As a result, we propose that the AS be decomposed into an application component responsible for coordinating the overall execution of the application (called the controller), and application server components that provide pieces of the overall application. These components are only loosely coupled with the coordinating application server. The loose coupling implies that the interaction between them is the same as the interaction between the user and the coordinating application server, which is, in turn, the same as the interation between the application server components and other application server components. The components can easily be from separate vendors, and the interactions support the needed security and routing features to allow them to be owned by separate providers, even.

The architecture is shown in Figure 2.

The goal of the decoupling is to break the application into as coarse-grained pieces as possible. Each component (the coordinator included) should need to know as little as possible about the detailed operations performed by other components. A coarse-grained decomposition means that there is a clean and simple break in the functionality provided by the components. This enables significantly simpler interfaces between those components.

Each component is really interested in passing a request for service to another, letting the other component perform its task, and then getting the final result of the task back as an output. From a software engineering perspective, this represents the classic function call; the call signaling component is making a function call to the media part. It is interested only in the return value - the debit card number, for example - and does not really care about the implementation of it. From a protocol perspective, this is a classic client-server system. The client makes a request of the server, and the server does whatever it needs to do to return the final response. The problem more closely resembes the client-server system than the function call, however. This is because we need the interaction to be across the network, rather than between code within the same process. This is because one of the key concepts here is that components can be provided by separate service providers.

In such a model, where does the state for the sessions live? Here, we define a session as the complete set of interactions amongst all components for the delivery of the service. Thus, a session might span multiple protocols, and even multiple calls. Not surprisingly,

session state is distributed amongst the components, and the distribution follows the architectural model of Figure 2. The top level server, the controller, maintains the high level pieces of state that deal with overall delivery of the service, and the state required to coordinate the interactions with the component servers. Each component server maintains only the state needed to execute their component, and to manage interactions with components below them. A component server does not know about the complete service being delivered, and does not know about sibling servers. This aspect of our model - hierarchical distribution of session state, leads to one of the primary benefits of the architecture - ease of development. Someone building a new application by reusing existing components only needs to manage the high level state for delivery of the service. State related to the details of operation of one of the components - timings between digits in an IVR server, for example, is not relevant to the coordinator, and does not need to be managed.

The difference between classic RPC or client/server interactions and the interactions between the components here is that the relationship between the components represents a long lived association (i.e., a session), during which a session level service is being provided, rather than a simple input/output service. As an example, consider a component providing continuous real-time text-to-speech translation services. The application coordinator that wishes to use this service acts as a client, initiating a request for service to the server (in this case, the TTS server). However, the text is not passed as an "argument" to the TTS server, it is continually streamed for the duration of an active session, and the TTS server would continuously stream back the speech version of the text, which is the output of the service.

Another example is a voice messaging server. The messaging server provides basic services like message drop, message retrieve, and message management. Each of these represent procedures that can be executed by a client component. To drop a message, for example, the client component would initiate a session with the messaging server. A prompt would be played over that session, something like "please record your message for Joe now", and then the component takes the media input stream, records it, and saves it. When it is done, the session is terminated.

In some cases, the session may require a "side channel" over which intermediate data is passed, needed to control the session interactions from that point forward. IVR is the classic example. In some cases the coordinating application server can kick off the IVR script, and then only get back the final result - a menu option, a credit card number, or what have you. In other cases, the coordinating component may need to get intermediate results, so that

```
                            +-----------+
                            |           |
                            |           |
                            |  AS       |
                            |coordinator|
                            |           |
                            |           |
                            +-----------+
                   SIP,     --    \    ---
                    RTP?  --       \      ----      SIP,
                        --         \          ----   RTP?
                      --            \ SIP,        ----
                    --               \ RTP?           ----
                  --                 \                    --
            +----------+        +-----\----+         +----------+
            |          |        |          |         |          |
            |          |        |          |         |          |
            |          |        |          |         |          |
            |  ASC     |        |    ASC   |         |   ASC    |
            |          |        |          |         |          |
            |          |        |          |         |          |
            +----------+        +----------+         +----------+
                                       \                    /
               /                        \\  SIP,           /
              / SIP,                      \  RTP?        //
             /   RTP?                      \\           / SIP,
            /                                \         /   RTP?
           /                               +----------+
     +----------+                          |          |
     |          |                          |          |
     |          |                          |          |
     |          |                          |   ASC    |
     |   ASC    |                          |          |
     |          |                          |          |
     |          |                          +----------+
     +----------+
```

Figure 2: Decoupled Architecture

it can guide the operation of the IVR moving forward. This requires a companion control channel that provides data output from the component server back to the client, and then returns further high level instructions from the client back to the server.

There is a thin line in some cases between this control channel and the tightly coupled interactions of a master-slave MGCP relationship. However, the loosely coupled nature of the interaction can be maintained by using coarse-grained data passing over a distributed client-server protocol, such as HTTP or Corba.

From this architectural description, it is clear that a client-server session establishment protocol, which allows for passing of parameters that describe service, is the ideal mechanism to coordinate the interaction between components. Clearly, SIP is perfect in such a role.

Following the example above, an IVR application server component would be completely responsible for the execution of the IVR piece of an application, including both the media and the signaling call control. It would know the menus to maneuver through, and it would know when to collect digits and present prompts. The coordinating application server would request service from the IVR component by initiating a call to it (possibly using third party call control [5] to direct the media directly to the IVR without passing through itself; more on that below). The application component takes the media from the incoming call, running it against the IVR application. When the IVR is done, the final result - in this case, the credit card number, is passed back to the coordinating AS, possibly throug an HTTP POST operation. The coordinating AS then terminates the call with the IVR.

4.2 Benefits of the Decoupling

This decoupled interaction between components provides several important benefits:

Separation of Businesses. The decoupled interaction between components is needed to allow the components to be provided by separate providers. Master-slave control interactions do not work well across service providers, let alone across vendors. By allowing separate providers to offer the components, new businesses can be created that specialize in the piece they are providing.

Rapid Development. Since the components can easily be placed in separate boxes from separate vendors, or even in separate

providers, we achieve a separation of function that allows each piece to be developed in complete isolation. We also get reuse of components for new applications. This allows for rapid service creation.

Better Interoperability. It can be argued that the decoupled interaction between components is more like to be interoperable that a master-slave mechanism. This is largely based on the assumption that a master-slave interaction requires a lot more messaging and exchange between the components, whereas the decoupled client-server mechanism requires less. The fewer information that passes back and forth, the easier it is to interoperate.

Architectural Flexibility. The loose coupling of the components means that a server, such as a conferencing application or IVR, need not be implemented as an actual server. Rather, complex networks of components, with proxies providing routing of requests in arbitrarily complex ways, can be built to provide a service. Since the interaction is SIP, the application controller accessing the service doesn't know whether it is communicating with a single server or a network built in this fashion. That allows ASPs flexibility in how they can construct their service networks.

Reliability The loose coupling of the components improves reliability compared to a tight coupling. Thats because the system can probably still continue to operate in the failure of a single component. For example, if a TTS server fails during a session, an application server can use a server from a completely different provider, or it can use a media server instead, converting the text to VoiceXML scripts. Depending on the service, the TTS component could possible be skipped altogether. Note, however, that the reliability is still not as good as a monolithic system. Having ten identical boxes each running a complete set of services is better than spreading the service across ten boxes, where some subset cause total failure.

5 Architecture for the Interfaces

Up to now, we have been fairly vague about exactly how such an interface would work in practice. We have argued that it is SIP, but not described in detail how SIP is actually used for this function.

SIP (along with SDP [6]) clearly provides the facilities for initiation and termination of the sessions between the controller and components, and for specification of the media addresses to and from which media is sent. However, SIP leaves a lot of flexibility in terms of naming, additional message content, session duration, and control. Here, we discuss each of these in turn.

5.1 Naming

In any remote procedure call system, a key component is naming. The identified resource must be properly addressed so that the underlying message passing system can properly determine where the request should go.

The same is true in SIP. Messages are routed based on the request URI, as it serves as the primary naming tool for routing messages. In its application to AS component interaction, the request URI serves as the primary tool to identify the resource to which the session is addressed. A critical piece of defining a session level service that can be accessed by SIP is defining the naming of the resources within that service. This point cannot be understated.

As an example, consider a conferencing service. In this case, the primary resource that is being accessed is a mixing service. We would like to have a way to identify which conference is being addressed by any given call. All calls for the same conference are all bridged together. By default, the bridging would operate in an N-1 configuration (that is, each user receives a mixed media stream that represents all of the other users besides themself). Conferences can be set up in two ways - ad-hoc, which are not pre-established at all, and exist so long as there is a participant in them, and scheduled, where they exist for a certain period of time.

One might imagine that a conferencing service breaks its URI namespace into two pieces - one piece that represents ad-hoc conferences, and another that represents scheduled conferences. Ad-hoc conferences are addressed using a URI of the form <conference ID>.adhoc@conferences.com. All users who initiate a call to the URI sip:as9dahas89.adhoc@conferences.com are bridged together. The conference state is established when the first call to a conference occurs, and destroyed when the last call terminates. In contrast, scheduled conferences might be named by <conference id>.scheduled@conferences.com, so that a call to sip:conference12.scheduled@conferences.com allows a user access to a pre-arranged conference.

There are several benefits to naming ad-hoc conferences vs. scheduled ones in this fashion. The primary one is convenience; the name makes

it the type of conference apparent to any entities that are interested. Secondly, it can avoid certain misconfigurations. Let's say there are no conventions for naming of ad-hoc versus scheduled conferences. I am asked to join a scheduled conference (conf2321@conferences.com), but I mis-type the URL in my browser (conf2123@conferences.com). I don't want this to drop me into an ad-hoc conference where I sit for 15 minutes thinking others will eventually join. If ad-hoc conferences are named differently, a call to cond2123@conferences.com is never going to be an ad-hoc conference, and so my call will be rejected immediately.

For an application server to use a conferencing service as a component, the AS must know the URI namespace conventions used to identify the various conferences. The above information, for example, would be provided by the conferencing provider to its customers.

This same concept of using the request URI as a service identifier has been described in detail for voicemail systems [7].

The great advantage of using the request URI as a service identifier comes because of the combination of two facts. First, unlike in the PSTN, where numbers are limited, URIs come from an infinite space. They are plentiful, and they are free. Secondly, the primary function of SIP is call routing through manipulations of the request URI. In the traditional SIP application, this URI represents people. However, the URI can also represent services, as we propose here. This means we can apply the routing services SIP provides to routing of calls to services. The result - the problem of service invocation and service location becomes a routing problem, for which SIP provides a scalable and flexible solution. Since there is such a vast namespace of services, we can explicitly name each service in a finely granular way. This allows the distribution of services across the network. In the conferencing example above, since we have separated the names of ad-hoc conferences from scheduled conferences, we can program proxies to route calls for ad-hoc conferences to one set of servers, and calls for scheduled ones to another, possibly even in a different provider. In fact, since each conference itself is given a URI, we can distribute conferences across servers, and easily guarantee that calls for the same conference always get routed to the same server.

This is in stark contrast to conferences in the telephone network, where the equivalent of the URI - the phone number - is scarce. An entire conferencing provider generally has one or two numbers. Conference IDs must be obtained through IVR interactions with the caller, or through a human attendant. This makes it difficult to distribute conferences across servers all over the network, since the PSTN routing only knows about the dialed number.

Care must be taken not to push this concept too far. Naming of services should not become so fine-grained that all parameters associated with the service simply become encoded into the request URI as well. The right level of granularity can be determined based on routing. If a service is represented by multiple URLs, but requests for each of those URLs are always routed in the same way, the naming is too fine-grained.

5.2 Additional Message Content

Sometimes, connecting to a service requires the service to know additional information that is not appropriate for the request URI. As an example, the conferencing server might need to know the name, address, phone number, company, and email address of the participants, which it converts to speech and uses as an announcement when the user joins and leaves the bridge.

This kind of content can easily be carried in the body of the SIP messages used to establish and manage the session with the service. For simple data, SIP headers may be appropriate. In the conferencing example above, the conferencing service might mandate that a vCard be attached to all INVITEs, in order to provide that information.

When existing data formats (like a vCard) are not defined to provide the needed information, it can be encoded in an XML document, for example, and carried along in the INVITE.

Each service would need to specify the content that it needs in order to process the session invitation.

5.3 Session Duration

The duration of the session that is established with a server depends entirely on the nature of the service. For example, for a conference, the initiation of the call begins the mixing service for that user, and the termination of the call results in that user leaving the conference.

For an IVR service, the INVITE request begins the interaction with the service. Once the INVITE transaction completes, the IVR would play out the initial prompt, and begin collecting data from the caller. How the IVR terminates depends on its usage. When the initiator of the service is an application server, we would argue that in almost all cases, it should be the responsibility of the controller to determine when the interaction is complete (and thus terminate the call with a BYE). However, when the initiator is an end user, the IVR will usually be the one to terminate the session. We discuss IVR interactions in more detail below in Section 6.1.

5.4 Third Party Call Control

Third party call control, as defined in [5], plays an integral role in this architecture.

In many cases, the controller orchestrating a service wishes to invoke the resources of an IVR or conferencing server. However, the AS is not the actual source of the media that drives the IVR. The source of the media is the end user that initiated the call to the controller. What is needed, then, is a way for the AS to call the IVR or conferencing server, and pass it the media information of the end user. Similarly, the media address of the IVR server (described in the SDP from the media server), needs to be passed to the end user that initiated the call. By using third party call control, an application server can direct the media of the end user to and from the components that it is using to provide the application. Once one service is complete, the controller can move the media to a different component. SIP re-INVITEs also allow the controller to request the caller to send multiple media streams, one, for example, containing only DTMF and tones. This allows for DTMF control of services without carrying DTMF in SIP itself.

Figure 3 shows how we use a component server to collect DTMF input for a service; specifically, a simple (and perhaps useless) service that allows a caller to press '1' to indicate that they want to put the call on hold. The service is, in principal, useless, since hold is so common that the end user can do this themselves. However, it is useful for example purposes.

The caller sends an INVITE request to the called party (1), which is routed to a server handling calls for the domain of the called party. In this case, the server is an application server. The AS decides that it would like to offer the caller advanced services based on DTMF events sent mid-call. As a result, it decides to invoke the services of a media server component. The AS will use third party call control mechanisms to have the caller send any DTMF related media to the media server, in addition to sending its media to the called party. To accomplish this, the AS sends an INVITE to the media server (2), with an indication that the media stream is send only (this is accomplised using the sendonly SDP attribute [6]). The request URI of this INVITE binds that session to a service that looks for any in-band DTMF, and reports it back to the AS through an HTTP GET or POST operation. In section 6.1, we show how this is easily done with a VoiceXML driven IVR server.

The media server responds with a 200 OK (3) that contains SDP with the address where the media should be sent to. The application server

ACKs this response (4), and holds on to that SDP. The AS then proxies the original INVITE request (5), and the called party answers the call (6). This acceptance is proxied upstream (7), and then acknowledged (8,9). At this point, media is flowing between the caller and called party (10). The next step for the AS is to get a stream of DTMF digits to flow from the caller to the media server. To do this, it sends a re-INVITE to the caller (11). This re-INVITE contains the same SDP as the response (6) from the called party, but with the addition of a new media line. This media line is audio, and contains a single codec, the RTP payload format for DTMF and tones [8]. The connection address and port are from the SDP returned from the media server. This tells the caller to send an additional media stream to the media server, using only the DTMF codec. The result is that RTP packets are sent only when the caller presses a button on the phone.

The caller accepts this re-INVITE (12), and the AS acknowledges it (13). Now, DTMF only RTP is flowing between the caller and the media server (14). At some point later, the caller presses the 1 key (which, for example, might imply call hold). This is processed by the media server, and the result is an HTTP request being sent to the AS (15). The HTTP request contains the value of the collected digit. The AS receives this request, and knows that the user keyed in a 1. Recognizing this input as call hold, the AS sends a re-INVITE to the called party (17). The SDP in this re-INVITE is the same as the SDP in the original INVITE from the called party (1), except that the connection address is set to zero, indicating call hold. The called party accepts the re-INVITE (18), and this is ACKed by the AS (19). The called party is now on hold.

Note that the call flow remains unchanged if the stimulus were based on voice recognition instead of DTMF. The only difference would be that a general purpose codec, such as G.711, would be used instead of RFC 2833 for communications between the caller and the media server. This achieves an important unification. Independent of the type of stimulus - voice, DTMF, or, in fact, direct http requests from the caller (if they were using a softphone), the service execution code is unchanged.

Others have proposed that DTMF digits be carried in SIP directly from the caller to the AS [9,10]. However, this approach does not work for anything beyond DTMF, while our approach works for DTMF, speech, and web interfaces. Another drawback of the DTMF-in-SIP approach is that all entities on the call signaling path will receive any DTMF digits dialed by the called party. Furthermore, since the caller doesn't know if there is an entity interested in DTMF, it is required to send DTMF within SIP messages all the time, even if no entity is interested.

```
    Caller          Coordinator         Media Server       Callee
      |                |                  |                 |
      |(1) SIP INV     |                  |                 |
      |--------------->|(2) SIP INV       |                 |
      |                |----------------->|                 |
      |                |(3) 200 OK        |                 |
      |                |<-----------------|                 |
      |                |(4) SIP ACK       |                 |
      |                |----------------->|                 |
      |                |(5) SIP INV       |                 |
      |                |----------------------------------->|
      |                |(6) 200 OK        |                 |
      |(7) 200 OK      |<-----------------------------------|
      |<---------------|                  |                 |
      |(8) SIP ACK     |                  |                 |
      |--------------->|(9) SIP ACK       |                 |
      |                |----------------------------------->|
      |(10) RTP        |                  |                 |
      |.....................................................|
      |                |                  |                 |
      |(11) SIP INV    |                  |                 |
      |<---------------|                  |                 |
      |(12) 200 OK     |                  |                 |
      |--------------->|                  |                 |
      |(13) SIP ACK    |                  |                 |
      |<---------------|                  |                 |
      |(14) RTP        |                  |                 |
      |...................................|                 |
      |                |                  |                 |
      |                |(15) HTTP GET     |                 |
      |                |<-----------------|                 |
      |                |(16) 200 OK       |                 |
      |                |----------------->|                 |
      |                |                  |                 |
      |                |(17) SIP INV      |                 |
      |                |------------------+---------------->|
      |                |(18) 200 OK       |                 |
      |                |<-----------------+-----------------|
      |                |(19) SIP ACK      |                 |
      |                |------------------+---------------->|
      |                |                  |                 |
      |                |                  |                 |
      |                |                  |                 |
      |                |                  |                 |
      |                |                  |                 |
```

Figure 3: Call Flow for DTMF Enabled Hold Service

There have been proposals for adding a subscription/notification mechanism on top of this to avoid this problem. However, this further complicates the system by adding a requirement for the caller to support a subscription and notification service just for DTMF.

Our approach fits well within the existing SIP framework, and requires no additional work from the end users. Furthermore, it transparently supports multiple application server components receiving DTMF. This is because an AS is able to send a DTMF stream to a component by adding a new media line to the list of media streams being sent by the caller. The list of media streams being sent by the caller is observed by each AS through the initial INVITE, along with any subsequent re-INVITEs which might modify it. Consider the situation with two application servers, A and B, depicted in Figure 4. The original call setup starts with the caller, flows through A, then B, then the called party. At some point later, A sends a re-INVITE (10) to the caller, adding a media stream, just as described in Figure 3. The SDP in this INVITE will be the same as provided by the caller in message (1), plus the additional DTMF stream. Note that this re-INVITE does not pass through B. Now, B decides to add a media stream for DTMF. So, it sends a re-INVITE (13). This goes first to A. As far as A is concerned, this re-INVITE is from the called party. A computes the difference between what it believes the called party should perceive as the set of media streams, and what is in the re-INVITE (13). This difference (the additional DTMF stream added by B) is added to the SDP that A had sent to the caller previously (10), and the result is sent in a re-INVITE to the caller (14). This SDP now contains the media streams meant for the actual called party, along with two DTMF streams; one for A, and one for B. The caller thus sends DTMF to both servers.

A further advantage of our approach is that the DTMF can even be sent using multicast, since it is being sent in RTP rather than as part of SIP. This allows for tremendous scalability, if needed, in the number of entites receiving the DTMF streams.

5.5 Side Channels

Side channels are used for passing of events from the application server components back to the client, and for passing control commands from the client to the application server component.

Unfortunately, side channels complicate the simple session level interface between components. It is our belief, at least for the components described here, that only minimal side channels are needed. Specifically, the only service below that requires one to be effective is the IVR service, for which HTTP forms an ideal side

channel. If the side channel becomes so complex as to introduce extensive synchronization, bandwidth, and transactional issues, the relationship between the components becomes tightly coupled once more, and the benefits we are espousing here begin to disappear.

As such, we believe that a reasonable side channel for decoupled server interactions is defined as follows:

o The event reporting and control components have no real time requirements.

o Event reporting from the component back to the client accessing it are infrequent; specifically, the intervals are much larger than the round trip times between the client and the component.

o Control from the client to the component is infrequent; specifically, the intervals are much larger than the round trip times between the client and component.

o Event reporting is coarsely granular, so that the client does not need to explicitly subscribe to specific events in order to avoid be overwhelmed with data.

o The amount of data passed in both the events and in the control is small.

o There are no requirements for transaction support.

Note that protocols like MGCP and megaco do not meet these requirements, as they require tight timing, synchronization, and explicit subscriptions. HTTP, as used in VoiceXML, however, does meet these requirements.

6 Patterns for Accessing Components

In this section, we propose a set of patterns that define the interaction of a controller with an application server component. These patterns manifest themselves in the description of the service invoked when a session is initiated, a discussion of the naming conventions of the service, and a description of any back channel used for control and data passing.

6.1 Interactive Voice Response Services

We have touched upon the basics of the interaction between a controller and an IVR server. The controller initiates a call to the server, the server executes some kind of IVR service, and data is

```
  Caller            A                B              Callee
    |               |                |                  |
    |(1) SIP INV    |                |                  |
    |-------------->|(2) SIP INV     |                  |
    |               |--------------->|(3) SIP INV      |
    |               |                |----------------->|
    |               |                |(4) 200 OK        |
    |               |(5) 200 OK      |<-----------------|
    |(6) 200 OK     |<---------------|                  |
    |<--------------|                |                  |
    |(7) SIP ACK    |                |                  |
    |-------------->|(8) SIP ACK     |                  |
    |               |--------------->|(9) SIP ACK       |
    |               |                |----------------->|
    |(10) SIP INV   |                |                  |
    |<--------------|                |                  |
    |(11) 200 OK    |                |                  |
    |-------------->|                |                  |
    |(12) SIP ACK   |                |                  |
    |<--------------|                |                  |
    |               |                |                  |
    |               |(13) SIP INV    |                  |
    |(14) SIP INV   |<---------------|                  |
    |<--------------|                |                  |
    |(15) 200 OK    |                |                  |
    |-------------->|(16) 200 OK     |                  |
    |               |--------------->|                  |
    |               |(17) SIP ACK    |                  |
    |(18) SIP ACK   |<---------------|                  |
    |<--------------|                |                  |
    |               |                |                  |
    |               |                |                  |
    |               |                |                  |
    |               |                |                  |
    |               |                |                  |
```

Figure 4: Multiple Application Servers and DTMF

possibly fed back to the controller with intermediate and/or final results of the IVR interaction.

1. How is the IVR service identified?

2. How can the controller specify the details of the dialog the IVR carries out with the user?

3. How does data from the IVR get passed back to the controller?

4. How is intermediate control performed (e.g., to interrupt or reset IVR based on some event at the controller, in this case)?

We believe that VoiceXML [11] represents the ideal partner for SIP in the development of distributed IVR servers. VoiceXML is an XML based scripting language for describing IVR services at an abstract level. VoiceXML supports DTMF recognition, speech recognition, text-to-speech, and playing out of recorded media files. The results of the data collected from the user are passed to a controlling entity through an HTTP form POST operation. The controller can then return another script, or terminate the interaction with the IVR server.

From a naming perspective, the primary issue is how a request URI is associated with a script to invoke when the call is answered. We see three primary mechanisms:

1. There is a one-to-one binding of the address in the request URI to a script to execute. These bindings are published by the provider of the IVR service.

2. The initial script to execute is actually carried as content in the body of the SIP INVITE request. The request URI indicates that the desired service is execution of content in the request (i.e., sip:executebody@servers.com).

3. The initial script to execute is fetched by the VoiceXML server; the URL to fetch it from is passed in the SIP INVITE message that initiates the IVR session. This can be accomplished either with the application/uri MIME type as a body, or using the new *-Info headers [12] which provide references to content to fetch.

We believe that the third approach is probably the best one. SIP is not the ideal transfer mechanism. Passing a URI allows a far better transfer tool, namely HTTP, to be used to actually fetch the script back from the controller.

HTTP is then also used to pass back form data from the IVR to the controller. The results of the HTTP POST can also contain additional

VoiceXML scripts to execute. It represents the side channel discussed in section 5.5

Note that in some cases, there needs to be interactions between the HTTP server that receives the HTTP POST requests, and the controller that initiates and terminates the SIP sessions with the IVR. This is the case when the data collected by the VoiceXML server is used to guide signaling behavior. For example, a pre-paid calling application might use the IVR to collect the users PIN code. The PIN code is looked up, and the number of minutes remaining is determined. This amount of time must be known to the SIP controller, as it will need to hang up the call once this time expires. Some kind of session sharing mechanism is needed between the SIP controller and the HTTP server in this case.

Figure 5 shows the interaction between an application server acting in a coordinating role, and an IVR server component. In this example, consider an application where the user makes a call, but the system needs additional information to determine where to forward it to. The user is prompted for the info, and once the name of the desired called party is obtained and looked up, the call is completed to the requested destination.

First, in step (1), the caller sends an INVITE to the controller. The controller then creates a brand new call to the IVR application server (2), using the SDP from the INVITE in (1). The IVR accepts the call (3), and the SDP from that acceptance is returned in a 183 response to the caller (4). The call to the IVR is acked (5), and now a media stream exists between the caller and the IVR server. The IVR server, in step (6), fetches the initial VoiceXML script to execute, which is returned by the controller (7). The prompts are played to the caller, and the identity of the called party is collected. This is passed to the controller through another POST (8), which returns an empty VoiceXML script (9)[1] complete, the controller hangs up with it (10 and 11). The information the controller got in the POST (8) is used to determine the next hop SIP server, and the initial INVITE is proxied there (12).

Its important to observe the all call control related to executing the service lives within the controlling application server. The IVR application server deals strictly with the media component. This division of work, as we have discussed above, allows for independent

[1] Note that it is unusual for an empty script to be returned; this is because we want the AS to maintain control of the call signaling

evolution of the call control and media components of services. For example, if the desired called party did not have a reachable SIP address, but they did have an email address, the call could be redirected to a mailto URL. To support this twist, only the controlling application server code need change. The media component remains completely and totally unchanged.

Readers familiar with VoiceXML will observe that VoiceXML almost achieves this perfect separation. It lacks any call control excepting a two - for call transfer and call termination. These tags are clearly not sufficient for many services. Our architecture would argue that instead of adding call control to VoiceXML, all control should be removed, so that call control can be left to other server components.

The separation of the control from the media component also allows the media component to change without affecting the control component. In fact, because of the http interface between the two, the media server can be completely removed and replaced with a normal web browser, with only a small effect on the call control component. As an example, if the calling party was coming from a web enabled SIP client (known by the presence of the Accept header with text/html as a value in the INVITE request), the controller could return an HTTP URL in the 183 with an actual web form that gets filled out by the caller. This would be instead of using an IVR server to collect the data. Interestingly, the representation of the collected data is identical in both cases. Both use an HTTP POST operation to send the data to the controller. This allows the data collection code in the controller to be unified across both voice access and web access.

6.2 Conferencing Servers

Conferencing servers today vary in type and complexity. Some are dialup only, supporting IVR access. Others support ad-hoc conferencing with web interfaces. Others still support three way calling as part of a PBX system.

We observe once more that all of these conferencing "servers" are really conferencing applications that are just bundled as a server. These conferencing applications can be decomposed into components in exactly the way we have described above. At the core of each of these conferencing applications is a mixing service. This service is responsible for taking N audio or video streams, mixing them according to some matrix, and returning the mixed stream to each participant. Issues such as conference policy, provisioning of conferences, and authentication are all completely separate and outside of this basic mixing component.

```
     |      INVITE (1)          |                           |
     |------------------------->|                           |
     |                          |        INVITE (2)         |
     |                          |-------------------------->|
     |                          |       200 OK (3)          |
     |                          |<--------------------------|
     |      183 (4)             |                           |
     |<-------------------------|                           |
     |                          |       ACK (5)             |
     |                          |-------------------------->|
     |           MEDIA          |                           |
     |------------------------------------------------------|
     |                          |                           |
     |                          |     HTTP GET (6)          |
     |                          |<--------------------------|
     |                          |     HTTP 200 OK (7)       |
     |                          |-------------------------->|
     |                          |                           |
     |                          |                           |
     |                          |                           |
     |                          |                           |
     |                          |     HTTP GET (8)          |
     |                          |<--------------------------|
     |                          |                           |
     |                          |     HTTP 200 OK (9)       |
     |                          |-------------------------->|
     |                          |                           |
     |                          |      BYE (10)             |
     |                          |-------------------------->|
     |                          |       200 OK (11)         |
     |                          |<--------------------------|
     |                          |                           |
     |                          |       INVITE (12)         |
     |                          |---------------------------------------->
     |                          |                           |
     |                          |                           |
     |                          |                           |
     |                          |                           |
     |                          |                           |
     |                          |                           |
     |                          |                           |


   Caller                    Controller               IVR Server
```

Figure 5: Interaction of App Server and IVR Component

For this reason, we argue that a large variety of conferencing applications can be easily constructed by having the mixing service as separate application server component.

What does the interface to such a mixing server look like? For the call control interface, users would join a conference by calling the server. The server would answer the call, thus appearing as a SIP UAS. The media sent from the user is mixed with other users in the conference, and the media sent back to the user is the mixed stream. The user can leave the conference by sending a BYE to the server, and the server can kick a user out of the conference by sending the user a BYE.

Since the primary resource being accessed is a conference, it is no surprise that we would argue that the request URI of an incoming call defines the conference a user is mixed in to. In other words, all users that call the server with the same request URI, are all mixed together. The conferences are not defined by Call-ID or other SIP header fields. Using the request URI has tremendous advtanges from a routing and naming perspective, as we have discussed more generally above.

It is not neccesary (in fact, not even advisable), for the conferencing server to require that the URIs that define the conference be set up ahead of time. Conference lifecycles in the mixing server are very simple. Conference state is created when the first call arrives for a particular URI, and ends when the last user with a call to that URI hangs up. This model allows the same mixing server to support both ad-hoc conferences, and pre-arranged conferences too. Pre-arranged conferences are handled through policy and control in a coordinating server external to the mixing server. This server lives entirely in the call control and signaling plane, not in the media plane.

SIP (and RTP, of course) alone is not sufficient for complete usage of a conferencing server. Media mixing policies (effectively, the matrix indicating which users hear which other users, and with what relative volumes) need to be set. Information on the status of the conference, such as the identity of the current speaker, number of users currently being mixed, etc., may need to be reported back to some control entity. These represent the requirements for the side channel. In IVR servers, the side channel used HTTP. We argue that to unify these concepts, HTTP is ideally suited here as well. Updates to the mixing policy can be made through HTTP POST requests against the mixing server, using well defined interfaces (possibly SOAP). Similarly, information about the status of the conference can be obtained through HTTP GET operations against the mixing server. The side channel here meets the requirements outlined in Section 5.5; it

is not real time in nature, does not reuqire transactional support, and passes relatively infrequent data and control. In fact, such a side channel will often not be needed at all. In 90 default mixing policy (the so-called N-1 matrix, where each user hears everyone but themselves, all at equal volume, with no floor control) will suffice.

Fans of the INFO method [13] will argue that instead of using HTTP for the control, why not INFO? This would eliminate the need for an additional protocol, after all. The answer is the same as to why SIP should not simply replace HTTP - the two have different strengths and weakenesses. SIP is a poor data transfer protocol. It has insufficent support for transfer of medium to large data sets, which is important here. Furthermore, we may want to allow an entity separate from the one that initiated the session to control the session. Usage of INFO would only work from the same device (because of the sequence numbering).

In the next few sections, we show how this basic application server component can be used, along with a controller and other components, to build more complex conferencing applications.

6.2.1 Web Scheduled Conference Services

In this application, we'd like a conferencing service where all conferences must be pre-scheduled. The pre-scheduling is done through a web page. At the page, the user will enter the start time (but not mandatory stop time) of the conference, the maximum number of attendees, and the identities of the attendees (if known). Once entered in a form, the server returns a SIP URL representing the conference.

To implement this, we use an coordinating application server that has a SIP and HTTP interface, along with the mixing application server just described.

Figure 6 shows a call flow for this service. A web client is first used to submit the information. Let us suppose a simple case where the conference can have up to two participants, and the conference starts immediately. The HTTP POST representing the form data is sent to the controller (1). It stores the information for the conference in a local data store, and chooses a SIP URL for the conference. This URL can be anything, so long as it is different from any URLs handed out so far by the controller. The URL is returned to the web client in step (2). As an additional convenience feature, the URL could be emailed to the participants. This would require the controller to have an SMTP interface, in addition to HTTP and SIP. Note that this SIP URL points to the controller, NOT the mixing server.

A few moments later, the first participant calls in using a SIP INVITE (3). The call is routed to the controller. It checks the conference ID. It finds that the policy permits up to two participants (not a practical example, but simplifies the call flow). It stores data indicating that one participant has now joined, and the proxies the INVITE request in step (4) to the mixer. The request URI in this request will have the same user part as (3), but the host part now represents the mixer. The mixer receives the INVITE, creates the initial conference state (as this is the first call for that URL), and returns a 200 OK (5), which is forward to the caller (6), and then ACKed (7 and 8).

In step (9), the second caller calls in. The controller sees that only one participant is on the call so far, so the second call is accepted. The controller stores the fact that there are now 2 participants, and proxies the INVITE (10). The INVITE is accepted by the mixer (11), and the response forwarded to the second caller (12), and then ACKed (13 and 14). The two participants A and B can now hear each other.

A third caller then calls in (15). The controller checks its records, and notices that this conference is now full. So, it rejects the INVITE (16), which is acknowleged (17).

The astute reader will observe that, strictly speaking, the HTTP server does not really need to be co-resident with the SIP server in the controller. The initial conference setup can be stored in a database by a web server, and the controller can simply read this database. However, in more complex cases, we may wish to have web access to learn dynamic information about the conference as it progresses (for example, which users are in the conference). For this kind of dynamic session state, using a shared database between components is cumbersome. Rather, an integrated HTTP/SIP server is much better suited, where integrated implies only that it has built in mechanisms for session state sharing between the SIP and HTTP components.

For this simple conferencing service, it was sufficient for the controller to act as a proxy. Thats because it does not need to forcibly kick anyone out of the conference once they are in. To support that kind of functionality, third party call control is needed. Let us examine a more complex service in the next section.

6.2.2 Web Scheduled, IVR supported, Time Limited Conference

In this more complex example, we once again wish to use a web interface to set up the conferences. However, we wish to add a stop time. If there are participants in the conference when the stop time

```
   |   |   |   | (1) HTTP POST  |                      |
   |--------------------------->|                      |
   |   |   |   | (2) 200 OK     |                      |
   |<---------------------------|                      |
   |   |   |   |                |                      |
   |   |   |   | (3) INVITE     |                      |
   |   |----------------------->|  (4) INVITE          |
   |   |   |   |                |--------------------->|
   |   |   |   |                |   (5) 200 OK         |
   |   |   |   |  (6) 200 OK    |<---------------------|
   |   |<-----------------------|                      |
   |   |   |   | (7) ACK        |                      |
   |   |----------------------->|   (8) ACK            |
   |   |   |   |                |--------------------->|
   |   |   |   |                |                      |
   |   |   |   | (9) INVITE     |                      |
   |   |   |------------------->|   (10) INVITE        |
   |   |   |   |                |--------------------->|
   |   |   |   |                |   (11) 200 OK        |
   |   |   |   | (12) 200 OK    |<---------------------|
   |   |   |<-------------------|                      |
   |   |   |   |  (13) ACK      |                      |
   |   |   |------------------->| (14) ACK             |
   |   |   |   |                |--------------------->|
   |   |   |   |                |                      |
   |   |   |   | (15) INVITE    |                      |
   |   |   |   |--------------->|                      |
   |   |   |   |(16) 500 Full   |                      |
   |   |   |   |<---------------|                      |
   |   |   |   |(17) ACK        |                      |
   |   |   |   |--------------->|                      |
   |   |   |   |                |                      |
   |   |   |   |                |                      |
   |   |   |   |                |                      |

  Web  A   B   C              Controller            Mixer
```

   Figure 6: Web Scheduled Conference Services arrives, a warning announcement is played 10 minutes prior, and then they are kicked off. In addition, when a user joins the conference, before they are added, they hear an announcement that states the name of the person that set up the conference, and what the start and stop times are. They are then asked to speak their name. Then, they are dropped in. The conference server then speaks their name, so that everyone knows who just joined.

This seemingly complex service is very easily constructed by adding an IVR server as described above. Now, we have a controller, a mixing server, and an IVR server, all working together to build the service. Each provides a specific component towards the overall solution, yet each is an application server in its own right, with both signaling and media interfaces.

We assume that the web setup is done as above. This time, the stop time is provided, along with the name of the person setting up the conference.

The call flow for the initial participant is shown in Figure 7.

The initial participant sends an INVITE, which is forwarded to the controller. The controller matches the request URI against the conference that the user wishes to join. The controller recognizes that it needs to play an announcement. So, in step (2), it initiates a call to an IVR server. This call is accepted in step (3), and the resulting SDP is passed back to the UAC in step (4) in a provisional response. After ACKing the call with the IVR in step (5), the controller receives an HTTP GET to fetch the root VoiceXML script in step (6). The controller dynamically generates the VoiceXML script, whose content will cause the server to read out "Welcome to the conference, Bob. The call will start at 10 am, and end at 11am.". The name of the caller, Bob, is extracted from the INVITE (1).

Once the prompt has been played, the IVR server prompts the caller for their name, and the result is recorded into a file. Then, the VoiceXML server attempts to fetch the next VoiceXML script from the controller (8). Before responding, the controller reconnects the media stream from the media server into the conference bridge. To do this, it first sends an INVITE to the conferencing server, using SDP indicating send only (9). The server accepts (10), and the controller ACKs (11). The SDP from the acceptance (10) is passed in a re-INVITE (12) to the IVR server. The IVR server then accepts (13) and the controller ACKs (14). Now, a unidirectional media stream from the IVR server into the conference bridge is set up. The controller returns the next VoiceXML script (15), which tells the IVR server to play the previously recorded file into the conference, announcing the joining

user. Once this is done, the IVR server fetches the next script (16), and gets back an empty response (17). The controller then disconnects from the IVR server (18,19). Finally, the controller re-INVITEs the conference server (20), updating the SDP to be that from the initial INVITE (1). The SDP from the acceptance (21) is passed on to the caller (22). Now, the caller is connected to the mixer as the first user in the conference.

The second user would join in much the same way.

Approximately 10 minutes before the end of the conference, a timer fires inside of the controller. It is time to play a warning announcement into the conference. The call flow for this is shown in Figure 8.

The basic idea is to initiate a call to the IVR server and mixer, connect them using third party call control, and then have the IVR server play the announcement into the conference. The controller then hangs up.

In step (1), the controller sends an INVITE to the mixer with a single audio stream on hold (i.e., "empty"). The request URI of the request is that of the conference. The mixer returns a 200 OK in step (2), and an ACK is sent in (3). The SDP from (2) is then used in step (4) to call the IVR server, which answers with its SDP in step (5). This is used in a re-invite (7,8,9) to the mixer to update the IP address and port as that of the IVR server. The IVR server then fetches the root VoiceXML document from the controller (11). This document instructs the server to read out some kind of conference warning - "Warning, your conference will end in 10 minutes". Once this is done, the IVR server fetches the next document (13), which is empty. The controller then hangs up with both the mixer (17) and the IVR server (19), disconnecting the IVR server from the conference.

These examples demonstrate the component model we are proposing. The mixing component does not have application level intelligence. It has a call control interface, allowing it to exist anywhere (and be provided by any ASP service) and yet be a callable resource by other application server components. By combining a controller with an IVR server and the mixing server, complex and useful applications can be constructed in a distributed fashion.

6.3 Continuous Text-to-Speech

Another example of an application server component is a continuous Text-to-Speech (TTS) converter. This kind of service allows a real time text stream (encapsulated in RTP using the RTP payload format

```
  Caller          Controller         IVR Server          Mixing Server
    |               |                  |                   |
    | (1) INVITE    |                  |                   |
    |-------------->| (2) INVITE       |                   |
    |               |----------------->|                   |
    |               | (3) 200 OK       |                   |
    | (4) 183       |<-----------------|                   |
    |<--------------|                  |                   |
    |               | (5) ACK          |                   |
    |               |----------------->|                   |
    |               | (6) HTTP GET     |                   |
    |               |<.................|                   |
    |               | (7) 200 OK       |                   |
    |               |.................>|                   |
    |               |                  |                   |
    |               | (8) HTTP GET     |                   |
    |               |<.................|                   |
    |               | (9) INVITE       |                   |
    |               |------------------------------------->|
    |               | (10) 200 OK      |                   |
    |               |<-------------------------------------|
    |               | (11) ACK         |                   |
    |               |------------------------------------->|
    |               | (12) INVITE      |                   |
    |               |----------------->|                   |
    |               | (13) 200 OK      |                   |
    |               |<-----------------|                   |
    |               | (14) ACK         |                   |
    |               |----------------->|                   |
    |               |                  |                   |
    |               | (15) 200 OK      |                   |
    |               |.................>|                   |
    |               | (16) HTTP GET    |                   |
    |               |<.................|                   |
    |               | (17) 200 OK      |                   |
    |               |.................>|                   |
    |               | (18) BYE         |                   |
    |               |----------------->|                   |
    |               | (19) 200 OK      |                   |
    |               |<-----------------|                   |
    |               | (20) INVITE      |                   |
    |               |------------------------------------->|
    |               | (21) 200 OK      |                   |
    | (22) 200 OK   |<-------------------------------------|
    |<--------------|                  |                   |
    | (23) ACK      |                  |                   |
    |-------------->| (24) ACK         |                   |
    |               |------------------------------------->|
    |               |                  |                   |
    |               |                  |                   |
    |               |                  |                   |

  Caller          Controller         IVR Server          Mixing Server
```

```
            | (1) INVITE empty SDP  |                          |
            |---------------------->|                          |
            | (2) 200 OK SDP A      |                          |
            |<----------------------|                          |
            | (3) ACK               |                          |
            |---------------------->|                          |
            |                       |   (4) INV SDP A          |
            |------------------------------------------------->|
            | (5) 200 OK SDP B      |                          |
            |<-------------------------------------------------|
            |                       |   (6) ACK                |
            |------------------------------------------------->|
            | (7) INV SDP B         |                          |
            |---------------------->|                          |
            | (8) 200 OK SDP A      |                          |
            |<----------------------|                          |
            | (9) ACK               |                          |
            |---------------------->|                          |
            |                       |  (11) HTTP GET           |
            |<-------------------------------------------------|
            |                       |  (12) 200 OK             |
            |------------------------------------------------->|
            |                       |                          |
            |                       |                          |
            |                       |  (13) HTTP GET           |
            |<-------------------------------------------------|
            |                       |  (14) 200 OK             |
            |------------------------------------------------->|
            |                       |                          |
            | (15) BYE              |                          |
            |------------------------------------------------->|
            |                       |  (16) 200 OK             |
            |<-------------------------------------------------|
            | (17) BYE              |                          |
            |---------------------->|                          |
            | (18) 200 OK           |                          |
            |<----------------------|                          |
            |                       |                          |
            |                       |                          |
   Controller               Mixer                      IVR Server
```

Figure 8: Advanced Web Scheduled Conference Service: Warning

Announcement

for text [14] to be received, which is then converted to speech and returned as an audio stream encoded using a traditional speech codec, be it G.723.1, G.711, or what have you.

Like the IVR server and mixing server, the TTS server acts as a user agent server. It answers incoming calls, and basically mirrors incoming text back as speech. It continutes to do so until the call is hung up by the initiating client.

A TTS service can be done using VoiceXML with an IVR server, as in the examples above. However, the difference is that here, the text stream to be converted is in the data path, not the control path. The stream is likely to be generated by other entities in the system, not the controller.

6.3.1 Service Interface

It is likely that the text-to-speech conversation process differs significantly depending on the language. As such, separate URIs SHOULD be used for language specific TTS services. Specifically, the convention sip:<server-specific-name>-<language-tag>@<domain> is RECOMMENDED. The language tags SHOULD be selected from the set defined in RFC1766 [15].

One of the unfortunate limitations of SDP is that it is not currently possible for a single media stream to be composed of separate media formats in each direction. The text over RTP stream is, in fact, based on the top level text MIME type (text/t140). As a result, two media streams are needed for this service - a unidirectional audio stream and a unidirectional text stream.

First, the client INVITEs the server. The SDP MUST indicate a two media streams. One stream MUST be of type audio. It SHOULD contain the set of audio codecs acceptable to the client. The stream MUST be marked as recv-only. The other stream MUST be of type text. It MUST contain a single codec, which is a dynamic payload number bound to text/t140. The stream MUST be marked as send-only. The 200 OK response from the TTS server that accepts the call has SDP with a two media lines, one of type audio, and one of type text, in the same order the streams appeared in the INVITE, as mandated by RFC2543. The audio stream SHOULD contain a subset of the codecs listed in the audio stream in the INVITE. The audio stream MUST be marked as send-only. The text stream MUST contain a single codec, which is a dynamic payload type number bound to text/t140. The stream MUST be marked as receive-only.

The client then ACKs the request. The TTS server SHOULD attempt to convert all text received on the incoming text stream to speech, and return the resulting speech on the outgoing audio stream.

6.3.2 Hearing Impaired Service

The TTS server is extremely useful in supporting hearing impaired services. Examples of such services are described in [16]. Specifically, Section 2.4 describes a service where a controller accesses a TTS service.

6.4 Messaging Servers

Another type of application server component is a messaging server. Messaging servers allow for callers to record audio messages for users on the system. Users can also call into the server to retrieve these messages, delete them, and file them. The system operates through the use of voice prompts combined with DTMF detection and/or speech recognition. The prompts that are played are context dependent. A messaging server can be viewed as a specialized version of an IVR server with an application specific controller associated with it. In fact, a messaging server can be implemented in this way exactly. However, the combination is also usefully viewed as a component in its own right, due to the frequent need for messaging components in more complex applications.

6.4.1 Service Interface

The service interface for communicating with a messaging server is described in detail in [7]. The interface provides well known URIs for the most common resources within a messaging server - user specific message drops with a variety of drop conditions (called party busy, called party not there, etc.), message retrievals using a variety of authentication mechanisms (PIN, SIP level authentication), and message drops that are not user specific, so that the target user is queried for as part of the interface.

6.4.2 Web Enabled Message Drops

An example usage of this application component is a web front end that allows users to leave voicemail for company employees through the company web page. The page has a URL for each company employee. If some user A clicks on a URL for employee B, A's phone rings. When A picks up, they hear a greeting to record a message for employee B.

The call flow for this application is the combination of third party call control combined with access to the service. It is shown in Figure 9.

```
     |      |              |                      |
     |      | (1) HTTP GET |                      |
     |-------------------->|                      |
     |      | (2) 200 OK   |                      |
     |<--------------------|                      |
     |      | (3) INV      |                      |
     |      |<-------------|                      |
     |      | (4) 200 OK   |                      |
     |      |------------->|                      |
     |      | (5) ACK      |                      |
     |      |<-------------|                      |
     |      |              |  (6) INV             |
     |      |              |--------------------->|
     |      |              |  (7) 200 OK          |
     |      |              |<---------------------|
     |      |              |  (8) ACK             |
     |      |              |--------------------->|
     |      | (9) INV      |                      |
     |      |<-------------|                      |
     |      | (10) 200 OK  |                      |
     |      |------------->|                      |
     |      | (11) ACK     |                      |
     |      |<-------------|                      |
     |      |              |                      |
     |      |              |                      |
     |      |              |                      |

    Web    SIP           Controller             Messaging
      Caller                                     Server
```

Figure 9: Web Enabled Message Drops

The caller, from a web page, clicks on the URL for the user they wish

to leave a message for. The result is an HTTP request (1) to the controller. The URI in this request would be some controller-specific identifier that tells the controller what it needs to do. The controller then calls the user (3) using an SDP with a single media stream on hold initially. This is accepted (4), and the resulting SDP is used in an INVITE to the messaging server (6). The URI of this INVITE is that for message drop with standard greeting (sip:sub-jdrosen-deposit@voiceserver.com). The call is accepted (7) and the 200 OK is used in a re-INVITE to the caller (9) to set the address of the media stream to that of the voicemail server. After the call is accepted (10) and ACKed (11), the caller hears the voice drop prompt for the messaging server, and can record their message.

7 Security Considerations

In many cases, authorization may need to be made to allow a caller access to a session level resource. Traditional SIP level authentication mechanisms can be used to accomplish this. Note, however, that in many cases the caller is the controller, which is acting as a third party call controller. In these cases, a two level trust model is really needed. The trust relationship in such situations is really between the session level resource and the controller (perhaps through an explicit business arrangement), and then between the controller and the caller. Thus, controllers should authenticate themselves to session resources they contact, rather than trying to proxy credentials from the caller.

8 Conclusion

In this paper, we have argued that rapid deployment of complex communications applications will require a distributed model where application components are spread across the network. These components could be offered by separate providers, for example, enabling an ASP component model to evolve. We have observed that many of the components can be described as having some kind of session level resource that can be communicated with, usually in an automated fashion. Access to these resources is typically parameterized. As a result, SIP access, using the request URI as a service indicator, is an ideal way to communicate across these components.

To validate this model, we examined the specific service interfaces that would be defined by IVR servers, conferencing servers, text-to-speech servers and messaging servers. We gave call flows of complex applications built up from these components using the specified interfaces.

9 Changes from -00

o Minor edits

10 Author's Addresses

Jonathan Rosenberg
dynamicsoft
72 Eagle Rock Avenue
First Floor
East Hanover, NJ 07936
email: jdrosen@dynamicsoft.com

Peter Mataga
dynamicsoft
72 Eagle Rock Avenue
First Floor
East Hanover, NJ 07936
email: pmataga@dynamicsoft.com

Henning Schulzrinne
Columbia University
M/S 0401
1214 Amsterdam Ave.
New York, NY 10027-7003
email: schulzrinne@cs.columbia.edu

11 Bibliography

[1] N. Greene, M. Ramalho, and B. Rosen, "Media gateway control protocol architecture and requirements," Request for Comments 2805, Internet Engineering Task Force, Apr. 2000.

[2] M. Arango, A. Dugan, I. Elliott, C. Huitema, and S. Pickett, "Media gateway control protocol (MGCP) version 1.0," Request for Comments 2705, Internet Engineering Task Force, Oct. 1999.

[3] F. Cuervo, N. Greene, C. Huitema, A. Rayhan, B. Rosen, and J. Segers, "Megaco protocol 0.8," Request for Comments 2885, Internet Engineering Task Force, Aug. 2000.

[4] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: session initiation protocol," Request for Comments 2543, Internet Engineering Task Force, Mar. 1999.

[5] J. Rosenberg, H. Schulzrinne, and J. Peterson, "Third party call control in SIP," Internet Draft, Internet Engineering Task Force, Mar. 2000. Work in progress.

[6] M. Handley and V. Jacobson, "SDP: session description protocol," Request for Comments 2327, Internet Engineering Task Force, Apr. 1998.

[7] B. Campbell and R. Sparks, "Control of service context using SIP Request-URI," Internet Draft, Internet Engineering Task Force, Oct. 2000. Work in progress.

[8] H. Schulzrinne and S. Petrack, "RTP payload for DTMF digits, telephony tones and telephony signals," Request for Comments 2833, Internet Engineering Task Force, May 2000.

[9] V. Bharatia, E. Cave, and B. Culpepper, "SIP INFO method for event reporting," Internet Draft, Internet Engineering Task Force, Apr. 2000. Work in progress.

[10] T. Choudhuri, C. Haun, P. Sollee, S. Orton, and S. Whynot, "SIP INFO method for DTMF digit transport and collection," Internet Draft, Internet Engineering Task Force, Apr. 2000. Work in progress.

[11] VoiceXML Forum, "Voice extensible markup language (voicexml) version 1.00," voicexml forum specification, VoiceXML Forum, Mar. 2000.

[12] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session initiation protocol," Internet Draft, Internet Engineering Task Force, Aug. 2000. Work in progress.

[13] S. Donovan, "The SIP INFO method," Request for Comments 2976, Internet Engineering Task Force, Oct. 2000.

[14] G. Hellstrom, "RTP payload for text conversation," Request for Comments 2793, Internet Engineering Task Force, May 2000.

[15] H. Alvestrand, "Tags for the identification of languages," Request for Comments 1766, Internet Engineering Task Force, Mar. 1995.

[16] J. Rosenberg, H. Schulzrinne, and H. Sinnreich, "Sip enabled services to support the hearing impaired," Internet Draft, Internet Engineering Task Force, July 2000. Work in progress.

Table of Contents