SIPPING J. Rosenberg
Internet-Draft dynamicsoft
Expires: December 29, 2003 June 30, 2003

A Framework and Requirements for Application Interaction in the Session Initiation Protocol (SIP)
draft-rosenberg-sipping-app-interaction-framework-01

Status of this Memo

Copyright Notice

Abstract

This document describes a framework and requirements for the interaction between users and Session Initiation Protocol (SIP) based applications. By interacting with applications, users can guide the way in which they operate. The focus of this framework is stimulus signaling, which allows a user agent to interact with an application without knowledge of the semantics of that application. Stimulus signaling can occur to a user interface running locally with the client, or to a remote user interface, through media streams. Stimulus signaling encompasses a wide range of mechanisms, ranging from clicking on hyperlinks, to pressing buttons, to traditional Dual Tone Multi Frequency (DTMF) input. In all cases, stimulus signaling is supported through the use of markup languages, which play a key

role in this framework.

Table of Contents

1. Introduction 3
2. Definitions 4
3. A Model for Application Interaction 7
3.1 Function vs. Stimulus 8
3.2 Real-Time vs. Non-Real Time 9
3.3 Client-Local vs. Client-Remote 9
3.4 Interaction Scenarios on Telephones 10
3.4.1 Client Remote 11
3.4.2 Client Local 11
3.4.3 Flip-Flop 11
4. Framework Overview 13
5. Client Local Interfaces 15
5.1 Discovering Capabilities 15
5.2 Pushing an Initial Interface Component 15
5.3 Updating an Interface Component 17
5.4 Terminating an Interface Component 18
6. Client Remote Interfaces 19
6.1 Originating and Terminating Applications 19
6.2 Intermediary Applications 19
7. Inter-Application Feature Interaction 21
7.1 Client Local UI 21
7.2 Client-Remote UI 22
8. Intra Application Feature Interaction 23
9. Examples 24
10. Security Considerations 25
11. Contributors 26
Informative References 27
Author's Address 28
Intellectual Property and Copyright Statements 29

1. Introduction

The Session Initiation Protocol (SIP) [1] provides the ability for users to initiate, manage, and terminate communications sessions. Frequently, these sessions will involve a SIP application. A SIP application is defined as a program running on a SIP-based element (such as a proxy or user agent) that provides some value-added function to a user or system administrator. Examples of SIP applications include pre-paid calling card calls, conferencing, and presence-based [3] call routing.

In order for most applications to properly function, they need input from the user to guide their operation. As an example, a pre-paid calling card application requires the user to input their calling card number, their PIN code, and the destination number they wish to reach. The process by which a user provides input to an application is called "application interaction".

Application interaction can be either functional or stimulus. Functional interaction requires the user agent to understand the semantics of the application, whereas stimulus interaction does not. Stimulus signaling allows for applications to be built without requiring modifications to the client. Stimulus interaction is the subject of this framework. The framework provides a model for how users interact with applications through user interfaces, and how user interfaces and applications can be distributed throughout a network. This model is then used to describe how applications can instantiate and manage user interfaces.

2. Definitions

SIP Application: A SIP application is defined as a program running on a SIP-based element (such as a proxy or user agent) that provides some value-added function to a user or system administrator. Examples of SIP applications include pre-paid calling card calls, conferencing, and presence-based [3] call routing.

Application Interaction: The process by which a user provides input to an application.

Real-Time Application Interaction: Application interaction that takes place while an application instance is executing. For example, when a user enters their PIN number into a pre-paid calling card application, this is real-time application interaction.

Non-Real Time Application Interaction: Application interaction that takes place asynchronously with the execution of the application. Generally, non-real time application interaction is accomplished through provisioning.

Functional Application Interaction: Application interaction is functional when the user device has an understanding of the semantics of the application that the user is interacting with.

Stimulus Application Interaction: Application interaction is considered to be stimulus when the user device has no understanding of the semantics of the application that the user is interacting with.

User Interface (UI): The user interface provides the user with context in order to make decisions about what they want. The user enters information into the user interface. The user interface interprets the information, and passes it to the application.

User Interface Component: A piece of user interface which operates independently of other pieces of the user interface. For example, a user might have two separate web interfaces to a pre-paid calling card application - one for hanging up and making another call, and another for entering the username and PIN.

User Device: The software or hardware system that the user directly interacts with in order to communicate with the application. An example of a user device is a telephone. Another example is a PC with a web browser.

User Input: The "raw" information passed from a user to a user interface. Examples of user input include a spoken word or a click on a hyperlink.

Client-Local User Interface: A user interface which is co-resident with the user device.

Client Remote User Interface: A user interface which executes remotely from the user device. In this case, a standardized interface is needed between them. Typically, this is done through media sessions - audio, video, or application sharing.

Media Interaction: A means of separating a user and a user interface by connecting them with media streams.

Interactive Voice Response (IVR): An IVR is a type of user interface that allows users to speak commands to the application, and hear responses to those commands prompting for more information.

Prompt-and-Collect: The basic primitive of an IVR user interface. The user is presented with a voice option, and the user speaks their choice.

Barge-In: In an IVR user interface, a user is prompted to enter some information. With some prompts, the user may enter the requested information before the prompt completes. In that case, the prompt ceases. The act of entering the information before completion of the prompt is referred to as barge-in.

Focus: A user interface component has focus when user input is provided fed to it, as opposed to any other user interface components. This is not to be confused with the term focus within the SIP conferencing framework, which refers to the center user agent in a conference [4].

Focus Determination: The process by which the user device determines which user interface component will receive the user input.

Focusless User Interface: A user interface which has no ability to perform focus determination. An example of a focusless user interface is a keypad on a telephone.

Feature Interaction: A class of problems which result when multiple applications or application components are trying to provide services to a user at the same time.

Inter-Application Feature Interaction: Feature interactions that occur between applications.

DTMF: Dual-Tone Multi-Frequency. DTMF refer to a class of tones generated by circuit switched telephony devices when the user presses a key on the keypad. As a result, DTMF and keypad input are often used synonymously, when in fact one of them (DTMF) is merely a means of conveying the other (the keypad input) to a client-remote user interface (the switch, for example).

Application Instance: A single execution path of a SIP application.

Originating Application: A SIP application which acts as a UAC, calling the user.

Terminating Application: A SIP application which acts as a UAS, answering a call generated by a user. IVR applications are terminating applications.

Intermediary Application: A SIP application which is neither the caller or callee, but rather, a third party involved in a call.

3. A Model for Application Interaction

```
         +---+            +---+            +---+             +---+
         |   |            |   |            |   |             |   |
         |   |            | U |            | U |             | A |
         |   |   Input    | s |   Input    | s |   Results   | p |
         |   | ---------> | e | ---------> | e | ----------> | p |
         | U |            | r |            | r |             | l |
         | s |            |   |            |   |             | i |
         | e |            | D |            | I |             | c |
         | r |   Output   | e |   Output   | f |   Update    | a |
         |   | <--------- | v | <--------- | a | <.......... | t |
         |   |            | i |            | c |             | i |
         |   |            | c |            | e |             | o |
         |   |            | e |            |   |             | n |
         |   |            |   |            |   |             |   |
         +---+            +---+            +---+             +---+
```

Figure 1: Model for Real-Time Interactions

Figure 1 presents a general model for how users interact with applications. Generally, users interact with a user interface through a user device. A user device can be a telephone, or it can be a PC with a web browser. Its role is to pass the user input from the user, to the user interface. The user interface provides the user with context in order to make decisions about what they want. The user enters information into the user interface. The user interface interprets the information, and passes it to the application. The application may be able to modify the user interface based on this information. Whether or not this is possible depends on the type of user interface.

User interfaces are fundamentally about rendering and interpretation. Rendering refers to the way in which the user is provided context. This can be through hyperlinks, images, sounds, videos, text, and so on. Interpretation refers to the way in which the user interface takes the "raw" data provided by the user, and returns the result to the application in a meaningful format, abstracted from the particulars of the user interface. As an example, consider a pre-paid calling card application. The user interface worries about details such as what prompt the user is provided, whether the voice is male or female, and so on. It is concerned with recognizing the speech that the user provides, in order to obtain the desired information. In this case, the desired information is the calling card number, the PIN code, and the destination number. The application needs that data, and it doesn't matter to the application whether it was collected using a male prompt or a female one.

User interfaces generally have real-time requirements towards the user. That is, when a user interacts with the user interface, the user interface needs to react quickly, and that change needs to be propagated to the user right away. However, the interface between the user interface and the application need not be that fast. Faster is better, but the user interface itself can frequently compensate for long latencies there. In the case of a pre-paid calling card application, when the user is prompted to enter their PIN, the prompt should generally stop immediately once the first digit of the PIN is entered. This is referred to as barge-in. After the user-interface collects the rest of the PIN, it can tell the user to "please wait while processing". The PIN can then be gradually transmitted to the application. In this example, the user interface has compensated for a slow UI to application interface by asking the user to wait.

The separation between user interface and application is absolutely fundamental to the entire framework provided in this document. Its importance cannot be overstated.

With this basic model, we can begin to taxonomize the types of systems that can be built.

3.1 Function vs. Stimulus

The first way to taxonomize the system is to consider the interface between the UI and the application. There are two fundamentally different models for this interface. In a functional interface, the user interface has detailed knowledge about the application, and is, in fact, specific to the application. The interface between the two components is through a functional protocol, capable of representing the semantics which can be exposed through the user interface. Because the user interface has knowledge of the application, it can be optimally designed for that application. As a result, functional user interfaces are almost always the most user friendly, the fastest, the and the most responsive. However, in order to allow interoperability between user devices and applications, the details of the functional protocols need to be specified in standards. This slows down innovation and limits the scope of applications that can be built.

An alternative is a stimulus interface. In a stimulus interface, the user interface is generic, totally ignorant of the details of the application. Indeed, the application may pass instructions to the user interface describing how it should operate. The user interface translates user input into "stimulus" - which are data understood only by the application, and not by the user interface. Because they are generic, and because they require communications with the application in order to change the way in which they render

information to the user, stimulus user interfaces are usually slower, less user friendly, and less responsive than a functional counterpart. However, they allow for substantial innovation in applications, since no standardization activity is needed to built a new application, as long as it can interact with the user within the confines of the user interface mechanism.

In SIP systems, functional interfaces are provided by extending the SIP protocol to provide the needed functionality. For example, the SIP caller preferences specification [5] provides a functional interface that allows a user to request applications to route the call to specific types of user agents. Functional interfaces are important, but are not the subject of this framework. The primary goal of this framework is to address the role of stimulus interfaces to SIP applications.

3.2 Real-Time vs. Non-Real Time

Application interaction systems can also be real-time or non-real-time. Non-real interaction allows the user to enter information about application operation in asynchronously with its invocation. Frequently, this is done through provisioning systems. As an example, a user can set up the forwarding number for a call-forward on no-answer application using a web page. Real-time interaction requires the user to interact with the application at the time of its invocation.

3.3 Client-Local vs. Client-Remote

Another axis in the taxonomization is whether the user interface is co-resident with the user device (which we refer to as a client-local user interface), or the user interface runs in a host separated from the client (which we refer to as a client-remote user interface). In a client-remote user interface, there exists some kind of protocol between the client device and the UI that allows the client to interact with the user interface over a network.

The most important way to separate the UI and the client device is through media interaction. In media interaction, the interface between the user and the user interface is through media - audio, video, messaging, and so on. This is the classic mode of operation for VoiceXML [2], where the user interface (also referred to as the voice browser) runs on a platform in the network. Users communicate with the voice browser through the telephone network (or using a SIP session). The voice browser interacts with the application using HTTP to convey the information collected from the user.

We refer to the second sub-case as a client-local user interface. In

this case, the user interface runs co-located with the user. The interface between them is through the software that interprets the users input and passes them to the user interface. The classic example of this is the web. In the web, the user interface is a web browser, and the interface is defined by the HTML document that it's rendering. The user interacts directly with the user interface running in the browser. The results of that user interface are sent to the application (running on the web server) using HTTP.

It is important to note that whether or not the user interface is local, or remote (in the case of media interaction), is not a property of the modality of the interface, but rather a property of the system. As an example, it is possible for a web-based user interface to be provided with a client-remote user interface. In such a scenario, video and application sharing media sessions can be used between the user and the user interface. The user interface, still guided by HTML, now runs "in the network", remote from the client. Similarly, a VoiceXML document can be interpreted locally by a client device, with no media streams at all. Indeed, the VoiceXML document can be rendered using text, rather than media, with no impact on the interface between the user interface and the application.

It is also important to note that systems can be hybrid. In a hybrid user interface, some aspects of it (usually those associated with a particular modality) run locally, and others run remotely.

3.4 Interaction Scenarios on Telephones

This same model can apply to a telephone. In a traditional telephone, the user interface consists of a 12-key keypad, a speaker, and a microphone. Indeed, from here forward, the term "telephone" is used to represent any device that meets, at a minimum, the characteristics described in the previous sentence. Circuit-switched telephony applications are almost universally client-remote user interfaces. In the Public Switched Telephone Network (PSTN), there is usually a circuit interface between the user and the user interface. The user input from the keypad is conveyed used Dual-Tone Multi-Frequency (DTMF), and the microphone input as PCM encoded voice.

In an IP-based system, there is more variability in how the system can be instantiated. Both client-remote and client-local user interfaces to a telephone can be provided.

In this framework, a PSTN gateway can be considered a "user proxy". It is a proxy for the user because it can provide, to a user interface on an IP network, input taken from a user on a circuit switched telephone. The gateway may be able to run a client-local user interface, just as an IP telephone might.

3.4.1 Client Remote

The most obvious instantiation is the "classic" circuit-switched telephony model. In that model, the user interface runs remotely from the client. The interface between the user and the user interface is through media, set up by SIP and carried over the Real Time Transport Protocol (RTP) [6]. The microphone input can be carried using any suitable voice encoding algorithm. The keypad input can be conveyed in one of two ways. The first is to convert the keypad input to DTMF, and then convey that DTMF using a suitance encoding algorithm for it (such as PCMU). An alternative, and generally the preferred approach, is to transmit the keypad input using RFC 2833 [7], which provides an encoding mechanism for carrying keypad input within RTP.

In this classic model, the user interface would run on a server in the IP network. It would perform speech recognition and DTMF recognition to derive the user intent, feed them through the user interface, and provide the result to an application.

3.4.2 Client Local

An alternative model is for the entire user interface to reside on the telephone. The user interface can be a VoiceXML browser, running speech recognition on the microphone input, and feeding the keypad input directly into the script. As discussed above, the VoiceXML script could be rendered using text instead of voice, if the telephone had a textual display.

3.4.3 Flip-Flop

A middle-ground approach is to flip back and forth between a client-local and client-remote user interface. Many voice applications are of the type which listen to the media stream and wait for some specific trigger that kicks off a more complex user interaction. The long pound in a pre-paid calling card application is one example. Another example is a conference recording application, where the user can press a key at some point in the call to begin recording. When the key is pressed, the user hears a whisper to inform them that recording has started.

The ideal way to support such an application is to install a client-local user interface component that waits for the trigger to kick off the real interaction. Once the trigger is received, the application connects the user to a client-remote user interface that can play announements, collect more information, and so on.

The benefit of flip-flopping between a client-local and client-remote user interface is cost. The client-local user interface will

eliminate the need to send media streams into the network just to wait for the user to press the pound key on the keypad.

The Keypad Markup Language (KPML) was designed to support exactly this kind of need [8]. It models the keypad on a phone, and allows an application to be informed when any sequence of keys have been pressed. However, KPML has no presentation component. Since user interfaces generally require a response to user input, the presentation will need to be done using a client-remote user interface that gets instantiated as a result of the trigger.

It is tempting to use a hybrid model, where a prompt-and-collect application is implemented by using a client-remote user interface that plays the prompts, and a client-local user interface, described by KPML, that collects digits. However, this only complicates the application. Firstly, the keypad input will be sent to both the media stream and the KPML user interface. This requires the application to sort out which user inputs are duplicates, a process that is very complicated. Secondly, the primary benefit of KPML is to avoid having a media stream towards a user interface. However, there is already a media stream for the prompting, so there is no real savings.

4. Framework Overview

In this framework, we use the term "SIP application" to refer to a broad set of functionality. A SIP application is a program running on a SIP-based element (such as a proxy or user agent) that provides some value-added function to a user or system administrator. SIP applications can execute on behalf of a caller, a called party, or a multitude of users at once.

Each application has a number of instances that are executing at any given time. An instance represents a single execution path for an application. Each instance has a well defined lifecycle. It is established as a result of some event. That event can be a SIP event, such as the reception of a SIP INVITE request, or it can be a non-SIP event, such as a web form post or even a timer. Application instances also have a specific end time. Some instances have a lifetime that is coupled with a SIP transaction or dialog. For example, a proxy application might begin when an INVITE arrives, and terminate when the call is answered. Other applications have a lifetime that spans multiple dialogs or transactions. For example, a conferencing application instance may exist so long as there are any dialogs connected to it. When the last dialog terminates, the application instance terminates. Other applications have a liftime that is completely decoupled from SIP events.

It is fundamental to the framework described here that multiple application instances may interact with a user during a single SIP transaction or dialog. Each instance may be for the same application, or different applications. Each of the applications may be completely independent, in that they may be owned by different providers, and may not be aware of each others existence. Similarly, there may be application instances interacting with the caller, and instances interacting with the callee, both within the same transaction or dialog.

The first step in the interaction with the user is to instantiate one of more user interface components for the application instance. A user interface component is a single piece of the user interface that is defined by a logical flow that is not synchronously coupled with any other component. In other words, each component runs more or less independently.

A user interface component can be instantiated in one of the user devices (for a client-local user interface), or within a network element (for a client-remote user interface). If a client-local user interface is to be used, the application needs to determine whether or not the user device is capable of supporting a client-local user interface, and in what format. In this framework, all client-local

user interface components are described by a markup language. A markup language describes a logical flow of presentation of information to the user, collection of information from the user, and transmission of that information to an application. Examples of markup languages include HTML, WML, VoiceXML, the Keypad Markup Language (KPML) [8] and the Media Server Control Markup Language (MSCML) [9].

The interface between the user interface component and the application is typically markup-language specific. For those markups which support rendering of information to a user, such as HTML, HTTP form POST operations are used. For those markups where no information is rendered to the user, the markup can play one of two roles. The first is called "one shot". In the one-shot role, the markup waits for a user to enter some information, and when they do, reports this event to the application. The application then does something, and the markup is no longer used. In the other modality, called "monitor", the markup stays permanently resident, and reports information back to an application continuously. However, the act of reporting information back to the application does not cause the installation of a new markup. In markups where one-shot or monitor modalities are used, a SIP MESSAGE request is used to report the status.

To create a client-local user interface, the application passes the markup document (or a reference to it) in a SIP message to that client. The SIP message can be one explicitly generated by the application (in which case the application has to be a UA or B2BUA), or it can be placed in a SIP message that passes by (in which case the application can be running in a proxy).

Client local user interface components are always associated with the dialog that the SIP message itself is associated with. Consequently, user interface components cannot be placed in messages that are not associated with a dialog.

If a user interface component is to be instantiated in the network, there is no need to determine the capabilities of the device on which the user interface is instantiated. Presumably, it is on a device on which the application knows a UI can be created. However, the application does need to connect the user device to the user interface. This will require manipulation of media streams in order to establish that connection.

Once a user interface component is created, the application needs to be able to change it, and to remove it. Finally, more advanced applications may require coupling between application components. The framework supports rudimentary capabilities there.

5. Client Local Interfaces

One key component of this framework is support for client local user interfaces.

5.1 Discovering Capabilities

A client local user interface can only be instantiated on a client if the user device has the capabilities needed to do so. Specifically, an application needs to know what markup languages, if any, are supported by the client. For example, does the client support HTML? VoiceXML? However, that information is not sufficient to determine if a client local user interface can be instantiated. In order to instantiate the user interface, the application needs to transfer the markup document to the client. There are two ways in which the markup document can be transferred. The application can send the client a URI which the client can use to fetch the markup, or the markup can be sent inline within the message. The application needs to know which of these modes are supported, and in the case of indirection, which URI schemes are supported to obtain the indirection.

Many applications will need to know these capabilities at the time an application instance is first created. Since applications can be created through SIP requests or responses, SIP needs to provide a means to convey this information. This introduces several concrete requirements for SIP:

REQ 1: A SIP request or response must be capable of conveying the set of markup languages supported by the UA that generated the request or response.

REQ 2: A SIP request or response must be capable of indicating whether a UA can obtain markups inline, or through an indirection. In the case of indirection, the UA must be capable of indicating what URI schemes it supports.

5.2 Pushing an Initial Interface Component

Once the application has determined that the UA is capable of supporting client local user interfaces, the next step is for the application to push an interface component to the user device.

Generally, we anticipate that interface components will need to be created at various different points in a SIP session. Clearly, they will need to be pushed during an initial INVITE, in both responses (so as to place a component into the calling UA) and in the request (so as to place a component into the called UA). As an example, a

conference recording application allows the users to record the media for the session at any time. The application would like to push an HTML user interface component to both the caller and callee at the time the call is setup, allowing either to record the session. The HTML component would have buttons to start and stop recording. To push the HTML component to the caller, it needs to be pushed in the 200 OK (and possibly provisional response), and to push it to the callee, in the INVITE itself.

To state the requirement more concretely:

REQ 3: An application must be able to add a reference to, or an inline version of, a user interface component into any request or response that passes through or is emanated from that application.

However, there will also be cases where the application needs to push a new interface component to a UA, but it is not as a result of any SIP message. As an example, a pre-paid calling card application will set a timer that determines how long the call can proceed, given the availability of funds in the user's account. When the timer fires, the application would like to push a new interface component to the calling UA, allowing them to click to add more funds.

In this case, there is no message already in transit that can be used as a vehicle for pushing a user interface component. This requires that applications can generate their own messages to push a new component to a UA:

REQ 4: A UA application must be able to send a SIP message to the UA at the other end of the dialog, asking it to create a new interface component.

In all cases, the information passed from the application to the UA must include more than just the interface component itself (or a reference to it). The user must be able to decide whether or not it wants to proceed with this application. To make that determination, the user must have information about the application. Specifically, it will need the name of the application, and an identifier of the owner or administrator for the application. As an example, a typical name would be "Prepaid Calling Card" and the owner could be "voiceprovider.com".

REQ 5: Any user interface component passed to a client (either inline or through a reference) must also include markup meta-data, including a human readable name of the application, and an identifier of the owner of the application.

Clearly, there are security implications. The user will need to

verify the identity of the application owner, and be sure that the user interface component is not being replayed, that is, it actually belongs with this specific SIP message.

REQ 6: It must be possible for the client to validate the authenticity and integrity of the markup document (or its reference) and its associated meta-data. It must be possible for the client to verify that the information has not been replayed from a previous SIP message.

If the user decides not to execute the user interface component, it simply discards it. There is no explicit requirement for the user to be able to inform the application that the component was discarded. Effectively, the application will think that the component was executed, but that the user never entered any information.

5.3 Updating an Interface Component

Once a user interface component has been created on a client, it can be updated in two ways. The first way is the "normal" path inherent to that component. The client enters some data, the user interface transfers the information to the application (typically through HTTP), and the result of that transfer brings a new markup document describing an updated interface. This is referred to as a synchronous update, since it is synchronized with user interaction.

However, synchronous updates are not sufficient for many applications. Frequently, the interface will need to be updated asynchronously by the application, without an explicit user action. A good example of this is, once again, the pre-paid calling card application. The application might like to update the user interface when the timer runs out on the call. This introduces several requirements:

REQ 7: It must be possible for an application to asynchronously push an update to an existing user interface component, either in a message that was already in transit, or by generating a new message.

REQ 8: It must be possible for the client to associate the new interface component with the one that it is supposed to replace, so that the old one can be removed.

Unfortunately, pushing of application components introduces a race condition. What if the user enters data into the old component, causing an HTTP request to the application, while an update of that component is in progress? The client will get an interface component in the HTTP response, and also get the new one in the SIP message.

Which one does the client use? There needs to be a way in which to properly order the components:

REQ 9: It must be possible for the client to relatively order user interface updates it receives as the result of synchronous and asynchronous messaging.

5.4 Terminating an Interface Component

User interface components have a well defined lifetime. They are created when the component is first pushed to the client. User interface components are always associated with the SIP dialog on which they were pushed. As such, their lifetime is bound by the lifetime of the dialog. When the dialog ends, so does the interface component.

This rule applies to early dialogs as well. If a user interface component is passed in a provisional response to INVITE, and a separate branch eventually answers the call, the component terminates with the arrival of the 2xx. That's because the early dialog itself terminates with the arrival of the 2xx.

However, there are some cases where the application would like to terminate the user interface component before its natural termination point. To do this, the application pushes a "null" update to the client. This is an update that replaces the existing user interface component with nothing.

REQ 10: It must be possible for an application to terminate a user interface component before its natural expiration.

The user can also terminate the user interface component. However, there is no explicit signaling required in this case. The component is simply dismissed. To the application, it appears as if the user has simply ceased entering data.

6. Client Remote Interfaces

As an alternative to, or in conjunction with client local user interfaces, an application can make use of client remote user interfaces. These user interfaces can execute co-resident with the application itself (in which case no standardized interfaces between the UI and the application need to be used), or it can run separately. This framework assumes that the user interface runs on a host that has a sufficient trust relationship with the application. As such, the means for instantiating the user interface is not considered here.

The primary issue is to connect the user device to the remote user interface. Doing so requires the manipulation of media streams between the client and the user interface. Such manipulation can only be done by user agents. There are two types of user agent applications within this framework - originating/terminating applications, and intermediary applications.

6.1 Originating and Terminating Applications

Originating and terminating applications are applications which are themselves the originator or the final recipient of a SIP invitation. They are "pure" user agent applications - not back-to-back user agents. The classic example of such an application is an interactive voice response (IVR) application, which is typically a terminating application. Its a terminating application because the user explicitly calls it; i.e., it is the actual called party. An example of an originating application is a wakeup call application, which calls a user at a specified time in order to wake them up.

Because originating and terminating applications are a natural termination point of the dialog, manipulation of the media session by the application is trivial. Traditional SIP techniques for adding and removing media streams, modifying codecs, and changing the address of the recipient of the media streams, can be applied. Similarly, the application can direclty authenticate itself to the user through S/MIME, since it is the peer UA in the dialog.

6.2 Intermediary Applications

Intermediary application are, at the same time, more common than originating/terminating applications, and more complex. Intermediary applications are applications that are neither the actual caller or called party. Rather, they represent a "third party" that wishes to interact with the user. The classic example is the ubiquitous pre-paid calling card application.

In order for the intermediary application to add a client remote user interface, it needs to manipulate the media streams of the user agent to terminate on that user interface. This also introduces a fundamental feature interaction issue. Since the intermediary application is not an actual participant in the call, how does the user interact with the intermediary application, and its actual peer in the dialog, at the same time? This is discussed in more detail in Section 7.

7. Inter-Application Feature Interaction

The inter-application feature interaction problem is inherent to stimulus signaling. Whenever there are multiple applications, there are multiple user interfaces. When the user provides an input, to which user interface is the input destined? That question is the essence of the inter-application feature interaction problem.

Inter-application feature interaction is not an easy problem to resolve. For now, we consider separately the issues for client-local and client-remote user interface components.

7.1 Client Local UI

When the user interface itself resides locally on the client device, the feature interaction problem is actually much simpler. The end device knows explicitly about each application, and therefore can present the user with each one separately. When the user provides input, the client device can determine to which user interface the input is destined. The user interface to which input is destined is referred to as the application in focus, and the means by which the focused application is selected is called focus determination.

Generally speaking, focus determination is purely a local operation. In the PC universe, focus determination is provided by window managers. Each application does not know about focus, it merely receives the user input that has been targeted to it when its in focus. This basic concept applies to SIP-based applications as well.

Focus determination will frequently be trivial, depending on the user interface type. Consider a user that makes a call from a PC. The call passes through a pre-paid calling card application, and a call recording application. Both of these wish to interact with the user. Both push an HTML-based user interface to the user. On the PC, each user interface would appear as a separate window. The user interacts with the call recording application by selecting its window, and with the pre-paid calling card application by selecting its window. Focus determination is literally provided by the PC window manager. It is clear to which application the user input is targeted.

As another example, consider the same two applications, but on a "smart phone" that has a set of buttons, and next to each button, an LCD display that can provide the user with an option. This user interface can be represented using the Wireless Markup Language (WML).

The phone would allocate some number of buttons to each application. The prepaid calling card would get one button for its "hangup"

command, and the recording application would get one for its "start/stop" command. The user can easily determine which application to interact with by pressing the appropriate button. Pressing a button determines focus and provides user input, both at the same time.

Unfortunately, not all devices will have these advanced displays. A PSTN gateway, or a basic IP telephone, may only have a 12-key keypad. The user interfaces for these devices are provided through the Keypad Markup Language (KPML). Considering once again the feature interaction case above, the pre-paid calling card application and the call recording application would both pass a KPML document to the device. When the user presses a button on the keypad, to which document does the input apply? The user interface does not allow the user to select. A user interface where the user cannot provide focus is called a focusless user interface. This is quite a hard problem to solve. This framework does not make any explicit normative recommendation, but concludes that the best option is to send the input to both user interfaces unless the markup in one interface has indicated that it should be suppressed from others. This is a sensible choice by analogy - its exactly what the existing circuit switched telephone network will do. It is an explicit non-goal to provide a better mechanism for feature interaction resolution than the PSTN on devices which have the same user interface as they do on the PSTN. Devices with better displays, such as PCs or screen phones, can benefit from the capabilities of this framework, allowing the user to determine which application they are interacting with.

Indeed, when a user provides input on a focusless device, the input must be passed to all client local user interfaces, AND all client remote user interfaces, unless the markup tells the UI to suppress the media. In the case of KPML, key events are passed to remote user interfaces by encoding them in RFC 2833 [7]. Of course, since a client cannot determine if a media stream terminates in a remote user interface or not, these key events are passed in all audio media streams unless the "Q" digit is used to suppress.

7.2 Client-Remote UI

When the user interfaces run remotely, the determination of focus can be much, much harder. There are many architectures that can be deployed to handle the interaction. None are ideal. However, all are beyond the scope of this specification.

8. Intra Application Feature Interaction

An application can instantiate a multiplicity of user interface components. For example, a single application can instantiate two separate HTML components and one WML component. Furthermore, an application can instantiate both client local and client remote user interfaces.

The feature interaction issues between these components within the same application are less severe. If an application has multiple client user interface components, their interaction is resolved identically to the inter-application case - through focus determination. However, the problems in focusless user interfaces (such as a keypad) generally won't exist, since the application can generate user interfaces which do not overlap in their usage of an input.

The real issue is that the optimal user experience frequently requires some kind of coupling between the differing user interface components. This is a classic problem in multi-modal user interfaces, such as those described by Speech Application Language Tags (SALT). As an example, consider a user interface where a user can either press a labeled button to make a selection, or listen to a prompt, and speak the desired selection. Ideally, when the user presses the button, the prompt should cease immediately, since both of them were targeted at collecting the same information in parallel. Such interactions are best handled by markups which natively support such interactions, such as SALT, and thus require no explicit support from this framework.

9. Examples

TODO.

10. Security Considerations

There are many security considerations associated with this framework. It allows applications in the network to instantiate user interface components on a client device. Such instantiations need to be from authenticated applications, and also need to be authorized to place a UI into the client. Indeed, the stronger requirement is authorization. It is not so important to know that name of the provider of the application, but rather, that the provider is authorized to instantiate components.

Generally, an application should be considered authorized if it was an application that was legitimately part of the call setup path. With this definition, authorization can be enforced using the sips URI scheme when the call is initiated.

11. Contributors

This document was produced as a result of discussions amongst the application interaction design team. All members of this team contributed significantly to the ideas embodied in this document. The members of this team were:

Eric Burger
Cullen Jennings
Robert Fairlie-Cuninghame

Informative References

[1] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[2] McGlashan, S., Lucas, B., Porter, B., Rehor, K., Burnett, D., Carter, J., Ferrans, J. and A. Hunt, "Voice Extensible Markup Language (VoiceXML) Version 2.0", W3C CR CR-voicexml20-20030220, February 2003.

[3] Day, M., Rosenberg, J. and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, February 2000.

[4] Rosenberg, J., "A Framework for Conferencing with the Session Initiation Protocol", draft-ietf-sipping-conferencing-framework-00 (work in progress), May 2003.

[5] Rosenberg, J., Schulzrinne, H. and P. Kyzivat, "Caller Preferences and Callee Capabilities for the Session Initiation Protocol (SIP)", draft-ietf-sip-callerprefs-08 (work in progress), March 2003.

[6] Schulzrinne, H., Casner, S., Frederick, R. and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", RFC 1889, January 1996.

[7] Schulzrinne, H. and S. Petrack, "RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals", RFC 2833, May 2000.

[8] Burger, E., "Keypad Markup Language (KPML)", draft-burger-sipping-kpml-02 (work in progress), July 2003.

[9] Dyke, J., Burger, E. and A. Spitzer, "Media Server Control Markup Language (MSCML) and Protocol", draft-vandyke-mscml-02 (work in progress), July 2003.

Author's Address

Jonathan Rosenberg
dynamicsoft
600 Lanidex Plaza
Parsippany, NJ 07054
US

Phone: +1 973 952-5000
EMail: jdrosen@dynamicsoft.com
URI: http://www.jdrosen.net

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.