aculab

ACULAB.COM

Aculab SIP programmer's guide

Revision 6.7.2

MAN1763

PROPRIETARY INFORMATION

Document Revision

Rev	Date	By	Detail
1.0.0	31.11.05	DJL	Initial release
1.0.1	20.02.06	DJL	Typological changes only
1.1.0	06.07.06	SP	Updates to support call redirect
1.2.0	15.11.06	SP	Updates to include STUN, out of dialog and SRTP media attributes. Re-worked the function populate_media_answer() to allow for multiple media streams
1.2.1	07.10.10	EBJ	Updated to corporate fonts
1.2.2	20.10.10	DF	Changed document name
1.2.3	16/12/10	NC	Fixed link to example code
1.2.4	05/01/11	EBJ	Hyperlinks removed
1.2.5	27/05/11	SWB	STUN references removed
1.2.6	05/12/11	NC	Updated document properties
6.7.0	10/03/17	NMC	Added Forking
6.7.1	19/05/22	KGB	Added enable_midcall_response_mask
6.7.2	13/03/24	ACP	Format only

CONTENTS

1 Introduction

The purpose of this guide is to illustrate to the application writer some typical SIP usage scenarios and how these may be implemented with the Aculab Generic call control and Extended SIP APIs. In addition to generic call control, it is now possible using the extended SIP API, to provide the following new features:

- Third party call control; the ability to redirect media streams between different end-points and retain the call control
- Fine control of SDP content, facilitating multiple media streams, non-RTP payload types, third party call control
- Selective presentation of raw SIP messages, enabling applications to be more “SIP aware” and have greater choice when responding to SIP requests
- Custom headers and message bodies in the call setup INVITE, adding greater flexibility to call setups and providing a means of passing PSTN information over IP calls
- Mid call signalling with custom headers and bodies, providing a means to send PSTN information
- Unlimited sizes of data can be passed, and received as details, by the API; `char*` types and linked lists are used to represent types whose size is undefined

The above features may be utilised in many different ways. However, third party call control seems to be particularly useful in the call centre domain; whereas custom SIP entities and mid call signalling may be used to enhance an existing gateway application.

This guide may be used by existing and new developers alike: existing developers will find the document useful to assist porting, and therefore enhancing, a gateway application; new developers will discover how to exploit fine control over SIP/SDP message to realise new opportunities for IP telephony deployments.

NOTE

Please refer to the sip_send_message and EV_DETAILS sections of the Extended SIP API document for details of which SIP messages can be sent and received. Additional functionality can be built in at a later date, please contact Aculab support to discuss your needs.

To assist readability this guide has been divided into groups of scenarios, depicting functionality, which an application may wish to implement:

- Setups and clears
- Supplementary services
- Mid call signalling
- Third party call control
- Out of dialog messages
- Miscellany

In each group of scenarios, individual usages are identified and described in detail. The content of usage descriptions comprising a textual overview, a schematic of the SIP messages and API calls and, for reference, some snippets of example code.

Despite this document containing some very descriptive SIP message flows, an application writer is advised to refer to one or more of the following documents, where relevant, to reinforce their knowledge of a given application area:

RFC 2327 – Session Description Protocol (SDP)
RFC 3665 – Basic call flow examples
RFC 3666 – SIP/PSTN call flows
RFC 3264 – Offer/Answer model with SDP
RFC 3725 – Third party call control best practices
RFC 3515 – The REFER method
RFC 3204 – MIME media types for QSIG/ISUP
RFC 2976 – INFO method
RFC 3891 – Replaces header
Draft-ietf-sipping-service-examples-09 – Hold and transfer.

The documents may be found at the following sites:

http://www.ietf.org/html.charters/sip-charter.html

http://www.ietf.org/html.charters/sipping-charter.html

NOTE

The above documents assume some familiarity with RFC 3261 SIP: Session Initiation Protocol.

1.1 Scope

This guide is intended to be of use in the development of applications which implement features provided by the functionality of the Extended SIP API. It may also be of value to the application writer who relies solely on Generic call control mechanisms, but wishes to gain further insight to the workings of SIP.

It should be noted at this stage, that a call control application, which uses SIP via the Generic Call control API, will not encounter the media events associated with session negotiation or be able to realise some of the features illustrated in this guide. The Generic call control API architecture and the “Aculab Media Handler Plugin” conceal these aspects of the underlying protocol exchange.

1.2 Background to the Extended SIP API

The Extended SIP API has been developed in order to facilitate the implementation of applications, which require SIP specific functionality not available in a generic telephony API. The Extended SIP API assists in the production of those applications by providing greater access to the contents of the SIP messages (and to protocol messages embedded therein, for example SDP, MIME) than that offered by the Generic call control API.

To facilitate faithful representation of the information contained within SIP messages no fixed size lists or buffers are used. Linked lists are used for collections of types and char* buffers (in preference to fixed size char arrays) for strings are extensively used. The developer porting an existing application to the Extended API should be aware that the addressing and the display name fields within the `sip_openout` call employ char*, rather than `char[MAX_SIZE]` types found in the `call_openout`.

Dynamically sized data types whether they be character pointers or linked lists, typically employ the allocation of free store (heap) memory using C library calls such as malloc. The programmer, hence, must be aware that such memory should be “freed” when no longer required, otherwise memory starvation will occur eventually resulting in system failure.

This guide provides helper functions to assist the application writer in correctly allocating memory necessary to store received SIP/SDP information structures and complementary routines to free this memory. The routines may be copied from either from this document or from the example code available on the Aculab website. Alternatively, the routines may merely be used as a starting point by the developer wishing to produce a specific solution. These helper functions are described in section Appendix A:

The Extended SIP API works well alongside the Generic call control API. To discover more about the API functions provided by the Extended SIP API and how it coexists with the Generic API, please refer to the “Extended SIP API Guide”.

2 SIP setups and clears

This section describes how to set up and clear down basic SIP calls. Reference is given to the accompanying example applications and to the commonly used C functions that are defined in the appendices. In addition to the example applications described in this document, there is one extra application, which will set up a call using the 3rd party call control techniques described herein. No more detail on this application will be given other than to note its name; `Controller.exe`.

The example code can be found in the Aculab installation directory under `Docs/SIPProgrammersGuide`

NOTE

Each of the example programs that are distributed with this guide display the current call flow on the screen. This display is very basic and if there are any errors (protocol or otherwise) it is highly unlikely that these flows will be correct when the application exits. They will only be correct when the call flow completes successfully. This applies to all subsequent sections of the guide.

2.1 Incoming call with no early media – SDP in the INVITE.

This example shows how to handle a basic incoming SIP call. The application will respond to an INVITE with the normal 100/180/200 sequence of responses and the example will show where the application should configure, start and stop the media resources. The application IncomingCalls with no command line options will run this example.

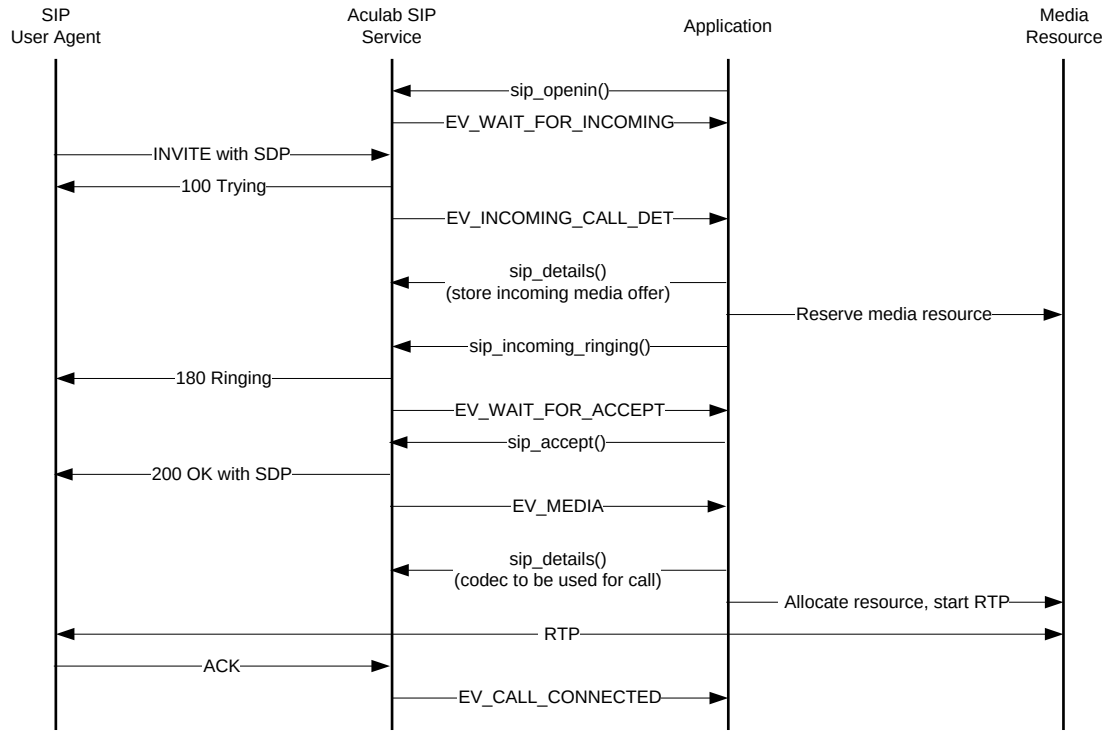

Figure 2-1 Incoming call – no early media – SDP in INVITE

Here the application must have at least one call handle open and waiting for SIP calls. When using the extended SIP API, this is achieved by calling `sip_openin` instead of `call_openin`. When a SIP User Agent sends an `INVITE`, an `EV_INCOMING_CALL_DET` will be raised to the application. Up until this point, there is no actual call in progress. Now that there is one, the application must reserve some media resources. In this example, the far end sends an `INVITE` containing several codecs. The application will simply choose the first supported codec in the list. At `EV_INCOMING_CALL_DET` the application will need to call `sip_details` and store the media offer that the `SIP_DETAIL_PARMS` structure contains. It is very important to note that `sip_free_details` must be called after every call to `sip_details`. If this is not done, the application will leak memory:

Alerting a call:

```
ACU_ERR handle_ev_incoming_call_det(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER** offer)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_INCOMING_RINGING_PARMS sip_incoming_ringing_parms;

   // Here it is necessary to call sip_details in order to determine what,
   // if there is one, is contained in the media offer.
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Copy the media offer. clone_media_offer is defined in section A.1
   *offer = clone_media_offer(sip_detail_parms.media_offer_answer);

   // At this stage a real application would need to reserve some media resource.
   // This will not be done here as these examples are concerned only with the
   // SIP signalling.

   // Free the SIP_DETAIL_PARMS structure. This is essential after any call to
   // sip_details. If this function is not called, the application will leak memory.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Alert the call
   INIT_ACU_CL_STRUCT(&sip_incoming_ringing_parms);
   sip_incoming_ringing_parms.handle = handle;

   // Send the 180 Ringing response to the UAC.
   rc = sip_incoming_ringing(&sip_incoming_ringing_parms);
   if(0 != rc)
   {
      printf("Error in sip_incoming_ringing(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // There is a large amount of dynamic memory allocation involved when creating
   // a media offer or answer. This memory must be freed. free_media_offer is defined
   // in section 0
   free_media_offer(&sip_incoming_ringing_parms.media_offer_answer);

   return 0;
}
```

Accepting a call

Now that the call has been alerted an EV_WAIT_FOR_ACCEPT will be raised. The application will need to decide which media parameters the call will use based on the media offer that was stored at EV_INCOMING_CALL_DET. It should then accept the call with a call to sip_accept using these parameters.

```
ACU_ERR handle_ev_wait_for_accept(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER* offer)
{
   ACU_ERR rc = 0;
   SIP_ACCEPT_PARMS accept;
   ACU_MEDIA_OFFER_ANSWER *temp_answer;

   INIT_ACU_STRUCT(&accept);
   accept.handle = handle;
   // There was an SDP body in the initial INVITE. An answer based on this
   // offer must be supplied here. populate_media_answer is defined in section B.2
   temp_answer = populate_media_answer(offer);

   accept.media_offer_answer = *temp_answer;
   // Send the 200 OK response.
   rc = sip_accept(&accept);
   if(0 != rc)
   {
      printf("Error in sip_accept(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free all dynamically assigned memory (0)
   free_media_offer(&temp_answer);
   free_media_offer(&offer);

   return 0;
}
```

The SIP service will now send a 200OK to the UAC containing an SDP body with the above media description. At this point, an EV_MEDIA will be raised to the application, which will then be able to extract all the media settings relevant to the call, then configure and start the media. When the ACK is received from the UAC, an EV_CALL_CONNECTED will be raised to the application. When the call clears down, the application will receive an EV_IDLE and at this point should stop and release any media resources and release the call handle. Clearing calls down is described in more detail in sections 2.11 & 2.12.

2.2 Outgoing call with no early media – SDP in the INVITE.

This example shows how to initiate a basic outgoing SIP call. It will indicate how the application should populate an `INVITE` request and show where the application should configure, start, and stop the media resources. The application `OutgoingCalls` with the following command line options will run this example:

```
OutgoingCalls r=<REMOTE IP ADDRESS>
```

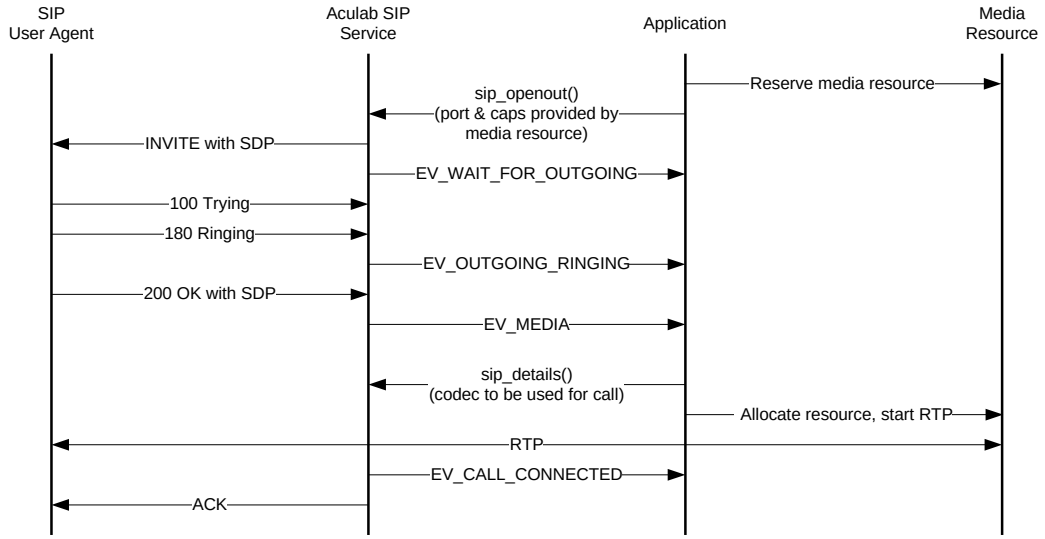

Figure 2-2 Outgoing call – no early media – SDP in INVITE

For outgoing calls with an SDP body, the application must supply a media offer when initiating the call with `sip_openout`. After that, the only thing it needs to do is configure the RTP stream when it receives an EV_MEDIA. Unlike the situation with incoming calls, however, the application must reserve some media resource before the call to sip_openout is made. An actual SIP call will be in progress at that stage and the application must know that there is resource available to provide the media:

Initiating an outgoing call

```
ACU_ERR open_outgoing(ACU_CALL_HANDLE *handle)
{
   ACU_ERR rc = 0;
   SIP_OUT_PARMS out;

   // When opening the SIP port for incoming calls using the SIP Bridge it is
   // necessary to use sip_openout instead of call_openout.
   INIT_ACU_CL_STRUCT(&out);
   // Populate the net and addressing fields
   out.net = settings.sip_port;
   out.originating_addr = (ACU_CHAR*)malloc(strlen(LOCAL_IP_ADDRESS) + 1);
   strcpy(out.originating_addr, LOCAL_IP_ADDRESS);
   out.destination_addr = (ACU_CHAR*)malloc(strlen (REMOTE_IP_ADDRESS) + 1);
   strcpy(out.destination_addr, REMOTE_IP_ADDRESS);

   // Before making an outbound call, a real application would need to reserve
   // some media resource.

   // An SDP body is required. The arguments to the function populate_media_offer (defined
   // in section B.1 will determine this SDP body.
   out.media_offer_answer = populate_media_offer(ACU_PCMA_PAYLOAD_NUMBER, 0, 0);

   // Send the INVITE to the UAS
   rc = sip_openout(&out);
   if(0 != rc)
   {
      printf("Error in sip_openout(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Store the call's handle to pass to the call control loop.
   *handle = out.handle;

   // There is a large amount of dynamic memory allocation involved when creating
   // a media offer or answer. This memory must be freed (0).
   free_media_offer(&out.media_offer_answer);

   return rc;
}
```

When the 200 OK is received from the UAS the SIP service will raise an `EV_MEDIA` to the application, which can then configure and start the media stream. The SIP service will also raise an `EV_CALL_CONNECTED`, which may be handled as the needs of the application demands.

2.3 Incoming call with early media – SDP in INVITE

This example shows how to handle an incoming SIP call with early media. This means that the RTP stream will be started when the call is alerted. The application will respond to an `INVITE` with the normal 100/180/200 sequence of responses but this time the 180 will contain an SDP body. The example will also show where the application should configure, start, and stop the media resources. The application `IncomingCalls` with the following command line options will run this example:

```
IncomingCalls early
```

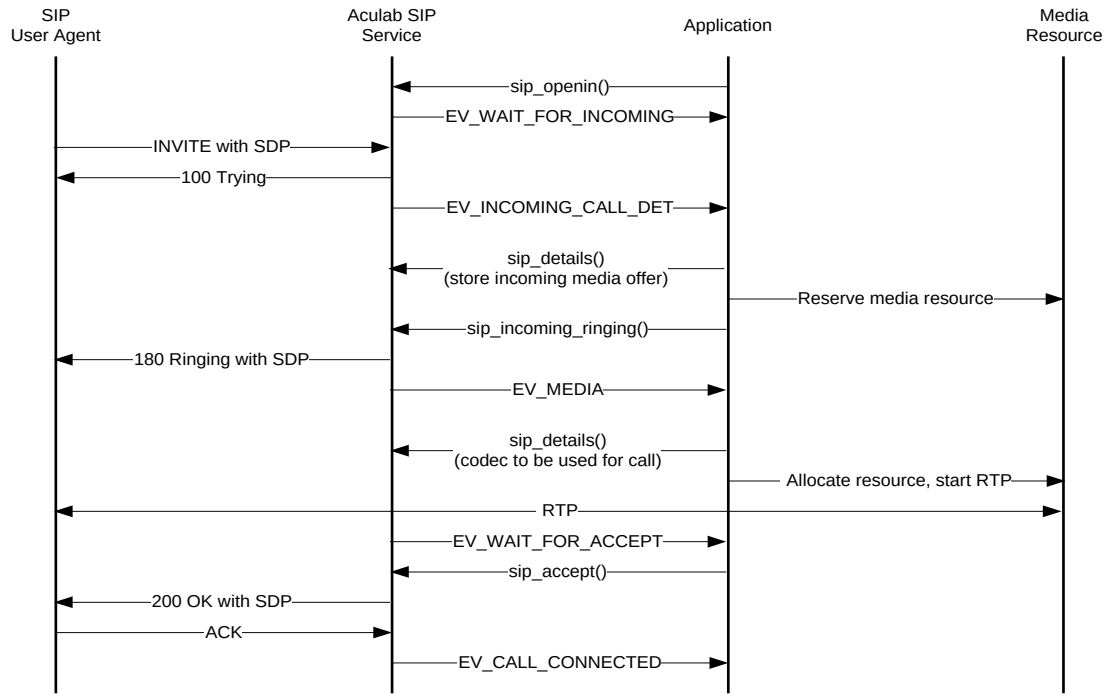

Figure 2-3 Incoming call – early media – SDP in INVITE

Here the application must have at least one call handle open and waiting for SIP calls. In this example, the far end sends an `INVITE` containing several codecs. The application will simply choose the first supported codec in the list. This time, however, the application will provide a media offer in the 180 Ringing response. This is generated by the application calling `sip_incoming_ringing` when it receives an `EV_INCOMING_CALL_DET`.

Alerting a call with early media

```
ACU_ERR handle_ev_incoming_call_det(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER** offer)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_INCOMING_RINGING_PARMS sip_incoming_ringing_parms;

   // Retrieve the call details
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Copy the media offer (A.1).
   *offer = clone_media_offer(sip_detail_parms.media_offer_answer);

   // At this stage a real application would need to reserve some media resource.

   // Free the SIP_DETAIL_PARMS structure. This is essential after any call to
   // sip_details. If this function is not called, the application will leak memory.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Alert the call
   INIT_ACU_CL_STRUCT(&sip_incoming_ringing_parms);
   sip_incoming_ringing_parms.handle = handle;

   // It is the UAS in an initial INVITE transaction that specifies whether
   // a call will utilise early media. An answer to the media offer given in
   // the INVITE must be specified here. i.e. the 180 Ringing response will
   // contain an SDP body.
   sip_incoming_ringing_parms.send_early_media = 1;
   sip_incoming_ringing_parms.media_offer_answer = populate_media_answer(*offer);
   free_media_offer(offer);
   // The 200 OK must contain the same SDP body as the 180 Ringing. This must
   // be stored for later use.
   *offer = clone_media_offer(sip_incoming_ringing_parms.media_offer_answer);

   // Send the 180 Ringing response to the UAC.
   rc = sip_incoming_ringing(&sip_incoming_ringing_parms);
   if(0 != rc)
   {
      printf("Error in sip_incoming_ringing(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }
```

```
    // There is a large amount of dynamic memory allocation involved when creating
    // a media offer or answer. This memory must be freed (0).
    free_media_offer(&sip_incoming_ringing_parms.media_offer_answer);

    return 0;
}
```

The 180 Ringing response will now be sent. It will contain an SDP body corresponding to the details set in the above function. An `EV_MEDIA` will now be raised to the application, which should then start the media stream. An `EV_WAIT_FOR_ACCEPT` will also be raised.

Accepting a call with early media

This is identical to accepting a call with no early media (section 0) except that the first supported codec has already been found before the call was alerted so there is no need to search for it again.

```
ACU_ERR handle_ev_wait_for_accept(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER* stored_answer)
{
   ACU_ERR rc = 0;
   SIP_ACCEPT_PARMS accept;
   ACU_MEDIA_OFFER_ANSWER *temp_answer;

   INIT_ACU_STRUCT(&accept);
   accept.handle = handle;

   // It is an early media call. The variable 'stored_answer' supplied to this function will contain the
   // answer given at EV_INCOMING_CALL_DET. The SDP body in the 200 OK must be the same.
   accept.media_offer_answer = *stored_answer;

   // Send the 200 OK response.
   rc = sip_accept(&accept);
   if(0 != rc)
   {
      printf("Error in sip_accept(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free all dynamically assigned memory
   free_media_offer(&stored_answer);

   return 0;
}
```

The 200 OK will now be sent to the UAC containing the same SDP body as the 180 Ringing. When the ACK is received from the UAC, an EV_CALL_CONNECTED will be raised to the application.

2.4 Outgoing call with early media – SDP in the INVITE.

This example shows how to handle an outgoing SIP call when the UAS establishes the media stream early. This is the same as for example 2.2 the only difference since the 180 (or possibly 183) response from the UAS will contain an SDP body. This simply means that the `EV_MEDIA` will be raised to the application earlier than in the previous example. All events should be handled in the same manner as before. This is reflected in the fact that the `OutgoingCalls` application will use the same command line as was used in the example where there was no early media:

```
OutgoingCalls r=<REMOTE IP ADDRESS>
```

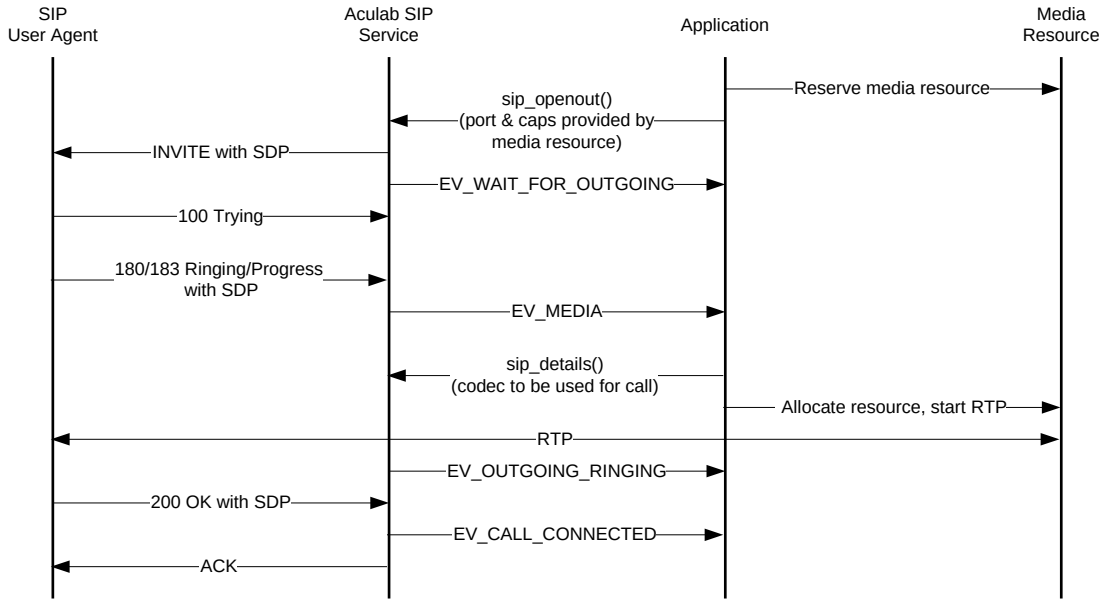

Figure 2-4 Outgoing call – early media – SDP in INVITE

2.5 Incoming call with no SDP in the INVITE

In this example, the incoming call will contain no media offer. The usual 100/180 sequence of messages will follow but now the 200 OK will need to contain a media offer of its own. The ACK from the UAC will then contain the answer and the media stream may be established in this manner. As such, this type of `INVITE` (containing no SDP body) is considered to be a request for an offer from the UAS. The application `IncomingCalls` with no command line options will run this example.

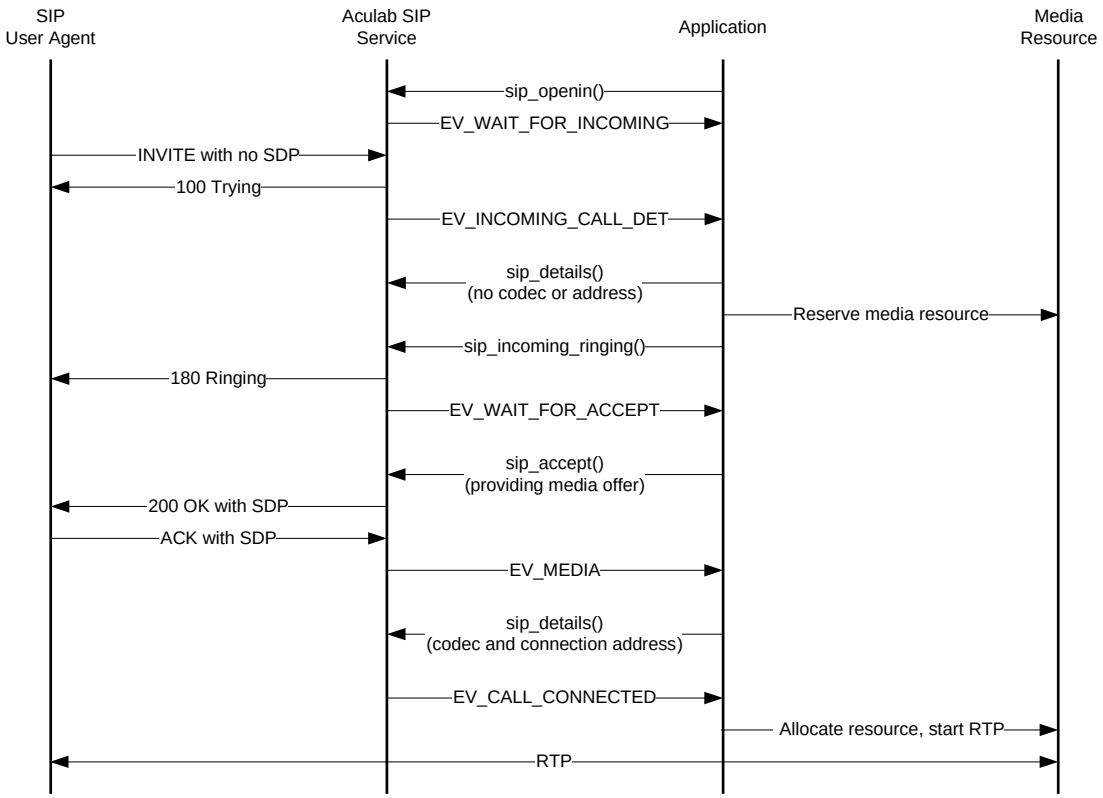

Figure 2-5 Incoming call – no SDP in INVITE

The same sequence of events will occur in this example as in the example 2.1. The handling of these events will be slightly different. At `EV_INCOMING_CALL_DET`, the application will need to determine whether there is any SDP in the `INVITE`. If there is not, it will need to make a not of this and provide an offer at `EV_WAIT_FOR_ACCEPT`:

Alerting a call with no SDP in the INVITE

```
ACU_ERR handle_ev_incoming_call_det(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER** offer)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_INCOMING_RINGING_PARMS sip_incoming_ringing_parms;

   // Here it is necessary to call sip_details in order to determine what,
   // if there is one, is contained in the media offer.
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
```

```
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Determine if there is any media offer
   if(NULL != sip_detail_parms.media_offer_answer)
   {
      // Copy the media offer. From here the flow would be identical to example 2.1
      *offer = clone_media_offer(sip_detail_parms.media_offer_answer);
   }
   else
   {
      // The received INVITE contains no media offer.
      g_no_sdp_in_invite = 1;
   }

   // At this stage a real application would need to reserve some media resource.
   // This will not be done here as these examples are concerned only with the
   // SIP signalling.

   // Free the SIP_DETAIL_PARMS structure. This is essential after any call to
   // sip_details. If this function is not called, the application will leak memory.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Alert the call
   INIT_ACU_CL_STRUCT(&sip_incoming_ringing_parms);
   sip_incoming_ringing_parms.handle = handle;
   // Send the 180 Ringing response to the UAC.
   rc = sip_incoming_ringing(&sip_incoming_ringing_parms);
   if(0 != rc)
   {
      printf("Error in sip_incoming_ringing(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // As usual, all dynamicallly allocated memory must be freed.
   free_media_offer(&sip_incoming_ringing_parms.media_offer_answer);

   return 0;
}
```

Now the application knows that it needs to make an offer of its own, it can do so at `EV_WAIT_FOR_ACCEPT`:

Accepting a call where no media offer was made in the initial INVITE

```
ACU_ERR handle_ev_wait_for_accept(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER* offer)
{
   ACU_ERR rc = 0;
   SIP_ACCEPT_PARMS accept;
   ACU_MEDIA_OFFER_ANSWER *temp_answer;

   INIT_ACU_STRUCT(&accept);
   accept.handle = handle;

   if(g_no_sdp_in_invite)
   {
      // There was no SDP body provided in the initial INVITE so it is up to
      // the UAS to provide one in the 200 OK. The UAC will provide an answer
      // in its ACK.
      temp_answer = populate_media_offer(ACU_PCMA_PAYLOAD_NUMBER, 0, 0);
   }
   else
   {
      // There was an SDP body in the initial INVITE. An answer based on this
      // offer must be supplied here.
      temp_answer = populate_media_answer(offer);
   }

   accept.media_offer_answer = *temp_answer;
   // Send the 200 OK response.
   rc = sip_accept(&accept);
   if(0 != rc)
   {
      printf("Error in sip_accept(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free all dynamically assigned memory
   free_media_offer(&temp_answer);
   free_media_offer(&offer);

   return 0;
}
```

The 200 OK will now be sent containing an SDP body. When the ACK is received containing the answer, EV_MEDIA (at which point the application should start the RTP stream) and then EV_CALL_CONNECTED will be raised.

2.6 Outgoing call with no SDP in the INVITE

In this example, the call is being made to a UAS as usual but it is not providing an SDP body in its `INVITE`. This is to be seen by the UAS as a request for a media offer which should then supply such an offer in its 200 OK. Once this offer has been made, the UAC will finally provide an SDP body (an answer to the UASs offer) in its `ACK`. The application `OutgoingCalls` with the following command line will run this example:

```
OutgoingCalls r=<REMOTE IP ADDRESS> nosdp
```

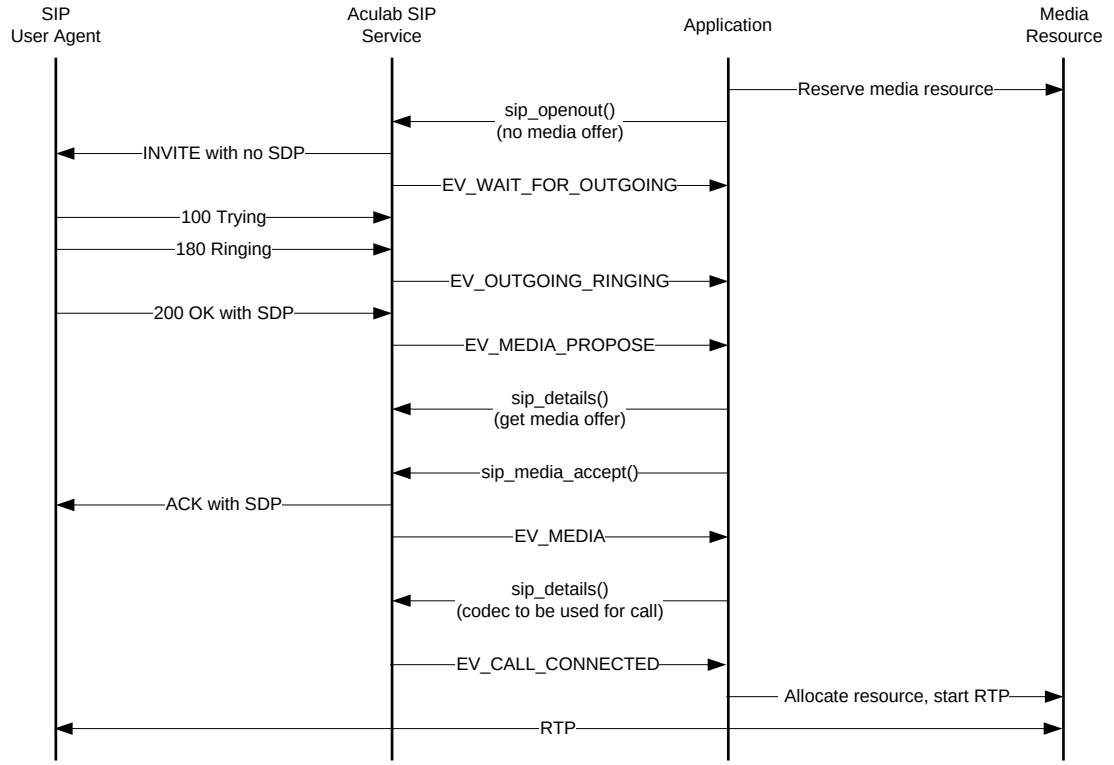

Figure 2-6 Outgoing call – no SDP in INVITE

The outgoing call will be made in exactly the same manner as in example 2.2 with the exception that the `sip_out_parms.media_offer_answer` field will `NULL`. The sequence of events is the same as in the earlier examples for the first few events. This time, however, an ACK will not automatically be sent on receipt of the 200 OK as only half of the offer/answer exchange will have taken place. Instead of this, an `EV_MEDIA_PROPOSE` will be raised to the application to indicate that it has received a media offer from the UAS. This will be handled in a way that is very similar to handling `EV_INCOMING_CALL_DET/EV_WAIT_FOR_ACCEPT` for an incoming call except that a new API call `sip_media_accept` will be used:

Accepting a media offer made by the UAS

```
ACU_ERR handle_ev_media_propose(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_MEDIA_ACCEPT_PARMS sip_media_accept_parms;
   ACU_MEDIA_OFFER_ANSWER* answer = NULL;

   // No SDP body was provided in the initial INVITE. This has prompted the UAS
   // to make its own media offer in the 200 OK.
```

```
   // Retrieve call details.
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Populate the SDP body of the response using the new media offer
   // as the starting point.
   answer = populate_media_answer(sip_detail_parms.media_offer_answer);
   if(NULL == answer)
   {
      // No supported codec has been offered. Reject call here.
   }

   INIT_ACU_CL_STRUCT(&sip_media_accept_parms);
   sip_media_accept_parms.handle = handle;
   sip_media_accept_parms.media_offer_answer = *answer;
   // Send the ACK containing an SDP body.
   rc = sip_media_accept(&sip_media_accept_parms);
   if(0 != rc)
   {
      printf("Error in sip_media_accept(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free the SIP_DETAIL_PARMS structure. This is essential after any call to
   // sip_details. If this function is not called, the application will leak memory.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free any dynamically assigned memory.
   free_media_offer(&answer);

   return 0;
}
```

2.7 Incoming call with black hole in SDP

Black hole SDP describes the situation where an INVITE is made with an SDP body but the connection address given in this body is 0.0.0.0. This is seen as an invitation to establish a one-way media stream from UAC to UAS. The UAS can only listen for RTP as it has not been provided with an address to send its media to. The application `IncomingCalls` with no command line options will run this example.

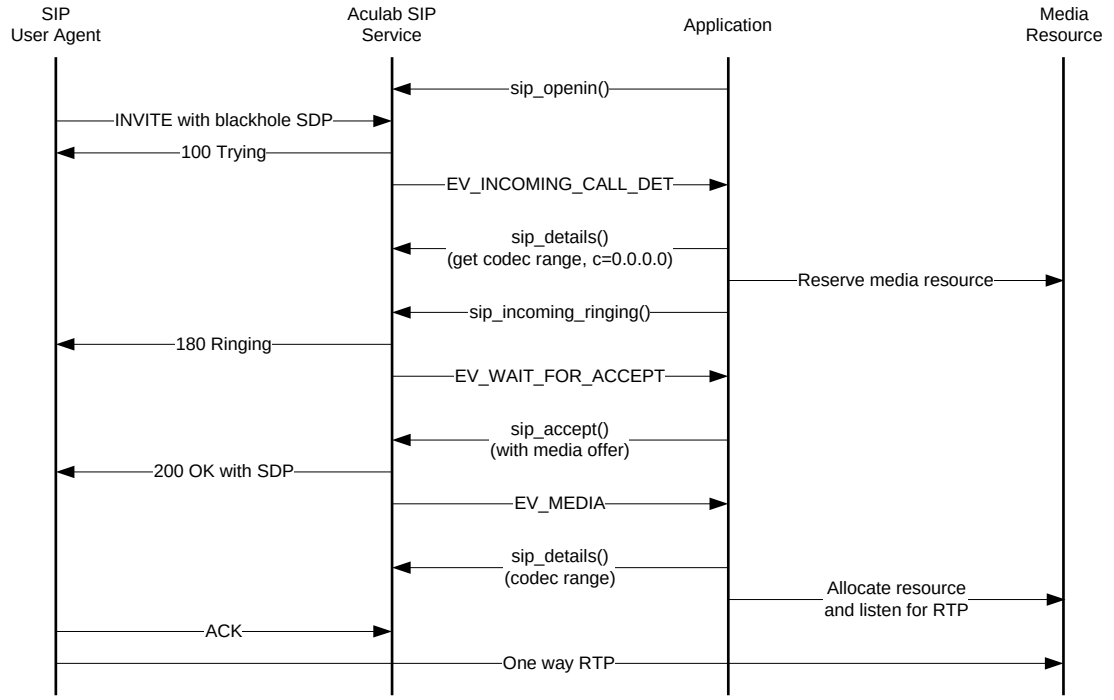

Figure 2-7 Incoming call – black hole in SDP

Handling incoming calls of this nature is identical to that in example 2.1. The only exception to this is that it must be aware that the UAC supplied no connection address. I.e. it should not start a media stream.

Alerting a call with black hole SDP

```
ACU_ERR handle_ev_incoming_call_det(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER** offer)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_INCOMING_RINGING_PARMS sip_incoming_ringing_parms;

   // Here it is necessary to call sip_details in order to determine what,
   // if there is one, is contained in the media offer.
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   if(NULL != sip_detail_parms.media_offer_answer)
   {
```

```
      // Copy the media offer.
      *offer = clone_media_offer(sip_detail_parms.media_offer_answer);
   }

   // There is a media offer. It is now necessary to determine the nature of the offer.
   if(!strcmp((*offer)->connection_address.address, BLACKHOLE_CONNECTION_ADDRESS))
   {
      // Here it is assumed that if the global connection address is black hole
      // then it the entire media offer is black hole. A real application would
      // need to process each media stream in this manner.
      g_black_hole = 1;
   }

   // At this stage a real application would need to reserve some media resource.
   // This will not be done here as these examples are concerned only with the
   // SIP signalling.

   // Free the SIP_DETAIL_PARMS structure. This is essential after any call to
   // sip_details. If this function is not called, the application will leak memory.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Alert the call
   INIT_ACU_CL_STRUCT(&sip_incoming_ringing_parms);
   sip_incoming_ringing_parms.handle = handle;

   // Send the 180 Ringing response to the UAC.
   rc = sip_incoming_ringing(&sip_incoming_ringing_parms);
   if(0 != rc)
   {
      printf("Error in sip_incoming_ringing(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free any dynamically allocated memory
   free_media_offer(&sip_incoming_ringing_parms.media_offer_answer);

   return 0;
}
```

Accepting the call is identical to example 2.1. When the `EV_MEDIA` is raised to indicate a successful offer/answer exchange, the global variable `g_black_hole` will indicate that the application need only listen for RTP.

2.8 Outgoing call with black hole in SDP

In this example, the outgoing INVITE will contain a standard (as in example 2.2) SDP body except the connection address will be set to 0.0.0.0. This is known as black hole SDP and indicates that the UAC intends to establish a one media stream to the UAS. It will not need to listen for RTP coming from the UAS. The application `OutgoingCalls` with the following command line will run this example:

```
OutgoingCalls r=<REMOTE IP ADDRESS> bh
```

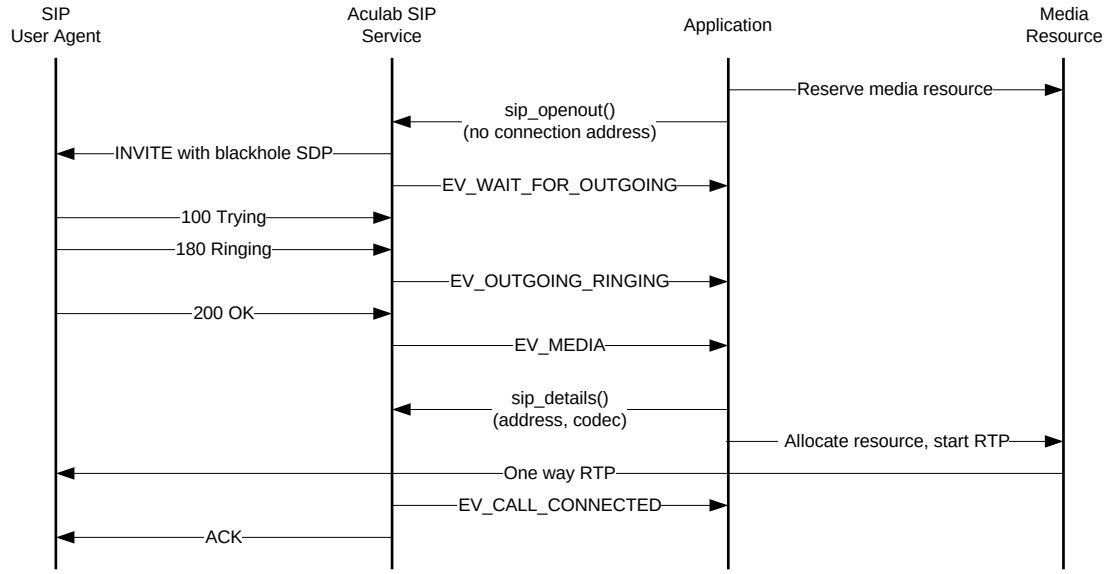

Figure 2-8 Outgoing call – black hole in SDP

The only difference to example 2.2 is that the field:

```
sip_out_parms.media_offer_answer->connection_address.address
```

will contain the string “0.0.0.0”. Every other field must still be supplied.

2.9 Incoming call with no media in SDP

Occasionally it is necessary to establish a session with no media stream at all. The received INVITE will contain a minimal SDP body with no media lines (m=) or any of the accompanying media attributes. In this situation, an application would neither start, nor listen for, an RTP stream. The application `IncomingCalls` with no command line options will run this example.

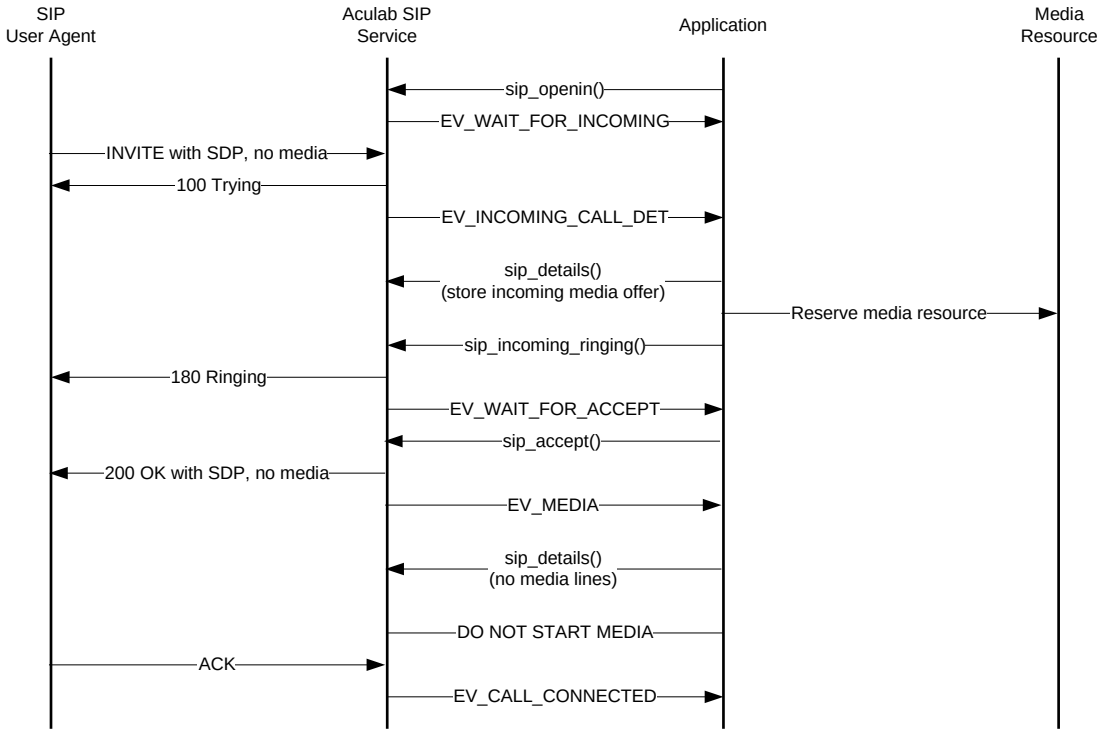

Figure 2-9 Incoming call – no media in SDP

Handling this situation is similar to the case where the `INVITE` contains a black hole connection address. When it receives an `EV_INCOMING_CALL_DET` it must call `sip_details` and check whether the media offer contains any media lines. If it does, the application must take note of this so that it does not start a media stream when it receives the `EV_MEDIA`.

Alerting a call with no media lines offered

```
ACU_ERR handle_ev_incoming_call_det(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER** offer)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_INCOMING_RINGING_PARMS sip_incoming_ringing_parms;

   // Here it is necessary to call sip_details in order to determine what,
   // if there is one, is contained in the media offer.
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }
```

```
   // Copy the media offer.
   *offer = clone_media_offer(sip_detail_parms.media_offer_answer);

   // There is a media offer. It is now necessary to determine the nature of the offer.
   if(NULL == (*offer)->media_descriptions)
   {
      // There is an SDP body contained in the INVITE. There are no media (m=) lines.
      g_no_media_lines = 1;
   }

   // At this stage a real application would need to reserve some media resource.
   // Even if there is no media stream offered, it is likely that one will be
   // nefotiated at a later stage.

   // Free the SIP_DETAIL_PARMS structure. This is essential after any call to
   // sip_details. If this function is not called, the application will leak memory.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Alert the call
   INIT_ACU_CL_STRUCT(&sip_incoming_ringing_parms);
   sip_incoming_ringing_parms.handle = handle;

   // Send the 180 Ringing response to the UAC.
   rc = sip_incoming_ringing(&sip_incoming_ringing_parms);
   if(0 != rc)
   {
      printf("Error in sip_incoming_ringing(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free any dynamically allocated memory
   free_media_offer(&sip_incoming_ringing_parms.media_offer_answer);

   return 0;
}
```

When the `EV_WAIT_FOR_ACCEPT` arrives, the application needs to check the global variable, `g_no_media_lines`, to see how to populate its answer:

Accepting a call where no media stream was offered

```
ACU_ERR handle_ev_wait_for_accept(const ACU_CALL_HANDLE handle, ACU_MEDIA_OFFER_ANSWER* offer)
{
   ACU_ERR rc = 0;
   SIP_ACCEPT_PARMS accept;
   ACU_MEDIA_OFFER_ANSWER *temp_answer;

   INIT_ACU_STRUCT(&accept);
   accept.handle = handle;

   if(g_no_media_lines)
   {
      // No media lines were offered in the initial INVITE so no media lines
      // are required in the 200 OK. A connection address must still be supplied.
      // This will establish a session with no RTP.
      temp_answer = (ACU_MEDIA_OFFER_ANSWER*)malloc(sizeof(ACU_MEDIA_OFFER_ANSWER));
      memset(temp_answer, 0,sizeof(ACU_MEDIA_OFFER_ANSWER));
      temp_answer->connection_address.address_type     = ACU_IPv4;
      temp_answer->connection_address.address          = (ACU_CHAR*)malloc(strlen(LOCAL_CONNECTION_ADDRESS) + 1);
      strcpy(temp_answer->connection_address.address, LOCAL_CONNECTION_ADDRESS);
   }
   else
   {
      // There was an SDP body in the initial INVITE. An answer based on this
      // offer must be supplied here.
      temp_answer = populate_media_answer(offer);
   }

   accept.media_offer_answer = *temp_answer;
   // Send the 200 OK response.
   rc = sip_accept(&accept);
   if(0 != rc)
   {
      printf("Error in sip_accept(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free all dynamically assigned memory
   free_media_offer(&temp_answer);
   free_media_offer(&offer);

   return 0;
}
```

Now, when the `EV_MEDIA` is received, the application must check the variable `g_no_media_lines` again. If it is set, the media stream should not be started.

2.10 Outgoing call with no media in SDP

The application wishes to establish a session with a UAS with, for the time being at least, no actual media. The flow of SIP messages will be as that for a normal call set up, except there will be no RTP stream started when the call is connected. The application `OutgoingCalls` with the following command line options will run this example:

```
OutgoingCalls r=<REMOTE IP ADDRESS> nomedia
```

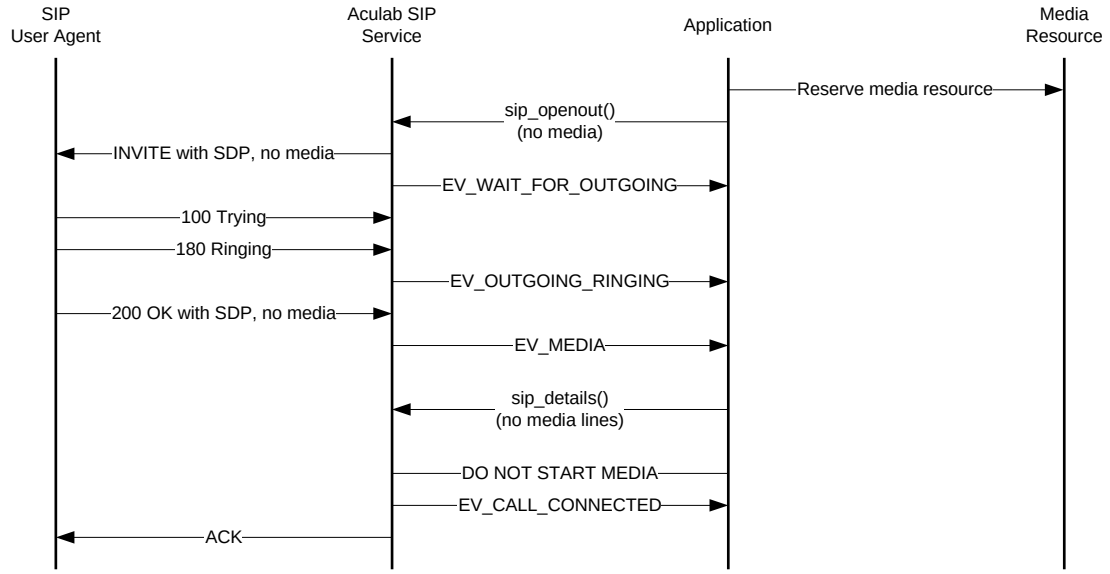

Figure 2-10 Outgoing call – no media in SDP

Again, this deviates from example 2.2 only slightly. The sequence of events is identical and the only change is there is considerably less work involved in populating the `out_sip_parms` structure.

Making an outgoing call with no media lines offered

```
ACU_ERR open_outgoing(ACU_CALL_HANDLE *handle)
{
   ACU_ERR rc = 0;
   SIP_OUT_PARMS out;

   // When opening the SIP port for incoming calls using the SIP Bridge it is
   // necessary to use sip_openout instead of call_openout.
   INIT_ACU_CL_STRUCT(&out);
   // Populate the net and addressing fields
   out.net = settings.sip_port;
   out.originating_addr = (ACU_CHAR*)malloc(strlen(settings.local_ip_address) + 1);
   strcpy(out.originating_addr, settings.local_ip_address);
   out.destination_addr = (ACU_CHAR*)malloc(strlen (settings.remote_ip_address) + 1);
   strcpy(out.destination_addr, settings.remote_ip_address);

   // Before making an outbound call, a real application may need to reserve
   // some media resource but it may well not. This method of establishing a
   // session may be used by a controller in a 3pcc call set up. If this is
   // the case, the controller itself will never start any media.

   // An SDP body is required with no media lines.
   out.media_offer_answer = (ACU_MEDIA_OFFER_ANSWER*)malloc(sizeof(ACU_MEDIA_OFFER_ANSWER));
   memset(out.media_offer_answer, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));
   out.media_offer_answer->connection_address.address = (ACU_CHAR*)malloc(strlen(LOCAL_CONNECTION_ADDRESS) + 1);
   strcpy(out.media_offer_answer->connection_address.address, LOCAL_CONNECTION_ADDRESS);
   out.media_offer_answer->connection_address.address_type = ACU_IPv4;

   // Send the INVITE to the UAS
   rc = sip_openout(&out);
   if(0 != rc)
   {
      printf("Error in sip_openout(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Store the call's handle to pass to the call control loop.
   *handle = out.handle;

   // Free any dynamically allocated memory
   free_media_offer(&out.media_offer_answer);

   return rc;
}
```

Now, as with the incoming calls where no media stream is established, there is nothing more for the application to do except handle the `EV_CALL_CONNECTED` as needs of the application demand. If it is acting as a controller in a 3pcc call set up, this is likely to be another outgoing call to another UAS with no SDP in the `INVITE`. The application need do nothing on receipt of the `EV_MEDIA` except pop the call's details off the queue.

2.11 Outgoing call with early media and forking – SDP in the INVITE.

This example shows how to handle an outgoing SIP call when the UAS establishes the media stream early with forked responses. This is the same as for example 2.2 the only difference being the 18x and 200 responses may originate from different sources. In this example the SIP User Agent may be a proxy relaying the 18x and 200 responses on behalf of two separate UAS servers. The first 180/183 response will contain an SDP body from one UAS while the 200 OK response originates from another UAS. This simply means that the `EV_MEDIA` will be raised to the application earlier followed by `EV_OUTGOING_RINGING` or/and `EV_PROGRESS` event. Upon reception of the 200 OK response the event `EV_CONNECTED` is raised. Due to the SIP service recognising that the responses originate from different sources (i.e, contain different 'To-tags') an `EV_FORKED` event will also be raised. All events should be handled in the same manner as before. In the case of an `EV_FORKED` it is recommended that a Re-INVITE is sent to the UAS that responded with the 200 OK response. Please refer to example 3.1 for more details.

Note that the media in the 200 response may differ to that received earlier in the 183 response therefore a new RTP session may need to be established via a Re-INVITE.

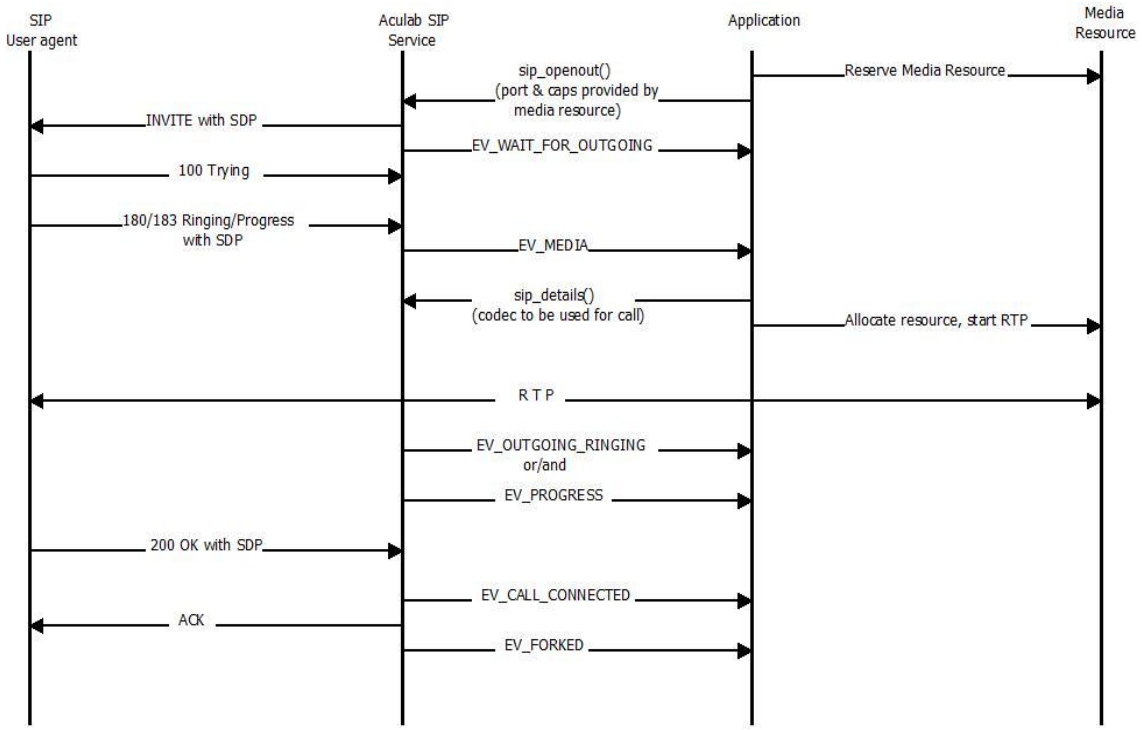

Figure 2-11 Outgoing call – early media and forking – SDP in INVITE

2.12 Call cleared locally

This is identical to clearing a call down using the generic call control API except that using the SIP Bridge API, the application controls the media resource and must therefore stop the RTP stream and release the media resource. These actions are not performed by the SIP Bridge API when the call idles.

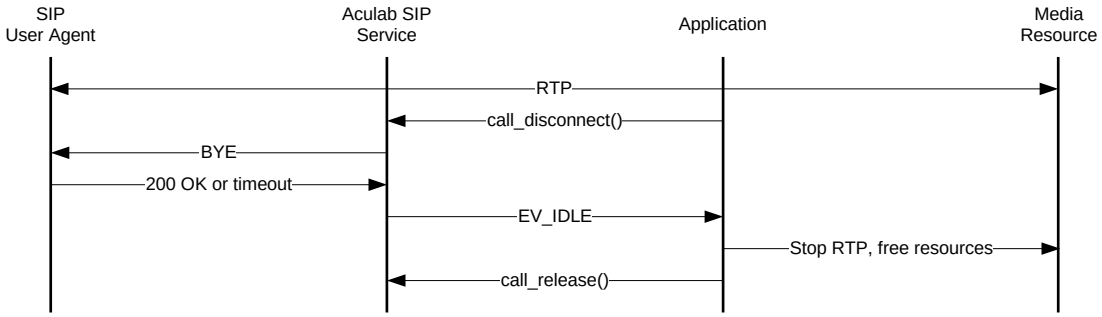

Figure 2-12 Call cleared locally

Disconnecting a call

```
ACU_ERR disconnect_call(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   CAUSE_XPARMS disconnect;

   INIT_ACU_CL_STRUCT(&disconnect);
   disconnect.handle = handle;
   // Although the call was created using the SIP Bridge API, it needs to be
   // disconnected using the generic API call, call_disconnect.
   rc = call_disconnect(&disconnect);
   if(0 != rc)
   {
      printf("Error in call_disconnect(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

The SIP service will now send a `BYE` to the UAS. It will raise an `EV_IDLE` to the application when either receives a 200 OK or the `BYE` times out. The application should handle this as follows:

Releasing a call

```
ACU_ERR handle_ev_idle(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   CAUSE_XPARMS release;

   // The call has received an EV_IDLE. Any previously allocated media resources
   // must be released here.

   INIT_ACU_CL_STRUCT(&release);
   release.handle = handle;
   // Although the call was created using the SIP Bridge API, it needs to be
   // released using the generic API call, call_release.
   rc = call_release(&release);
   if(0 != rc)
   {
      printf("Error in call_release(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

The call is now complete and all resources relating to it have been released.

2.13 Call cleared remotely

Again, this is identical to clearing a call down using the generic call control API except that using the SIP Bridge API, the application controls the media resource and must therefore stop the RTP stream and release the media resource when the call idles.

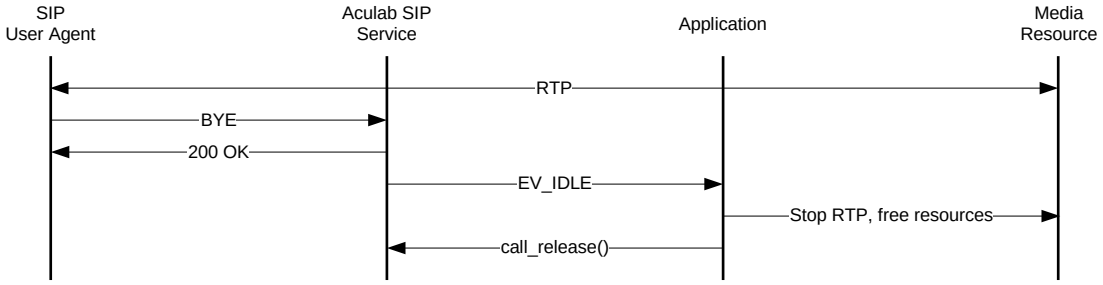

Figure 2-13 Call cleared remotely

Here, the SIP service has received a `BYE`, automatically responded with a 200 `OK` and raised an `EV_IDLE` to the application. This should be handled in the same manner as the previous example.

3 3pcc - mid call flows

The call flows in this section assume that a call is already established between the local and remote SIP user agents. Reference is given to the accompanying example applications and to the commonly used C functions that are defined in the appendices. All of these examples assume a single media stream can be found in the media offer. Multiple media offers involve more logic for the application but are dealt with in essentially the same manner. The following methods, in conjunction with those described earlier, can be used to implement the call flows shown in RFC3725 - Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP).

3.1 Aculab sending re-INVITE with SDP (UAC)

For whatever reason the application wishes to re-negotiate the media stream. In SIP terms, this will mean an `INVITE` being sent to the UAS with different media settings in the SDP body to those currently being used. The flow of SIP messages will be the same as that for an initial `INVITE` except that the UAS will not send any 180 response. The application `ReINVITE_UAC` with no arguments will run this example.

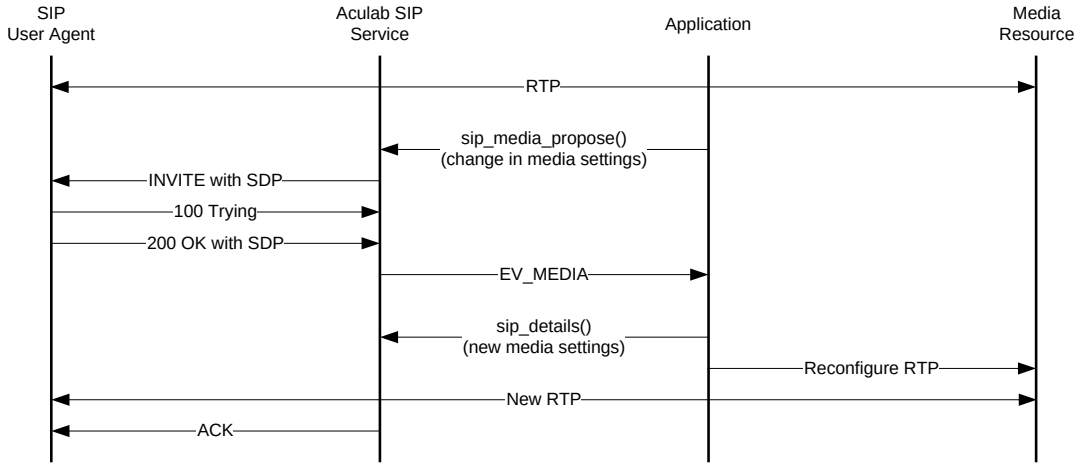

Figure 3-1 Aculab sending re-INVITE with SDP (UAC)

The example code here will establish a call as the UAS (using `sip_openin`) and re-negotiate the codec to use G.729. The application calls `sip_media_propose` populating the `SIP_MEDIA_PROPOSE_PARMS` structure with the existing call handle and the new media offer. Here it is assumed that the call is already connected:

Create a new media offer for an existing call.

```
ACU_ERR send_reinvite(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_MEDIA_PROPOSE_PARMS proposal;
   ACU_MEDIA_OFFER_ANSWER* offer = NULL;

   // populate_media_offer is defined in appendix B.1
   offer = populate_media_offer(ACU_G729_PAYLOAD_NUMBER, 0, 0);

   INIT_ACU_CL_STRUCT(&proposal);
   proposal.handle = handle;
   proposal.media_offer_answer = *offer;
   // Send the re-INVITE
   rc = sip_media_propose(&proposal);
   if(0 != rc)
   {
      printf("Error in sip_media_propose(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free any dynamically allocated memory.
   free_media_offer(&offer);

   return 0;
}
```

The `INVITE` with the new media offer will now be sent to the remote SIP which (if it is happy with the offer) will respond with a 200OK containing its answer. In response to this, the SIP service will raise an `EV_MEDIA` to the application, which should then reconfigure the RTP stream as required. The UAS may, of course, reject the new offer in which case it is up to the application to decide how to proceed.

3.2 Aculab receiving re-INVITE with SDP (UAS)

This section will describe how to respond to a `ReINVITE` as described in section 3.1 above. The application `ReINVITE_UAS` with the following arguments will run this example:

```
ReINVITE_UAS r=<REMOTE IP ADDRESS>
```

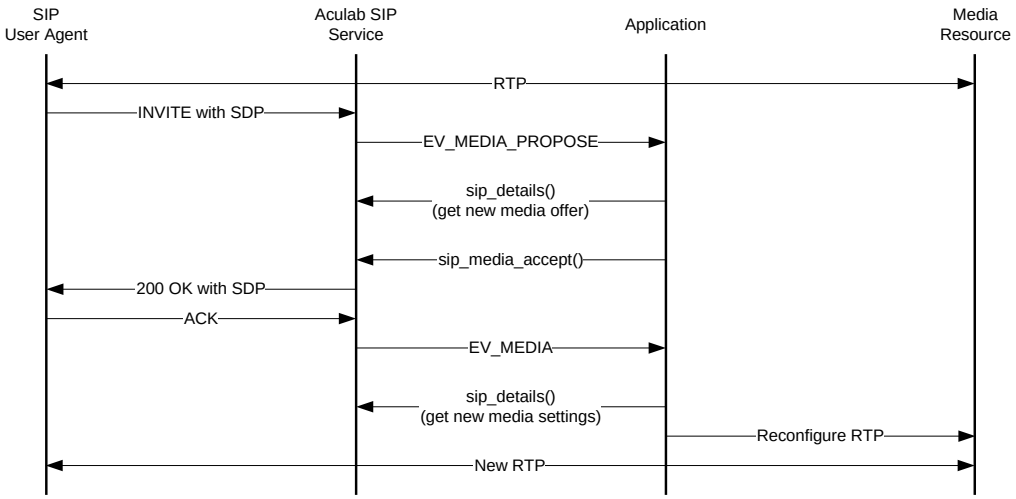

Figure 3-2 Aculab receiving re-INVITE with SDP (UAS)

Here Aculab receives a new media offer mid call. This will prompt the SIP service to raise an `EV_MEDIA_PROPOSE` to the application, which should call `sip_details` in order to determine the nature of the new media offer. It will then accept the new media offer using `sip_media_accept` (see section 3.7 for details on how to reject this offer):

Accept a mid call media offer

```
ACU_ERR handle_ev_media_propose(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_MEDIA_ACCEPT_PARMS sip_media_accept_parms;
   ACU_MEDIA_OFFER_ANSWER* answer = NULL;

   // Retrieve call details.
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Populate the SDP body of the response using the new media offer
   // as the starting point.
   answer = populate_media_answer(sip_detail_parms.media_offer_answer);
   if(NULL == answer)
   {
      // No supported codec has been offered.
      // The new offer should be rejected here. See section 3.7
   }

   INIT_ACU_CL_STRUCT(&sip_media_accept_parms);
   sip_media_accept_parms.handle = handle;
   sip_media_accept_parms.media_offer_answer = *answer;
   // Send the 200 OK response
   rc = sip_media_accept(&sip_media_accept_parms);
   if(0 != rc)
   {
      printf("Error in sip_media_accept(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free any dynamically allocated memory
   free_media_offer(&answer);

   // Free the SIP_DETAIL_PARMS structure.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

A 200 OK will now be sent to the far end. The application will receive an `EV_MEDIA` and should then reconfigure the RTP stream accordingly.

3.3 Aculab sending re-INVITE with black hole SDP (UAC)

Sending a mid call re-`INVITE` with black hole SDP (c=0.0.0.0) is the method described in RFC2543 (the original SIP standard, superseded by RFC3261) by which a UAC would place a UAS on hold. This has been deprecated as the means to place a call on hold (although many user agents still use this method) but is used instead in various third party call control scenarios. Used on its own, however, it is indistinguishable from placing a call on hold. The application `ReINVITE_UAC` with the following command line options will run this example:

```
ReINVITE_UAC bh
```

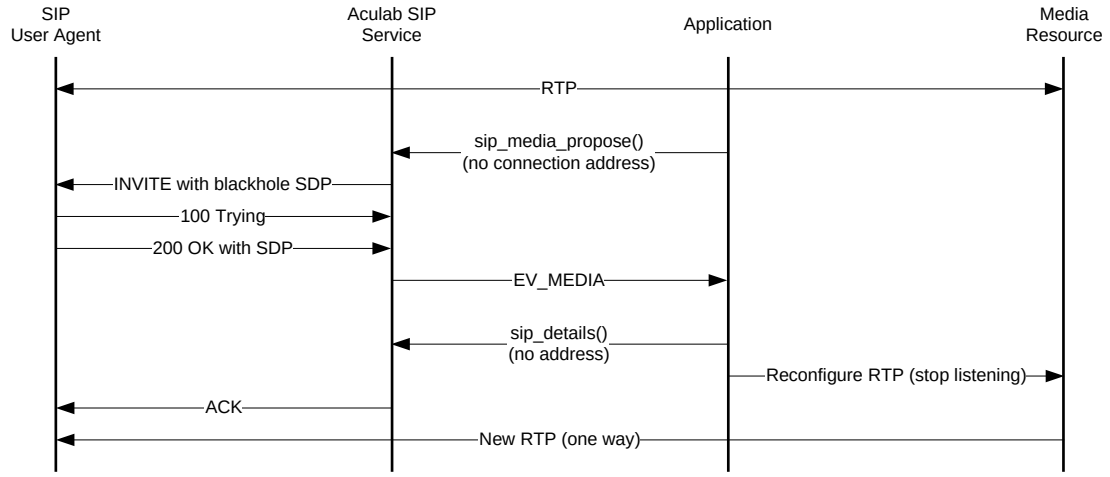

Figure 3-3 Aculab sending re-INVITE with black hole SDP (UAC)

Here the UAC will call `sip_media_propose` specifying a connection address of "0.0.0.0" in the `SIP_MEDIA_PROPOSE_PARMS` and setting all other parameters to be the same as the current media session. A re-INVITE will then be sent to the UAS containing an SDP connection address of c=0.0.0.0:

Create a new media offer for an existing call with black hole SDP

```
ACU_ERR send_black_hole_reinvite(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_MEDIA_PROPOSE_PARMS sip_media_proposal_parms;
   ACU_MEDIA_OFFER_ANSWER* offer = NULL;

   // The second paramter here specifies that the offer is to contain
   // a connection address of "0.0.0.0"
   offer = populate_media_offer(ACU_PCMA_PAYLOAD_NUMBER, TRUE, 0);

   INIT_ACU_CL_STRUCT(&sip_media_proposal_parms);
   sip_media_proposal_parms.handle = handle;
   sip_media_proposal_parms.media_offer_answer = *offer;
   // Send the re-INVITE
   rc = sip_media_propose(&sip_media_proposal_parms);
   if(0 != rc)
   {
      printf("Error in sip_media_propose(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free any dynamically allocated memory.
   free_media_offer(&offer);

   return 0;
}
```

The re-`INVITE` will now be sent to the UAS and an `EV_MEDIA` will be raised to the application when the SIP service receives the 200 OK from the far end. It should then reconfigure the media resource so that it no longer listens for RTP. It will keep sending RTP as before unless this was also re-negotiated to use a different codec.

3.4 Aculab receiving re-INVITE with black hole SDP (UAS)

This section will describe how to respond to a `ReINVITE` as described in section 3.3 above. The application `ReINVITE_UAS` with the following arguments will run this example:

```
ReINVITE_UAS r=<REMOTE IP ADDRESS>
```

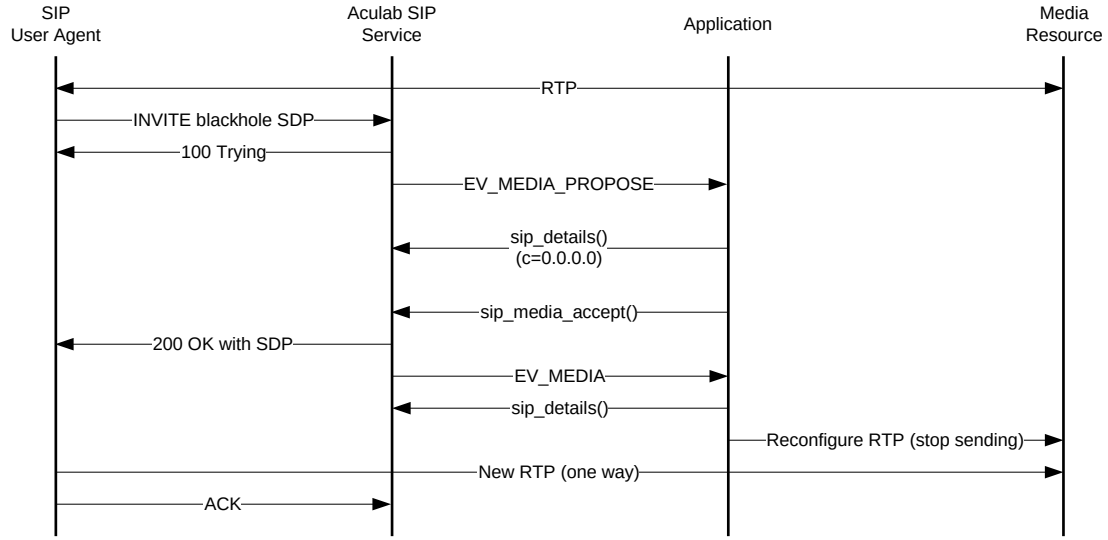

Figure 3-4 Aculab receiving re-INVITE with black hole SDP (UAS)

On receipt of the re-`INVITE`, the SIP service will raise an `EV_MEDIA_PROPOSE` to the application, which, as usual, will have to call `sip_details` in order to determine the nature of the re-INVITE. In this case, the connection address is found to be “0.0.0.0”, i.e. black hole SDP. There is no need for any code here as accepting this re-INVITE is the same as that described in section 3.2 (for that matter, it is essentially the same as section 2.1). The only extra work that is required is for the application to note that there is a connection address of “0.0.0.0” given in the media offer (a straight forward `strcmp`) so that, when the `EV_MEDIA` is raised, it can stop sending any RTP.

3.5 Aculab sending re-INVITE with no SDP (UAC)

Although in this example there is already a media stream established, this method will more normally be used in a 3pcc call set up scenario. It is a request to the UAS to make an offer of its own which, in a 3pcc situation, would generally be used by a controller to construct another offer to another user agent. The basic method described here is all that is needed to do this. The application `ReINVITE_UAC` with the following command line options will run this example:

```
ReINVITE_UAC nosdp
```

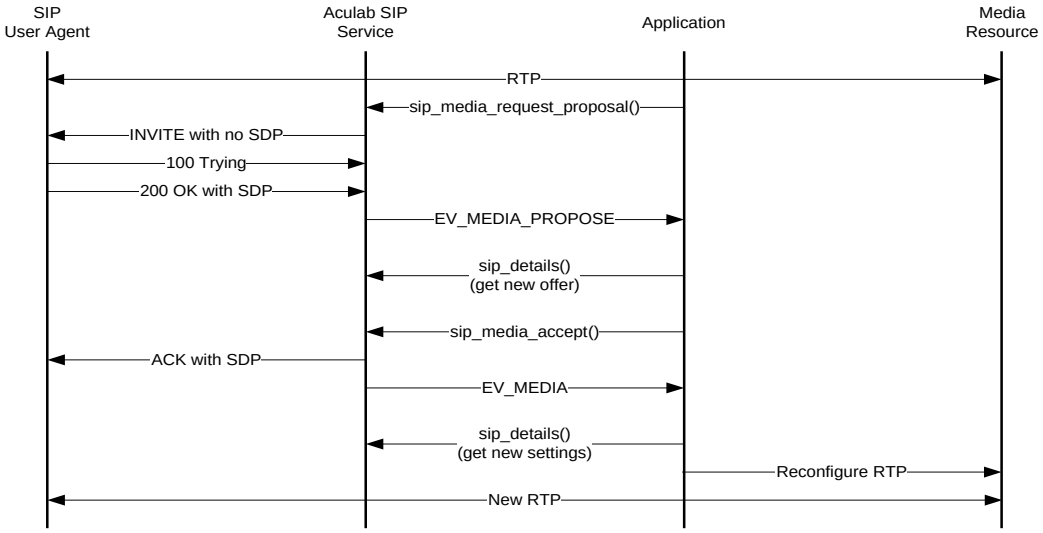

Figure 3-5 Aculab sending re-INVITE with no SDP (UAC)

The application will populate a `SIP_MEDIA_REQUEST_PROPOSAL_PARMS` structure (this consists only of the call's handle) and call `sip_media_request_proposal`. In response to this, the SIP service will send a re-INVITE to the UAS, which contains no SDP body.

Requesting a new media offer from a UAS

```
ACU_ERR request_new_media_offer(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_MEDIA_REQUEST_PROPOSAL_PARMS sip_media_request_proposal_parms;

   // The re-INVITE is to be sent with no SDP body, i.e. it is a request to the
   // UAS for a new media offer.
   INIT_ACU_CL_STRUCT(&sip_media_request_proposal_parms);
   sip_media_request_proposal_parms.handle = handle;
   // Send the re-INVITE
   rc = sip_media_request_proposal(&sip_media_request_proposal_parms);
   if(0 != rc)
   {
      printf("Error in sip_media_request_proposal(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

The UAS will realise that this is a request for a media offer and will provide its own in

the 200 OK. The SIP service will raise an EV_MEDIA_PROPOSE to the application which will need to process this offer in exactly the same manner as in section 2.6. An answer to the media offer will be given in the ACK after a call to sip_media_accept. An EV_MEDIA will then be raised to the application, which can then reconfigure the RTP as appropriate.

3.6 Aculab receiving re-INVITE with no SDP (UAS)

This section will describe how to respond to a ReINVITE as described in section 3.5 above. The application ReINVITE_UAS with the following arguments will run this example:

```
ReINVITE_UAS r=<REMOTE IP ADDRESS>
```

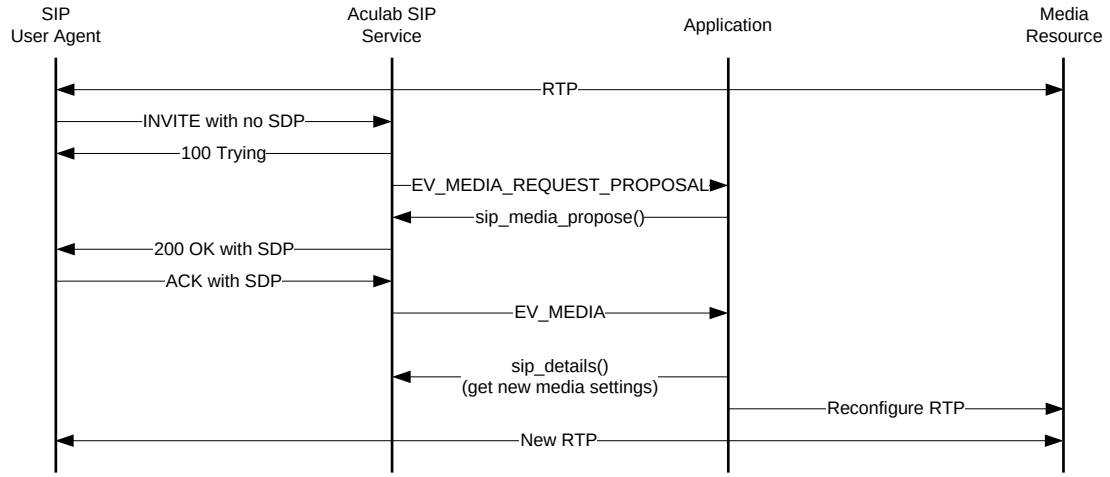

Figure 3-6 Aculab receiving re-INVITE with no SDP (UAS)

Unlike as in previous examples, when a re-INVITE with no SDP is received, the application need not determine this from the result of a call to sip_details as this will have been done by the SIP service itself. Instead, the service will raise an EV_MEDIA_REQUEST_PROPOSAL to the application which should read this as a requirement for it to make a proposal of its own with a call to sip_media_propose. This should be done in the same way as in section 3.1 although instead, the SIP service will supply a media offer in the 200 OK rather than an INVITE. Once the ACK (containing an SDP body) has been received from the UAC, an EV_MEDIA will be raised to the application, which should reconfigure the RTP stream accordingly.

3.7 Aculab rejecting a new media offer (UAS)

In some instances (or possibly the majority) the application may wish to reject a mid call media offer. For example, a UAC wishes to reduce the bandwidth being used by the call by changing the codec from G711 to G723. It would send a re-`INVITE` to the UAS containing a new offer that reflects this. However, the UAS does not support G723 and would need to reject the new offer. The application `ReINVITE_UAS` with the following arguments will run this example:

```
ReINVITE_UAS r=<REMOTE IP ADDRESS> reject
```

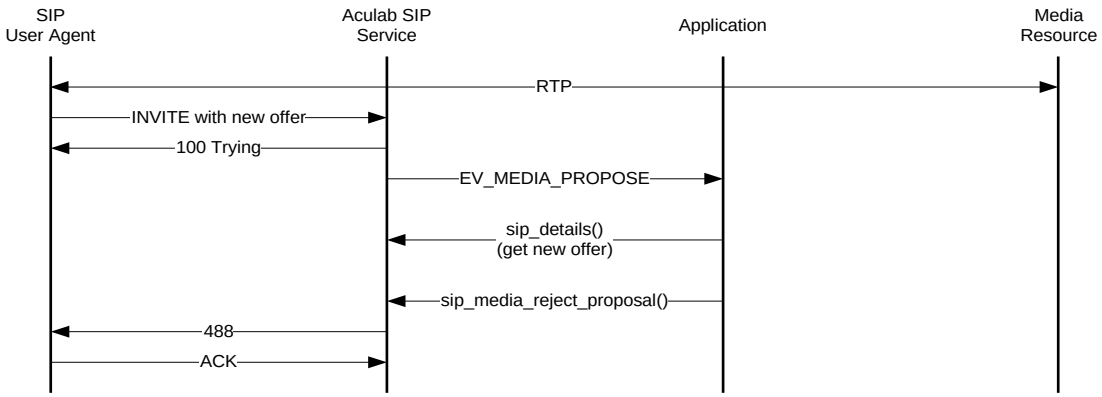

Figure 3-7 Aculab rejecting a new media offer (UAS)

There is very little to this. An `EV_MEDIA_PROPOSE` would be raised to the application on receipt of a re-`INVITE`. The application would then call `sip_details` to determine the nature of the new offer and, based on certain criteria (which would be defined by the programmer), would accept or reject the offer. Here it will be rejected automatically.

Rejecting a new media offer

```
ACU_ERR handle_ev_media_propose(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_MEDIA_REJECT_PROPOSAL_PARMS sip_media_reject_proposal_parms;

   // Retrieve call details.
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // An application would usually have certain criteria that it would
   // use in order to determine whether a re-INVITE should be accepted
   // or rejected. For the purposes of this example it is rejected
   // regardless of the offer's contents.
   INIT_ACU_CL_STRUCT(&sip_media_reject_proposal_parms);
   sip_media_reject_proposal_parms.handle = handle;
   rc = sip_media_reject_proposal(&sip_media_reject_proposal_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free the SIP_DETAIL_PARMS structure.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

Notice two points from the above code and call flow. Firstly, if nothing else is supplied, the default SIP response sent when an offer is rejected is ‘488 – Not acceptable here’, however, any valid SIP code may be supplied in the `sip_media_reject_proposal_parms.protocol_specific. sip_code` field. Secondly, the media resource has not been modified in any way. It is possible that either the UAC or UAS would want to clear down the call at this point but certainly not required. Taking this SIP transaction on its own, there is no need to modify the media at all.

3.8 Handling a rejected mid call media offer (UAC)

This section will describe what happens when a re-`INVITE` sent to a UAS has been rejected. The application `ReINVITE_UAC` with no arguments will run this example.

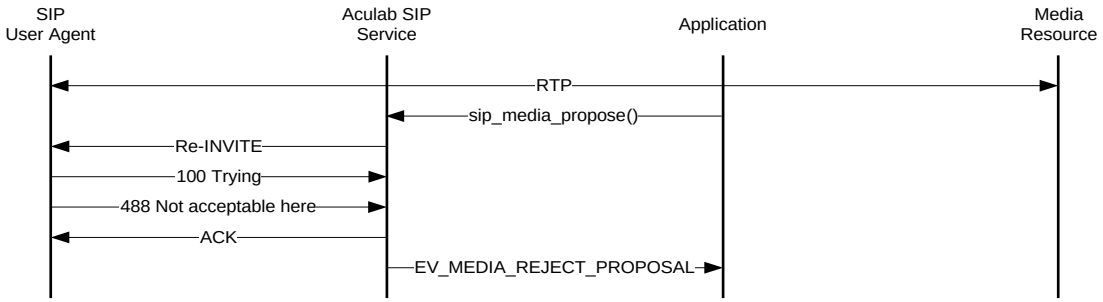

Figure 3-8 Handling a rejected mid call media offer (UAC)

There is essentially nothing to say here as the above figure says it all. When the UAS rejects the re-`INVITE`, an `EV_MEDIA_REJECT_PROPOSAL` will be raised to the application and the RTP stream will require no modification as in section 3.7. In the accompanying example code, the UAC will clear down the call but there is no reason why an application must do this.

4 Supplementary services

The call flows in this section assume that a call is already established between the local and remote SIP user agents. Reference is given to the accompanying example applications where appropriate. This guide is concerned with the extended SIP API, however, the user wishing to implement hold, reconnect or transfer should follow the same guidelines whilst bearing the following in mind:

1. Wherever `sip_feature_send` is used, substitute `call_feature_send` in its place.
2. The media handling will be dealt with by the SIP service (through the Media Handler Plugin) and so no `EV_MEDIAs` will be raised.

4.1 Hold – Aculab initiating hold request

Placing a call on hold with the extended SIP API is almost the same as placing a call on hold with the generic call control API. The notable exception being that, if the call was opened with `sip_openin/out` rather than `call_openin/out`, an `EV_MEDIA` will be raised to the application after the `EV_HOLD`. Placing a call on hold is illustrated in the call transfer example applications that accompany sections 4.5, 4.6 & 4.7 below.

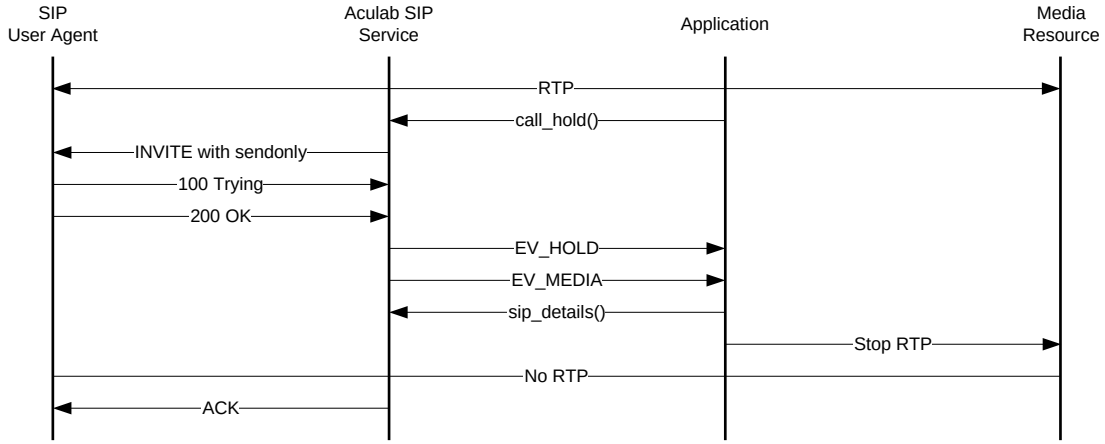

Figure 4-1 Hold – Aculab initiating hold request

4.2 Hold – Aculab responding to hold request

Responding to a hold request with the extended SIP API is the same as described in the generic call control API guide except that `sip_feature_send` is used instead of `call_feature_send` and an `EV_MEDIA` will be raised. Responding to a hold request is illustrated in the call transfer example applications that accompany sections 4.5, 4.6 & 4.7 below.

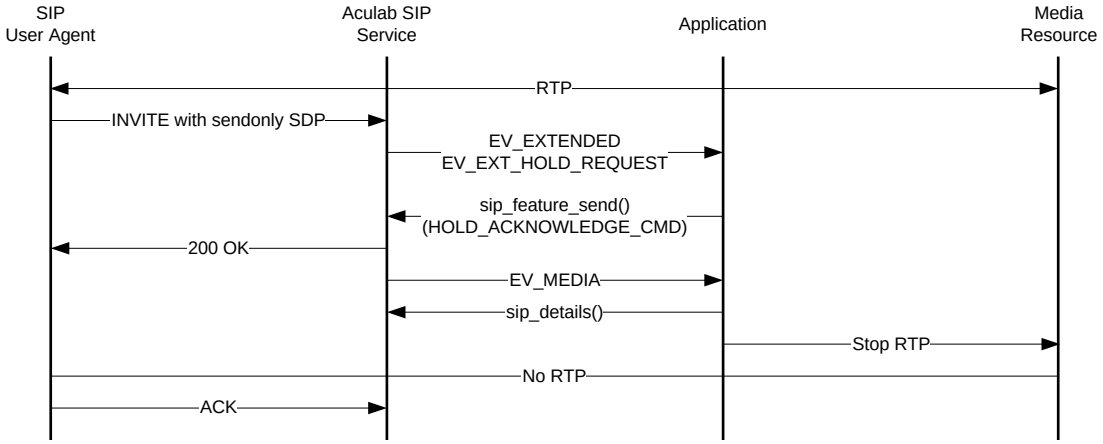

Figure 4-2 Hold – Aculab responding to hold request

When the SIP service receives an `INVITE` with an SDP containing the `a= attribute` of `a=sendonly`, it will recognise this as a hold request and, instead of raising an `EV_MEDIA_PROPOSE`, it will raise an `EV_EXTENDED` with `state_xparms.extended_state = EV_EXT_HOLD_REQUEST`. To accept this request, the application needs to call `sip_feature_send`:

Acknowleding a hold request

```
ACU_ERR handle_ev_ext_hold_request(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_FEATURE_DETAIL_PARMS sip_feature_detail_parms;

   // Received hold request. Here the application will blindly accept all such
   // requests. This will not always be the case. If, for any reason, an
   // application wishes to reject the hold request, it should set
   // sip_feature_detail_parms.feature.hold.command = HOLD_REJECT_CMD; instead.
   INIT_ACU_CL_STRUCT(&sip_feature_detail_parms);
   sip_feature_detail_parms.handle = handle;
   sip_feature_detail_parms.feature_type = FEATURE_HOLD_RECONNECT;
   sip_feature_detail_parms.feature.hold.command = HOLD_ACKNOWLEDGE_CMD;
   rc = sip_feature_send(&sip_feature_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_feature_send(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

The 200 OK will now be sent (containing the attribute, `a=recvonly`) and an `EV_MEDIA` will be raised to the application, which should pause the media stream.

4.3 Reconnect – Aculab initiating reconnect request

Reconnecting a call that has previously been placed on hold with the extended SIP API is almost the same as reconnecting a call with the generic call control API. The notable exception being that, if the call was opened with `sip_openin/out` rather than `call_openin/out`, an `EV_MEDIA` will be raised to the application after the `EV_CALL_CONNECTED`. Reconnecting a call is illustrated in the call transfer example applications that accompany sections 4.5, 4.6 & 4.7 below. In this case, the following application and arguments must be used:

Transferor reconnect

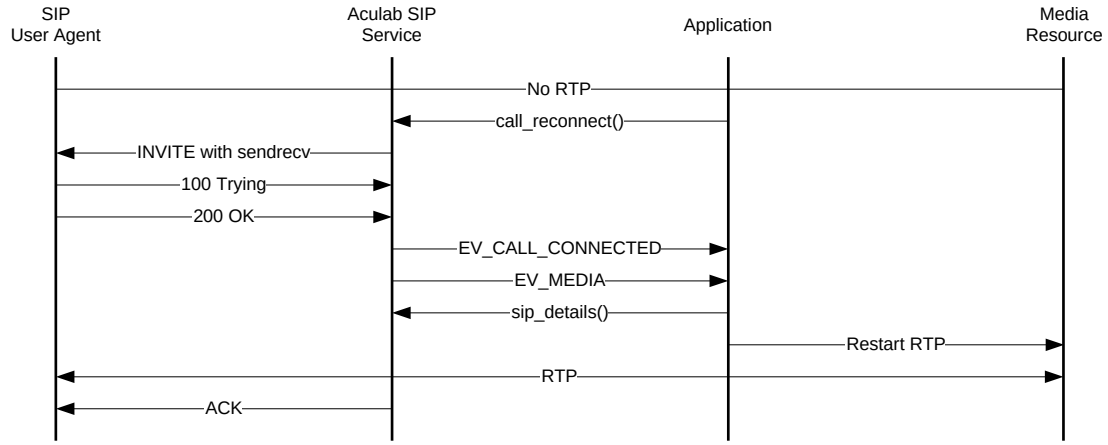

Figure 4-3 Reconnect – Aculab initiating reconnect request

4.4 Reconnect – Aculab responding to reconnect request

Responding to a reconnect request with the extended SIP API is the same as described in the generic call control API guide except that `sip_feature_send` is used instead of `call_feature_send` and an `EV_MEDIA` will be raised. Responding to a reconnect is illustrated in the call transfer example applications that accompany sections 4.5, 4.6 & 4.7 below.

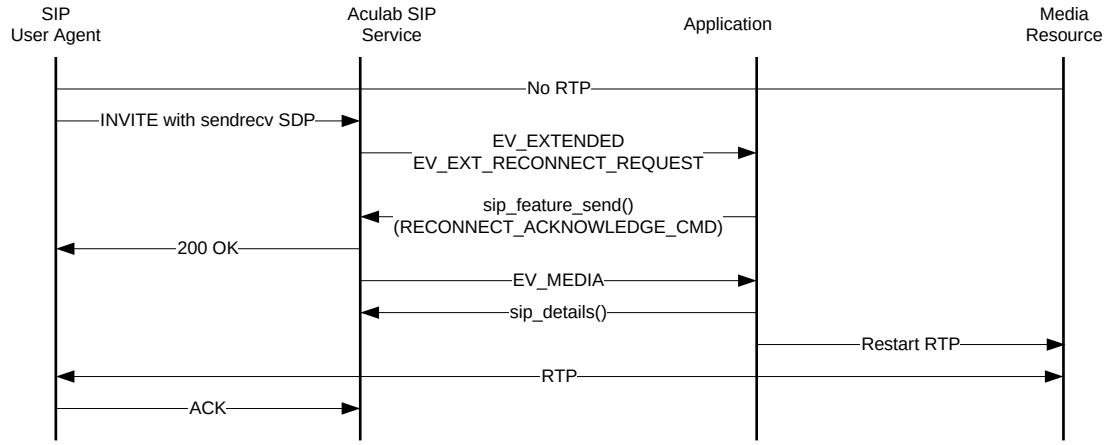

Figure 4-4 Reconnect – Aculab responding to reconnect request

When the SIP service receives an `INVITE` with an SDP body containing the `a=attribute` of `a=sendrecv` and the call is currently on hold, it will recognise this as a reconnect request and, instead of raising an `EV_MEDIA_PROPOSE`, it will raise an `EV_EXTENDED` with `state_xparms.extended_state = EV_EXT_RECONNECT_REQUEST`. To accept this request, the application needs to call sip_feature_send:

Acknowledging a reconnect request

```
ACU_ERR handle_ev_ext_reconnect_request(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_FEATURE_DETAIL_PARMS sip_feature_detail_parms;

   INIT_ACU_CL_STRUCT(&sip_feature_detail_parms);
   sip_feature_detail_parms.handle = handle;
   // The transferor has decided not to transfer the call and wishes to
   // reconnect instead. If it is desired that the reconnect should not
   // take place, RECONNECT_REJECT_CMD should be used instead.
   sip_feature_detail_parms.feature_type = FEATURE_HOLD_RECONNECT;
   sip_feature_detail_parms.feature.hold.command = RECONNECT_ACKNOWLEDGE_CMD;
   rc = sip_feature_send(&sip_feature_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_feature_send(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

The 200 OK will now be sent (with a=sendrecv in the SDP body) and an EV_MEDIA will be raised to the application which should restart the media stream.

4.5 Transfer – Aculab initiating transfer

In SIP terms this party in a call transfer scenario is know as the transferor. Transfer is one of the more complex call flows in SIP. Aculab have implemented Attended Call Transfer as defined in section 2.5 of Session Initiation Protocol Service Examples (to find the latest revision of this document, go to URL http://www.ietf.org/ and then search on ‘ sipping service examples ’). Initiating a call transfer in SIP is an identical process to that described in the generic Call Control API Guide with the following exceptions:

1. When `call_transfer` is called, the enquiry call will receive an extra `EV_HOLD`.
2. `call/sip_openout` may be used instead of `call_enquiry` as `call_enquiry` in SIP simply creates a normal outgoing call.

With these two points to bear in mind, the reader is referred to the Call Transfer section starting at 4.58 of the generic Call Control API Guide.

The call flow shown below shows when the several `EV_MEDIAs` will be raised in a call transfer scenario. These are needed for the application to know when and where to start/stop the media streams.

The application `Transferor.exe`

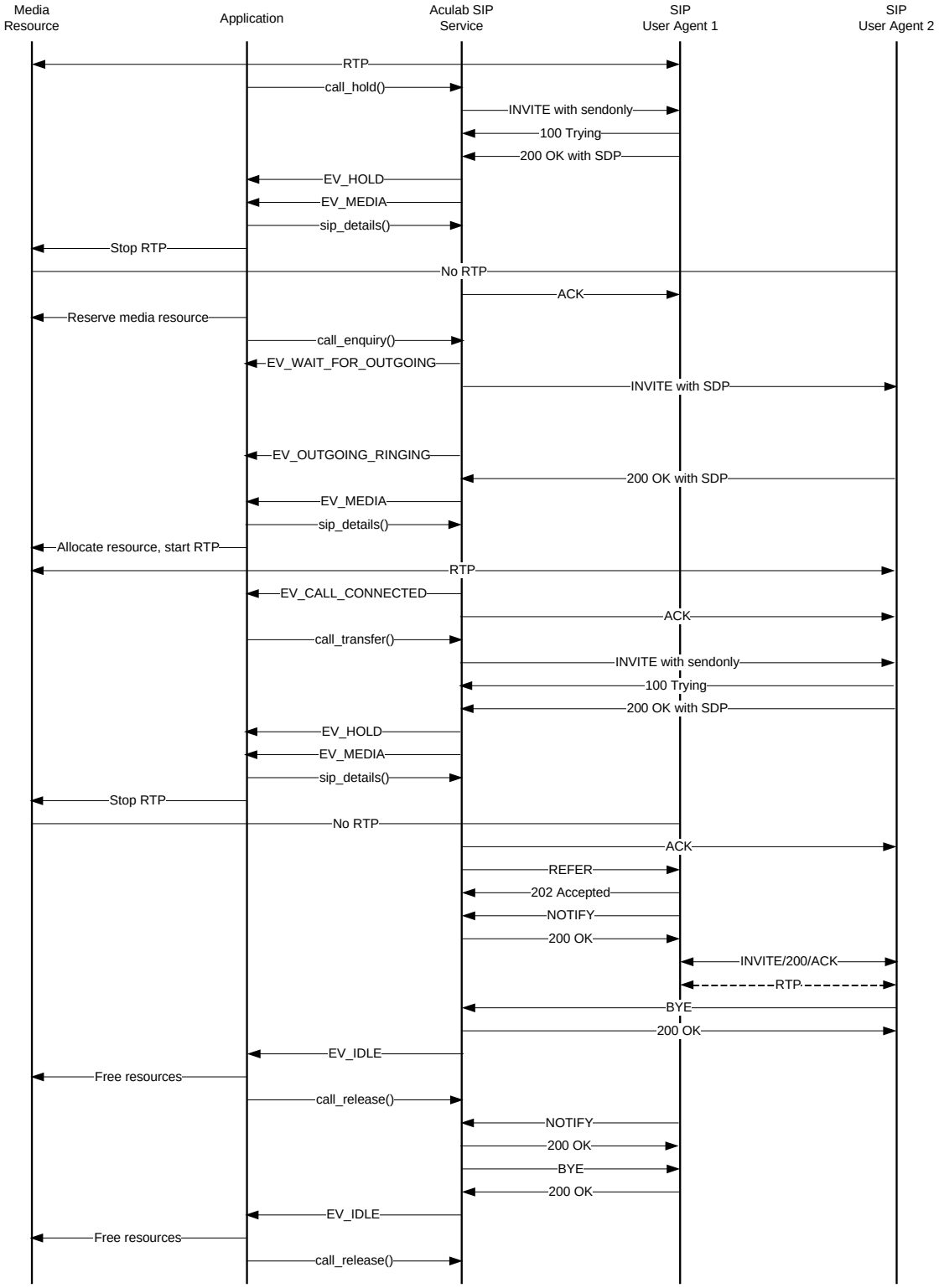

Figure 4-5 Transfer – Aculab initiating transfer

4.6 Transfer – Aculab responding to transfer request

In SIP terms this party in a call transfer scenario is know as the transferee. The application Transferee.exe with the following arguments will run this example:

```
Transferee r=<TRANSFEROR IP ADDRESS>
```

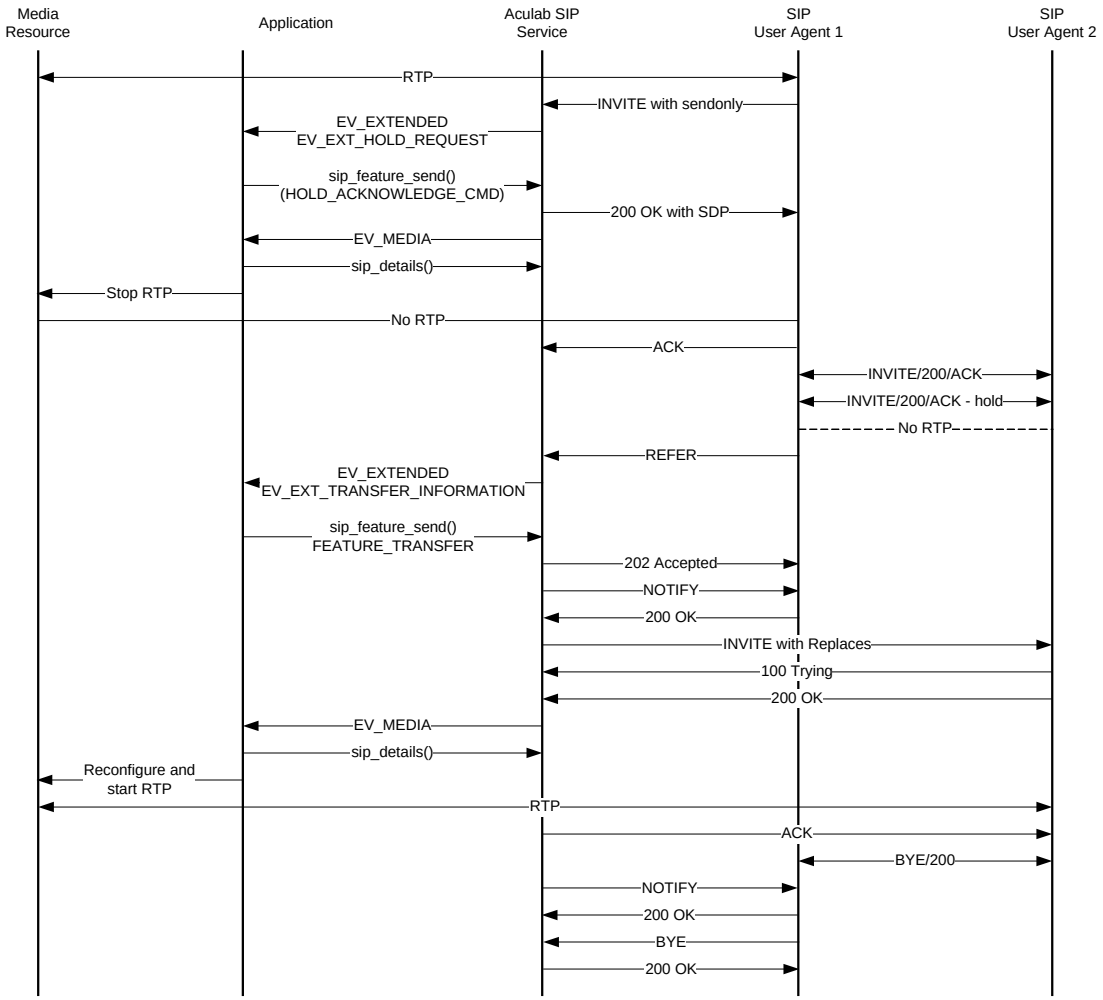

Figure 4-6 Transfer – Aculab responding to transfer request

As can be seen in the above call flow, the transferee needs to be able to react to two additional extended events, `EV_EXT_HOLD_REQUEST` and `EV_EXT_TRANSFER_INFORMATION`. The hold request should be handled in the same way as in section 4.2. The `EV_EXT_TRANSFER_INFORMATION` is handled in a similar fashion as follows:

Handling a transfer request from a transferor

```
ACU_ERR handle_ev_ext_transfer_information(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_FEATURE_DETAIL_PARMS sip_feature_detail_parms;

   INIT_ACU_CL_STRUCT(&sip_feature_detail_parms);
   sip_feature_detail_parms.handle = handle;
   // Received some transfer information. This can either be a REFER request
   // (as it is in this case) or an INVITE with Replaces. If the transfer is
   // to be rejected, the application must specify a valid SIP code in
   // the sip_feature_detail_parms.feature.transfer.failure_code field.
   sip_feature_detail_parms.feature_type = FEATURE_TRANSFER;
   // This is where the offer to the transfer target is populated. It is not
   // mandatory to supply a media offer here but the normal offer/answer
   // exchange must take place so it either must be supplied here or in the
   // 200 OK response from the transfer target.
   sip_feature_detail_parms.feature.transfer.media_offer_answer =
populate_media_offer(ACU_PCMA_PAYLOAD_NUMBER, 0, 0);
   // Send the INVITE with Replaces
   rc = sip_feature_send(&sip_feature_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_feature_send(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free any dynamically allocated memory
   free_media_offer(&sip_feature_detail_parms.feature.transfer.media_offer_answer);

   return 0;
}
```

As noted in the above code, receipt of an `EV_EXT_TRANSFER_INFORMATION` may be the result of either a `REFER` request or an `INVITE` with Replaces. In the case of the transferee it is the `REFER` as can be seen from the above call flow (the `INVITE` with Replaces will be dealt with in section 4.7 below). Several messages will be sent by the SIP service in response to this API call:

1. The `REFER` from the transferor will be acknowledged with a '202 Accepted'.
2. The transferor will be informed that the transfer is about to take place with a `NOTIFY` message (which the transferor will acknowledge with a 200 OK).
3. An `INVITE` with Replaces will be sent to the transfer target, the details of which will be contained in the Refer-to header of the `REFER` request.

Once the 200 OK has been received from the transfer target, an `EV_MEDIA` will be raised with the new media settings. The application will then reconfigure the RTP stream so that it is now sending to the transfer target. Behind the scenes, the SIP service will send a `NOTIFY` to the transferor to inform it that the transfer has been successful. The transferor will in turn send a `BYE` message, which will be acknowledged with a 200 OK. Unlike other call scenarios, this `BYE` request will not result in an `EV_IDLE` being raised to the application as the call handle is now associated with the call to the transfer target.

4.7 Transfer – Aculab acting as transfer target

In SIP terms this party in a call transfer scenario is know as the transfer target. The application TransferTarget.exe with no arguments will run this example.

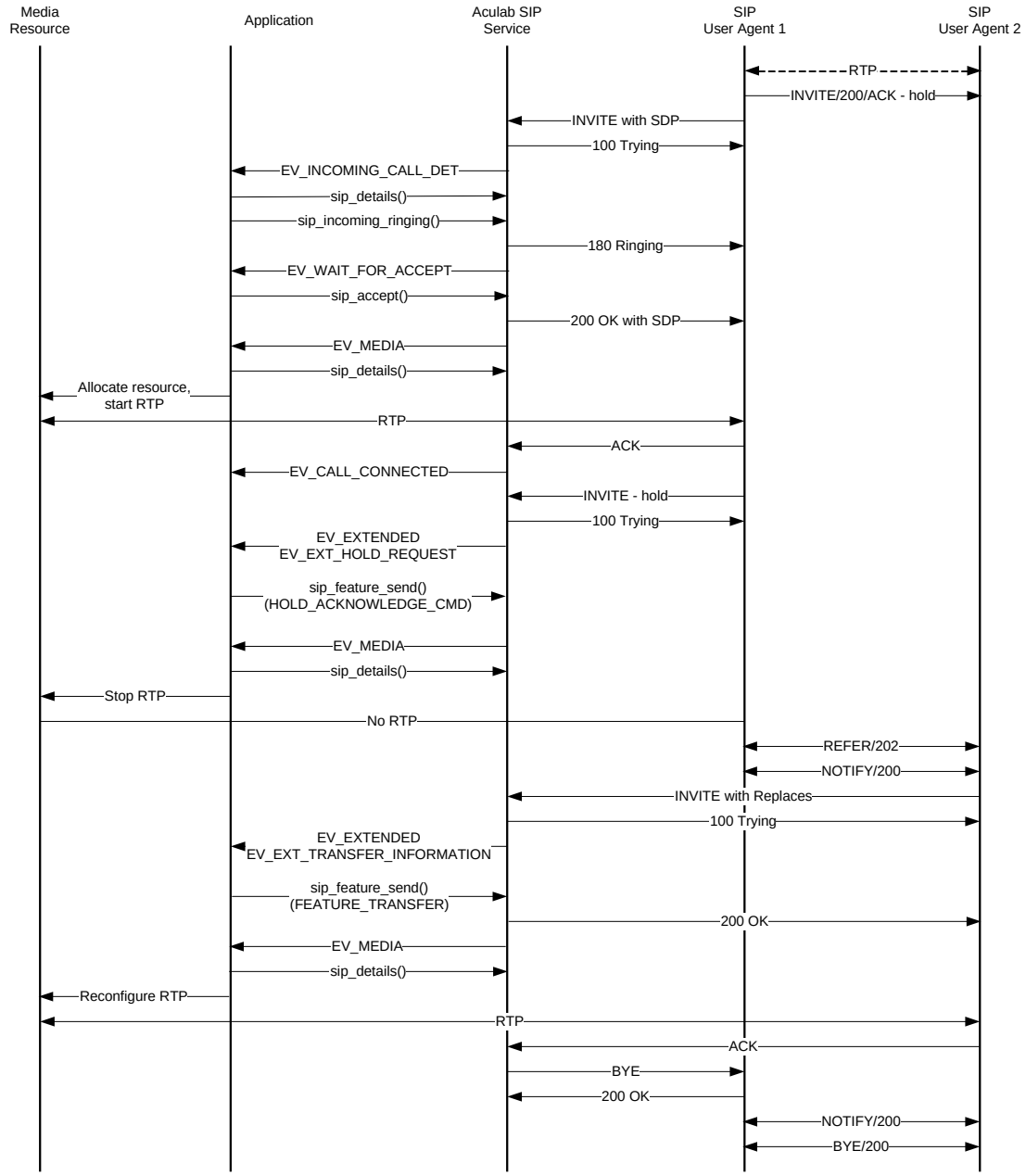

Figure 4-7 Transfer – Aculab acting as transfer target

As can be seen in the following call flow, the same two additional extended events must be catered for by the application as are in example 4.6 above. The `EV_EXT_HOLD_REQUEST` is the extra hold that is mentioned in section 4.5 and should be handled in the same way as in section 4.2.

The `EV_EXT_TRANSFER_INFORMATION` should be handled in a similar manner as in section 4.6 the difference being that a media answer must be provided to the offer given in the `INVITE` with Replaces. It is possible that this `INVITE` will not contain a media offer in which case the transfer target would have to provide one here. For the purposes of these examples it is assumed that such an offer exists in the `INVITE` and

the reader is referred to section 2.9 for how to handle an `INVITE` with Replaces containing no SDP body.

Handling a transfer request from a transferee

```
ACU_ERR handle_ev_ext_transfer_information(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_FEATURE_DETAIL_PARMS sip_feature_detail_parms;
   SIP_DETAIL_PARMS sip_detail_parms;

   // Retrieve the call details
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   INIT_ACU_CL_STRUCT(&sip_feature_detail_parms);
   sip_feature_detail_parms.handle = handle;
   // Received some transfer information. This can either be a INVITE with Replaces
   // (as it is in this case) or a REFER . If the transfer is to be rejected, the
   // application must specify a valid SIP code in
   // the sip_feature_detail_parms.feature.transfer.failure_code field.
   sip_feature_detail_parms.feature_type = FEATURE_TRANSFER;
   // This field is mandatory for the transfer target. It will comprise either the
   // answer in an offer/answer exchange (this is most likely) or an offer if the
   // INVITE with Replaces did not contain an offer.
   sip_feature_detail_parms.feature.transfer.media_offer_answer =
populate_media_answer(sip_detail_parms.media_offer_answer);
   // Send the appropriate SIP message(s)
   rc = sip_feature_send(&sip_feature_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_feature_send(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free the SIP_DETAIL_PARMS structure. This is essential after any call to
   // sip_details. If this function is not called, the application will leak memory.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

In response to the call to `sip_feature_send`, the SIP service will send a 200 OK to the transferee and raise an `EV_MEDIA` to the application now that the new media settings are known. The application should then reconfigure the media stream accordingly.

There is nothing left for the application to do but it should be noted that this is a small amount of SIP signalling remaining for the call transfer scenario to be complete. When the transfer target receives the ACK from the transferee it will clear down the original call by sending a `BYE` to the transferor, which will acknowledge with a 200 OK. As in section 4.6, this will not result in an `EV_IDLE` being raised to the application as the call handle is now associated with a call to the transferee.

5 SIP mid call messages

It is currently possible to send two mid call SIP messages in addition to re-INVITE. They are the INFO and NOTIFY methods. Here we will only describe the INFO method, as the two are essentially identical.

NOTE

By default, where appropriate, Aculab SIP will respond to mid call messages.

Please refer to the enable_midcall_response_mask in the Extended SIP API document for details of which messages can be set to enable the application to respond to them instead.

This would need to be used in conjunction with response_notification_mask, detailed below, to inform the application of the arrival of such messages.

5.1 Sending an INFO request

The application wishes to send an INFO mid call. The application MidCallMessage_UAC with no arguments (use the NOTIFY argument to send NOTIFY instead of INFO) will establish an inbound call and send a mid call INFO.

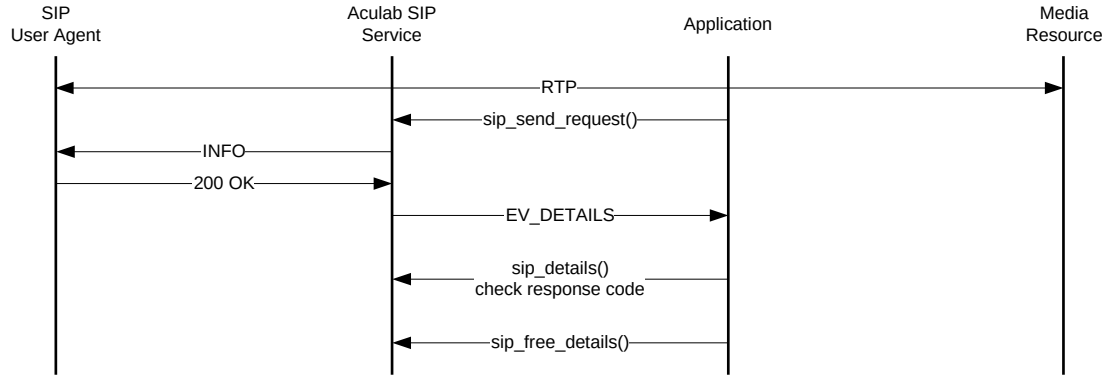

Figure 5-1 Sending a mid call INFO request

Creating a mid call message is very similar to other API calls. The point to really be aware of is that the application will need to be notified of the response. This will not automatically occur and so an additional parameter must be set when calling sip_openin (or sip_openout). In order for an application to be notified of these messages it must open the call in the following way:

Opening a call so that it is notified of INFO responses

```
ACU_ERR open_incoming(ACU_CALL_HANDLE *handle)
{
   ACU_ERR rc = 0;
   SIP_IN_PARMS in;

   // When opening the SIP port for incoming calls using the SIP Bridge it is
   // necessary to use sip_openin instead of call_openin.
   INIT_ACU_CL_STRUCT(&in);
   in.net = settings.sip_port;
   // This application needs to be notified of any response to any mid call
   // message that it will send. EV_DETAILS will be raised on
   // receipt of any such message.
   in.response_notification_mask = ACU_SIP_INFO_NOTIFICATION;
```

```
   rc = sip_openin(&in);
   if(0 != rc)
   {
      printf("Error in sip_openin(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Store the call's handle to pass to the call control loop.
   *handle = in.handle;

   return 0;
}
```

Now that this is set, any response to an `INFO` or `NOTIFY` will be raised to the application via an `EV_DETAILS`. Now that the application will be notified of any response, we need to know how to send the message:

Sending a mid call `INFO` message

```
ACU_ERR send_mid_call_info(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_SEND_REQUEST_PARMS sip_send_request_parms;
   ACU_RAW_MESSAGE_BODY acu_raw_message_body;

   ACU_UCHAR MSG_BODY[] = {
      'T', 'h', 'i', 's', ' ', 'i', 's', ' ', 't', 'h', 'e', ' ', 'm', 'e', 's', 's', 'a', 'g',
      'e', ' ', 'b', 'o', 'd', 'y', ' ', 'o', 'f', ' ', 'a', ' ', 'm', 'i', 'd', ' ', 'c', 'a',
      'l', 'l', ' ', 'S', 'I', 'P', ' ', 'm', 'e', 's', 's', 'a', 'g', 'e', '.', '\n', '\0'};

   // Only the handle & message_type fields are mandatory. Here, for purposes
   // of illustration, all fields are populated.
   INIT_ACU_CL_STRUCT(&sip_send_request_parms);
   sip_send_request_parms.handle = handle;
   sip_send_request_parms.message_type = ACU_SIP_INFO_NOTIFICATION;
   sip_send_request_parms.custom_headers = "Subject: Test INFO message";

   memset(&acu_raw_message_body, 0, sizeof(ACU_RAW_MESSAGE_BODY));
   // The message body defined above is clearly not an ISUP message and is used only for
   // illustrative purposes.
   acu_raw_message_body.body_type = "application/isup";
   acu_raw_message_body.body = (ACU_UCHAR*) MSG_BODY;
   acu_raw_message_body.body_length = sizeof(MSG_BODY);

   sip_send_request_parms.message_bodies = &acu_raw_message_body;

   // Send INFO to UAS.
   rc = sip_send_request(&sip_send_request_parms);
   if(0 != rc)
   {
      printf("Error in sip_send_request(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return 0;
}
```

Now the message has been sent, an EV_DETAILS will be raised when any response is received. The application should call sip_details at this point but beyond that it is free to do as it chooses.

5.2 Receiving an INFO request

The application wishes to be notified of any `INFO` requests that may be sent to it. The application `MidCallMessage_UAS` with the following arguments will run this example:

```
MidCallMessage_UAS r=<REMOTE IP ADDRESS>
```

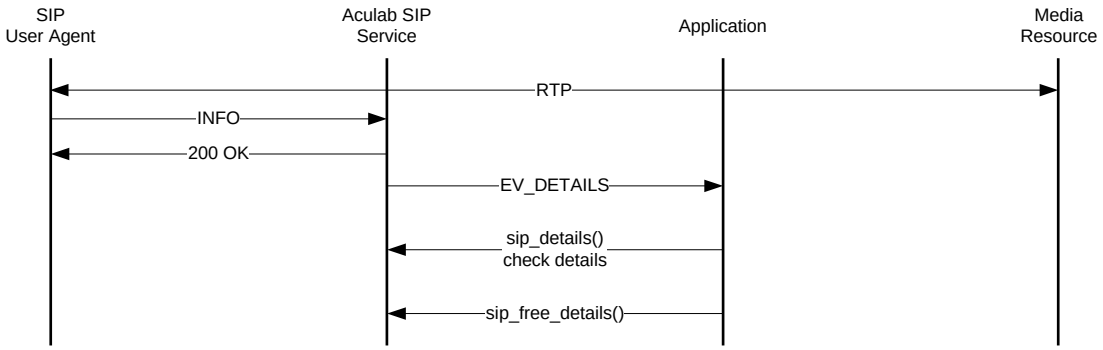

Figure 5-2 Receiving a mid call INFO request

Similarly to the preceding example, the application will not automatically be notified of any mid call `INFO` messages. In order for the application to be notified, it must set an extra field in the `SIP_OPENIN/OUT_PARMS`:

Opening a call so that it is notified of `INFO` requests

```
ACU_ERR open_outgoing(ACU_CALL_HANDLE *handle)
{
   ACU_ERR rc = 0;
   SIP_OUT_PARMS out;

   // When opening the SIP port for incoming calls using the SIP Bridge it is
   // necessary to use sip_openout instead of call_openout.
   INIT_ACU_CL_STRUCT(&out);
   out.net = settings.sip_port;
   out.originating_addr = (ACU_CHAR*)malloc(strlen(settings.local_ip_address) + 1);
   strcpy(out.originating_addr, settings.local_ip_address);
   out.destination_addr = (ACU_CHAR*)malloc(strlen(settings.remote_ip_address) + 1);
   strcpy(out.destination_addr, settings.remote_ip_address);

   // Populate the media offer
   out.media_offer_answer = populate_media_offer(ACU_PCMA_PAYLOAD_NUMBER, 0, 0);

   // The application needs to be notified of any inbound INFO requests. Set this
   // so that an EV_DETAILS willbe raised when any such message is received.
   out.request_notification_mask = ACU_SIP_INFO_NOTIFICATION;

   // Send the INVITE
   rc = sip_openout(&out);
   if(0 != rc)
   {
      printf("Error in sip_openout(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Store the call's handle to pass to the call control loop.
   *handle = out.handle;
```

```
    // There is a large amount of dynamic memory allocation involved when creating
    // a media offer or answer. This memory must be freed.
    free_media_offer(&out.media_offer_answer);

    return rc;
}
```

Therefore, if an application needs to be notified of receipt of mid call requests it assigns the appropriate value to the `request_notification_mask` parameter. If it needs to be notified of responses to such requests, it must set the `response_notification_mask`.

Little else needs to be done in this example. If a mid call `INFO` is received, an `EV_DETAILS` will be raised. At this point, the application should call `sip_details` but it is entirely up to the application how it uses the information contained in the `INFO`. The 200 OK response will be automatically generated by the SIP service.

6 Miscellaneous call flows

6.1 Redirect an incoming call (send a 3xx response)

In this example the application behaves as a redirect server. The application Redirector with the following arguments will run the example:

Redirector r=<REDIRECTED CONTACT 1> r=<REDIRECTED CONTACT 2>...

The 3 dots indicating that many contacts may be supplied. The default 3xx response (302) may be changed with c=301 (or any other response code).

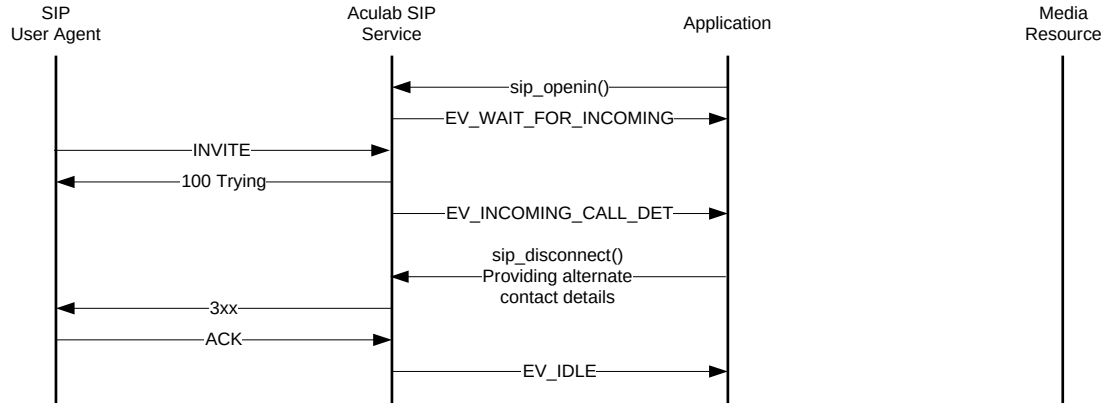

Figure 6.1 – Redirecting an incoming call

As can be seen above, this is a very simple feature to implement. On receipt of an EV_INCOMING_CALL_DET the application will call sip_disconnect providing a list of alternative contact details and a response code. This response will then be sent and the call will idle. No EV_MEDIA will be raised in this scenario and so no media stream will be established.

Redirecting a call:

```
ACU_ERR handle_ev_incoming_call_det(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_DISCONNECT_PARMS sip_disconnect_parms;

   // Normally, an appllication would call sip_incoming_ringing here to send
   // a 180 response back to the caller. Here we wish to send a 3xx response
   // containing redirect information - i.e. alternative contact address(es).
   INIT_ACU_CL_STRUCT(&sip_disconnect_parms);
   sip_disconnect_parms.handle = handle;
   sip_disconnect_parms.sip_code = settings.sip_code; // 3xx
   // See parse_command_line for where the contact list is set up.
   sip_disconnect_parms.redirect_contact_list = settings.redirect_contacts;

   rc = sip_disconnect(&sip_disconnect_parms);
   if(0 != rc)
   {
      printf("Error in sip_disconnect(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // As with the rest of the SIP API, the API structures are populated with char*'s
```

```
    // rather than fixed length arrays. Any dynamically allocated memory must be freed.
    // free_string_list is defined in section 0
    free_string_list(&sip_disconnect_parms.redirect_contact_list);

    return 0;
}
```

The list of alternative contacts is supplied by the command line in this example (unlikely in a real application). The set up of the list can be found in Redirector.c (parse_command_line()).

6.2 Outgoing call redirected (handle a 3xx response)

The application attempts to make an outgoing call but receives a 3xx response indicating that the call is being redirected. The application Redirectee with the following arguments will run this example:

```
Redirectee r=<REMOTE IP ADDRESS>
```

If there is no redirect server or other UA that may be able to redirect INVITEs, the application Redirector may be used instead.

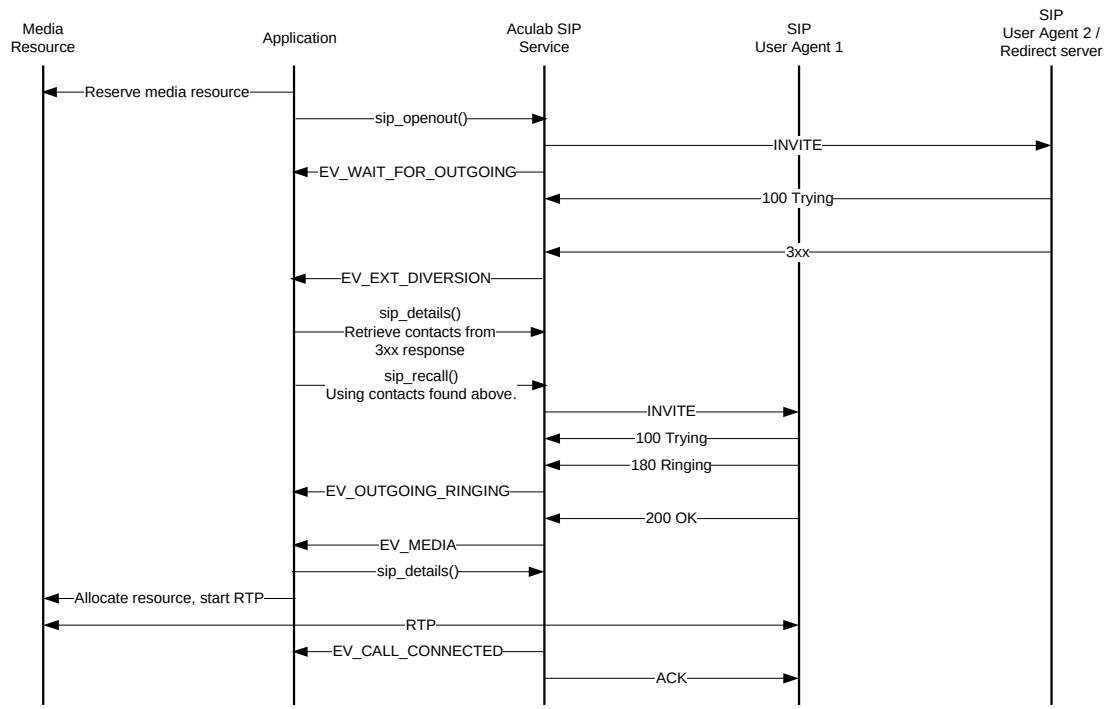

Figure 6.2 – Handling a redirected call

The application makes a call to sip_openout to a UA or SIP server but instead of receiving a 180 response it gets a 3xx response. These category of responses contain alternative contact details for the UAC to try (in the form of one or more Contact headers). In response to this, the SIP service will raise an EV_EXTENDED event with the extended_state field set to EV_EXT_DIVERSION. The application should then call sip_details to retrieve the redirection information from the sip_detail_parms.redirect_info field. The application can then choose one of the available contacts and provide it as input to sip_recall.

Handling a 3xx response:

```
ACU_ERR handle_ev_ext_diversion(const ACU_CALL_HANDLE handle)
{
   ACU_ERR rc = 0;
   SIP_DETAIL_PARMS sip_detail_parms;
   SIP_RECALL_PARMS sip_recall_parms;

   // Retrieve the media settings for the call.
   INIT_ACU_CL_STRUCT(&sip_detail_parms);
   sip_detail_parms.handle = handle;
   rc = sip_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   if(NULL == sip_detail_parms.redirect_info)
   {
       printf("sip_details returned no contact information.\n");
       return -1;
   }

   INIT_ACU_CL_STRUCT(&sip_recall_parms);
   sip_recall_parms.handle = handle;
   // There may be multiple contacts in sip_detail_parms.redirect_info. Here we simply use the first one.
   // There may also be
   sip_recall_parms.destination_addr =
(ACU_CHAR*)malloc(strlen(sip_detail_parms.redirect_info->contact_list->string) + 1);
   strcpy(sip_recall_parms.destination_addr, sip_detail_parms.redirect_info->contact_list->string);
   rc = sip_recall(&sip_recall_parms);
   if(0 != rc)
   {
      printf("Error in sip_recall(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Free the SIP_DETAIL_PARMS structure. This is essential after any call to
   // sip_details. If this function is not called, the application will leak memory.
   rc = sip_free_details(&sip_detail_parms);
   if(0 != rc)
   {
      printf("Error in sip_free_details(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return rc;
}
```

A new (and identical apart from the Cseq and To headers) INVITE will now be sent to the new contact address. The call flow will then be the same as for a normal outgoing call. It could, of course, be redirected again.

7 SIP out of dialog messages

The SIP service provides support for sending and receiving out of dialog SIP messages. The possible message types are as follows:

- NOTIFY
- REGISTER
- SUBSCRIBE
- OPTIONS

There is a considerable variation in the amount of work that would need to be done behind the scenes in order to fully implement these methods, for example, if an application wished to accept REGISTER requests it would essentially be a SIP registrar which would involve a lot more than simply sending a 200 OK to REGISTER requests. Construction of NOTIFY requests and processing of SUBSCRIBE requests would depend entirely on the event notification scheme that is to be implemented. However, in terms of the extended SIP API, the handling of these messages is basically the same – they use the same API calls and the same events are raised. As such, these examples will concentrate only on the OPTIONS method as all UAs should implement this in order to conform to RFC 3261.

7.1 Sending an OPTIONS request

The application wishes to send an OPTIONS request to a remote party. The application OutOfDialog_UAC with the following arguments will run this example:

OutOfDialog_UAC r=<remote IP address>

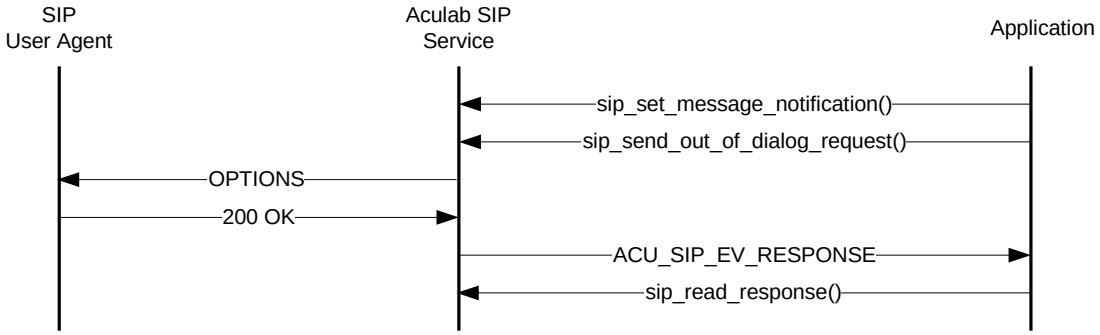

Figure 7.1 – Sending an out of dialog request

Not all applications would necessarily be interested in receiving notification of responses to the out of dialog requests that it sends. For example, an application that is performing event notification functions may not be interested in the contents of responses to NOTIFY requests. In the case of OPTIONS, however, the purpose is to determine the remote party's capabilities so notification of the response is very likely to be required. This does not happen automatically, the application must make a call to sip_set_message_notification() in order to specify which responses the application is interested in. Once this is done, the OPTIONS request may be sent.

Requesting notification of responses:

```
ACU_ERR set_response_notification(const ACU_PORT_ID sip_port)
{
   ACU_ERR rc = ERR_NO_ERROR;
   SIP_MESSAGE_NOTIFICATION_PARMS sip_message_notification_parms;

   INIT_ACU_CL_STRUCT(&sip_message_notification_parms);
   sip_message_notification_parms.port_id = sip_port;
   // This parameter specifies which requests the application is interested in
   // being notified of any response. Here, only the OPTIONS request is of
   // interest.
   sip_message_notification_parms.response_notification_mask = ACU_SIP_OPTIONS_NOTIFICATION;
   rc = sip_set_message_notification(&sip_message_notification_parms);
   if(ERR_NO_ERROR != rc)
   {
      printf("Error in sip_set_message_notification(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   return ERR_NO_ERROR;
}
```

Sending an OPTIONS request:

```
ACU_ERR send_options_request(const ACU_PORT_ID sip_port, ACU_POINTER *transaction_id)
{
   // Send the OPTIONS request. The following fields of sip_send_out_of_dialog_request_parms
   // are optional and are only described here:
   //
   // request_uri - Request-URI to be used for this request. If not present, the 'to' field is used
   // local_address - reserved for future use
   // custom_headers - A CRLF delimited list of additional SIP headers to be included in the request
   // message_bodies - A list of message bodies to be included in the request
   ACU_ERR rc = ERR_NO_ERROR;
   SIP_SEND_OUT_OF_DIALOG_REQUEST_PARMS sip_send_out_of_dialog_request_parms;

   INIT_ACU_CL_STRUCT(&sip_send_out_of_dialog_request_parms);
   sip_send_out_of_dialog_request_parms.message_type = ACU_SIP_MESSAGE_OPTIONS;
   // Using dummy IP addresses for the purposes of these examples
   sip_send_out_of_dialog_request_parms.from = "sip:192.168.100.100";
   sip_send_out_of_dialog_request_parms.to = "sip:192.168.100.101";

   rc = sip_send_out_of_dialog_request(&sip_send_out_of_dialog_request_parms);
   if(ERR_NO_ERROR != rc)
   {
      printf("Error in sip_send_out_of_dialog_request(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Store the transaction ID so the response may be matched to the request
   *transaction_id = sip_send_out_of_dialog_request_parms.transaction_id;

   return ERR_NO_ERROR;
}
```

Now the OPTIONS request has been sent, the SIP service will raise an

ACU_SIP_EV_RESPONSE when the response has been received. At this point, the application should retrieve the response using sip_read_response(). No discussion of processing the response will be given as that is a matter for the application's own local policy. Here, it will simply be shown how to retrieve the response from the SIP service. It is assumed that the application has set up the relevant event queue and an event has been raised. For more information on setting up event queues and waiting for events, see the V6 resource management API guide (acu_allocate_event_queue() & acu_get_event_from_queue()) and the V6 call control API guide (call_set_port_notification_queue()).

Reading a SIP response:

```
ACU_ERR handle_port_event(const ACU_PORT_ID sip_port, const ACU_POINTER transaction_id)
{
   ACU_ERR rc = ERR_NO_ERROR;
   CALL_PORT_NOTIFICATION_PARMS call_port_notification_parms;
   SIP_READ_MESSAGE_PARMS sip_read_message_parms;

   // Retrieve the event
   INIT_ACU_CL_STRUCT(&call_port_notification_parms);
   call_port_notification_parms.port_id = sip_port;
   rc = call_get_port_notification(&call_port_notification_parms);
   if(ERR_NO_ERROR != rc)
   {
      printf("Error in call_get_port_notification(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Check the event type
   switch(call_port_notification_parms.event)
   {
   case ACU_SIP_EV_RESPONSE:
      INIT_ACU_CL_STRUCT(&sip_read_message_parms);
      sip_read_message_parms.port_id = sip_port;
      rc = sip_read_response(&sip_read_message_parms);
      if(ERR_NO_ERROR != rc)
      {
         printf("Error in sip_read_response(): %d, %s\n", rc, error_2_string(rc));
      }

      // Check the transaction ID matches that returned by sip_send_out_of_dialog_request()
      // In these examples there is no real need as only one request has been sent but
      // in general, an application would need to match requests with responses.
      if(sip_read_message_parms.transaction_id != transaction_id)
      {
         printf("Error. Transaction ID does not match request.\n");
         rc = -1;
      }
      break;
   case ACU_SIP_EV_REQUEST:
      // In these examples, this would indicate a serious error as
      // notifications of requests has not been set. In general this
      // should be handled. See section 7.2.
      rc = -1;
      break;
   case ACU_SIP_EV_REQUEST_TIMEOUT:
```

```
      // The OPTIONS request has received no response. Here an application would
      // call sip_read_timeout() in order to retrieve the transaction ID of the request.
      // In these examples, it is known as only one request has been sent.
      break;
   default:
      break;
   }

   return rc;
}
```

The response to an OPTIONS request will contain information about the capabilities of the remote user agent. It is up to the individual application as to what it does with that information.

7.2 Receiving an OPTIONS request

The application wishes to process incoming OPTIONS requests and send appropriate responses. The application OutOfDialog_UAS will run this example.

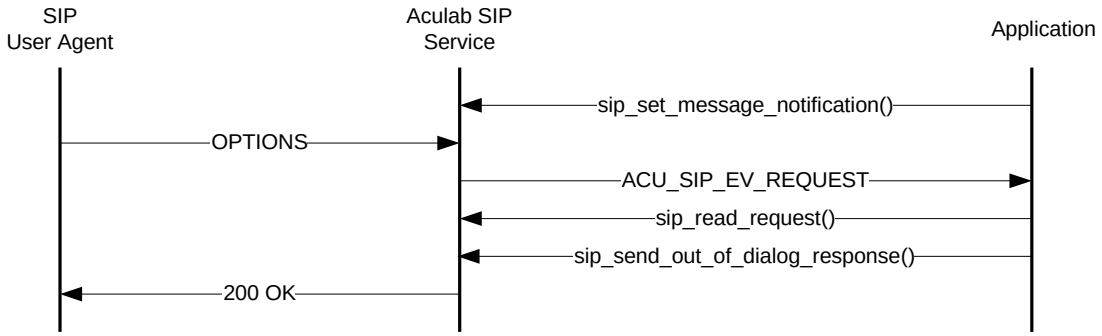

Figure 7.2 Receiving an out of dialog request

The application will need to request notification of incoming OPTIONS messages. This is done in exactly the same manner as it is in section 7.1 except that:

```
sip_message_notification_parms.response_notification_mask =
ACU_SIP_OPTIONS_NOTIFICATION;
```

is replaced by:

```
sip_message_notification_parms.request_notification_mask =
ACU_SIP_OPTIONS_NOTIFICATION;
```

and:

```
sip_message_notification_parms.enable_response_mask =
ACU_SIP_OPTIONS_NOTIFICATION;
```

is added. This additional parameter specifies that the application would like to create and send the response itself. If it is not set, the SIP service will create a default response. In the case of OPTIONS, the SIP service can only create a very limited response as it has no knowledge of the capabilities of the particular media server in use.

The application now needs to call sip_set_message_notification as in example 7.2. It is entirely possible that an application would wish to receive notification of both requests and responses, in which case, both request and response masks would need to be set.

Reading a SIP request:

It is assumed that an OPTIONS request has been received and an event has been

raised to acu_get_event_from_queue. See the V6 resource management API guide for more details on event queues.

```
ACU_ERR handle_port_event(const ACU_PORT_ID sip_port)
{
   ACU_ERR rc = ERR_NO_ERROR;
   CALL_PORT_NOTIFICATION_PARMS call_port_notification_parms;
   SIP_READ_MESSAGE_PARMS sip_read_message_parms;

   // Retrieve the event
   INIT_ACU_CL_STRUCT(&call_port_notification_parms);
   call_port_notification_parms.port_id = sip_port;
   rc = call_get_port_notification(&call_port_notification_parms);
   if(ERR_NO_ERROR != rc)
   {
      printf("Error in call_get_port_notification(): %d, %s\n", rc, error_2_string(rc));
      return rc;
   }

   // Check the event type
   switch(call_port_notification_parms.event)
   {
   case ACU_SIP_EV_RESPONSE:
      // In these examples, this would indicate a serious error as
      // notifications of responses has not been set. In general this
      // should be handled. See section 7.1.
      rc = -1;
      break;
   case ACU_SIP_EV_REQUEST:
      INIT_ACU_CL_STRUCT(&sip_read_message_parms);
      sip_read_message_parms.port_id = sip_port;
      rc = sip_read_request(&sip_read_message_parms);
      if(ERR_NO_ERROR != rc)
      {
         printf("Error in sip_read_response(): %d, %s\n", rc, error_2_string(rc));
      }

      // Usually, an application would need to parse the message in order to determine
      // the message type. Here, only notification of OPTIONS messages has been
      // requested so it is known to be an OPTIONS. For OPTIONS, no further processing
      // is necessary.
      rc = send_options_response(sip_port, sip_read_message_parms.transaction_id); // See below.
      break;
   case ACU_SIP_EV_REQUEST_TIMEOUT:
      // In these examples, this would indicate a serious error as timeouts
      // only occur for requests. In general this would indicate a previously
      // sent request has timed out. See section 7.1
      break;
   default:
      break;
   }

   return rc;
}
```

Sending an OPTIONS response:

```
ACU_ERR send_options_response(const ACU_PORT_ID sip_port, const ACU_POINTER transaction_id)
{
   ACU_ERR rc = ERR_NO_ERROR;
   SIP_SEND_OUT_OF_DIALOG_RESPONSE_PARMS sip_send_out_of_dialog_response_parms;

   // This is an example SDP body containing much of what Aculab supports in terms
   // of media capablities. In general, an application would only offer a subset
   // of this. The actual capabilities would need to be ascertained from the particular
   // hardware or software that is being used (e.g. Prosody X, Prosody S...) and the
   // specific needs of the application.
   // Note. This is not necessary for all out of dialog requests. Some responses may
   // require a message body (which may or may not be SDP) and others will not. This will
   // depend on the application. Most will require additional processing according to
   // the application's needs.
   static const char* SDP_BODY =
      "v=0\r\n"
      "o=Aculab 0 0 IN IP4 192.168.1.2\r\n"
      "s=Example SDP body\r\n"
      "c=IN IP4 192.168.1.2\r\n"
      "t=0 0\r\n"
      "m=audio 0 RTP/AVP 18 8 4 0 96\r\n"
      "a=rtpmap:18 G729/8000\r\n"
      "a=fmtp:18 annexb=no\r\n"
      "a=rtpmap:8 PCMA/8000\r\n"
      "a=rtpmap:4 g723/8000\r\n"
      "a=rtpmap:0 PCMU/8000\r\n"
      "a=rtpmap:96 telephone-event/8000\r\n"
      "a=fmtp:96 0-15"
      "m=video 0 RTP/AVP 34\r\n"
      "a=rtpmap:34 H263/90000\r\n"
      "a=fmtp:34 CIF=1 QCIF=1 SQCIF=1 MaxBR=10240\r\n"
      "m=image 1211 UDPTL t38\r\n"
      "a=T38MaxBitRate:14400\r\n"
      "a=T38FaxMaxBuffer:1024\r\n"
      "a=T38FaxUDPEC:t38UDPRedundancy\r\n"
      "a=T38FaxVersion:2\r\n"
      "a=T38FaxRateManagement:transferredTCF\r\n"
      "a=T38FaxMaxDatagram:160\r\n";

   // Add other assorted SIP headers. Again, these will vary according to the
   // application's needs.
   static const char* ADDITIONAL_OPTIONS_HEADERS =
      "Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, NOTIFY, REFER, PRACK, INFO\r\n"
      "Supported: 100rel, timer, replaces\r\n"
      "Accept: application/sdp, application/isup, application/qsig, multipart/mixed\r\n"
      "Accept-Encoding: identity\r\n"
      "Accept-Language: en\r\n";

   INIT_ACU_CL_STRUCT(&sip_send_out_of_dialog_response_parms);
   sip_send_out_of_dialog_response_parms.port_id = sip_port;
   sip_send_out_of_dialog_response_parms.transaction_id = transaction_id;
   // The application may wish to reject the request in which case a suitable SIP response code must be used.
   sip_send_out_of_dialog_response_parms.sip_code = 200;
```

```
    sip_send_out_of_dialog_response_parms.message_bodies =
                                                    (ACU_RAW_MESSAGE_BODY*)malloc(sizeof(ACU_RAW_MESSAGE_BODY));
    memset(sip_send_out_of_dialog_response_parms.message_bodies, 0, sizeof(ACU_RAW_MESSAGE_BODY));
    sip_send_out_of_dialog_response_parms.message_bodies->body_type = "application/sdp";
    sip_send_out_of_dialog_response_parms.message_bodies->body_length = strlen(SDP_BODY);
    sip_send_out_of_dialog_response_parms.message_bodies->body = (unsigned char*)SDP_BODY;

    sip_send_out_of_dialog_response_parms.custom_headers = (char*)ADDITIONAL_OPTIONS_HEADERS;

    rc = sip_send_out_of_dialog_response(&sip_send_out_of_dialog_response_parms);
    if(ERR_NO_ERROR != rc)
    {
       printf("Error in sip_send_out_of_dialog_response(): %d, %s\n", rc, error_2_string(rc));
       return rc;
    }

    free(sip_send_out_of_dialog_response_parms.message_bodies);

    return ERR_NO_ERROR;
}
```

Appendix A: Handling dynamic memory allocation

Most commonly, SIP messages contain SDP bodies describing the media session. These message bodies are presented to the application in the structure ACU_MEDIA_OFFER_ANSWER. Due to the nature of SDP, this structure contains several linked lists and many strings whose length will vary from message to message. This means that the application will need to handle a large amount of dynamic memory allocation and subsequent releasing of that memory. These functions are intended to make this process straight forward.

A.1 clone_media_offer

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   clone_media_offer()                                                   |
|                                                                         |
|   Copy all elements of a media offer for later use.                     |
|                                                                         |
\-------------------------------------------------------------------------*/
ACU_MEDIA_OFFER_ANSWER *clone_media_offer(const ACU_MEDIA_OFFER_ANSWER* offer)
{
   ACU_MEDIA_OFFER_ANSWER *temp;

   if(NULL == offer)
   {
      // No media offer present
      return NULL;
   }

   temp = (ACU_MEDIA_OFFER_ANSWER*)malloc(sizeof(ACU_MEDIA_OFFER_ANSWER));
   memset(temp, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));

   // Copy connection address information
   temp->connection_address.address_type = offer->connection_address.address_type;
   if(NULL != offer->connection_address.address)
   {
      temp->connection_address.address = (ACU_CHAR*)malloc(strlen(offer->connection_address.address) + 1);
      strcpy(temp->connection_address.address, offer->connection_address.address);
   }

   // Copy raw SDP
   if(NULL != offer->raw_sdp)
   {
      temp->raw_sdp = (ACU_CHAR*)malloc(strlen(offer->raw_sdp) + 1);
      strcpy(temp->raw_sdp, offer->raw_sdp);
   }

   // clone_media_descriptions is defined in section 0
   temp->media_descriptions = clone_media_descriptions(offer->media_descriptions);

   return temp;
}
```

A.2 clone_media_descriptions

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   clone_media_descriptions()                                            |
|                                                                         |
|   Copy all media descriptions for a given media offer.                  |
|                                                                         |
\-------------------------------------------------------------------------*/
ACU_MEDIA_DESCRIPTION *clone_media_descriptions(ACU_MEDIA_DESCRIPTION *media_descriptions)
{
   ACU_MEDIA_DESCRIPTION *new_desc;

   if(NULL == media_descriptions)
   {
      // There is no media description to copy
      return NULL;
   }

   // Although there may be several media descriptions this function is
   // recursive so only enough memory for one media description needs to
   // be allocated here.
   new_desc = (ACU_MEDIA_DESCRIPTION*)malloc(sizeof(ACU_MEDIA_DESCRIPTION));
   memset(new_desc, 0, sizeof(ACU_MEDIA_DESCRIPTION));

   if(NULL != media_descriptions->next)
   {
      // The next media description in the list is not NULL
      // so it needs to be copied first.
      new_desc->next = clone_media_descriptions(media_descriptions->next);
   }

   // Now copy all of the media attributes contained in the media description
   new_desc->connection_address.address_type = media_descriptions->connection_address.address_type;
   if(NULL != media_descriptions->connection_address.address)
   {
      new_desc->connection_address.address = (ACU_CHAR*)malloc(MAXADDR);
      strcpy(new_desc->connection_address.address, media_descriptions->connection_address.address);
   }
   if(NULL != media_descriptions->transport)
   {
      new_desc->transport = (ACU_CHAR*)malloc(strlen(media_descriptions->transport) + 1);
      strcpy(new_desc->transport, media_descriptions->transport);
   }
   new_desc->packet_length    = media_descriptions->packet_length;
   new_desc->media_direction  = media_descriptions->media_direction;
   new_desc->media_type       = media_descriptions->media_type;
   new_desc->port             = media_descriptions->port;

   // Copy the payloads given in this media description. clone_payloads is defined in section 0
   new_desc->payloads = clone_payloads(media_descriptions->payloads, media_descriptions->media_type);

   // Copy any miscellaneous attributes. clone_misc_attributes is defined in section 0
   new_desc->miscellaneous_attributes = clone_misc_attributes(media_descriptions->miscellaneous_attributes);
```

```
   return new_desc;
}
```

A.3 clone_payloads

```
/*-------------------------------------------------------------------------------\
|                                                                               |
|   clone_payloads()                                                            |
|                                                                               |
|   Check media type and copy all payloads in the list.                         |
|                                                                               |
\-------------------------------------------------------------------------------*/
ACU_PAYLOAD *clone_payloads(const ACU_PAYLOAD *payloads, const ACU_MEDIA_TYPES media_type)
{
   ACU_PAYLOAD *new_payload = NULL;

   if(NULL == payloads)
   {
      return NULL;
   }

   switch(media_type)
   {
   case ACU_AUDIO:
   case ACU_VIDEO:
      // clone_audio_video_payloads is defined in section A.4
      new_payload = clone_audio_video_payloads(payloads);
      break;
   case ACU_IMAGE:
      // clone_image_payloads is defined in section 0
      new_payload = clone_image_payloads(payloads);
      break;
   case ACU_CONTROL:
      // clone_control_payloads is defined in section 0
      new_payload = clone_control_payloads(payloads);
      break;
   case ACU_TEXT:
   case ACU_APPLICATION:
   default:
      // Nothing to copy
      break;
   }

   return new_payload;
}
```

A.4 clone_audio_video_payloads

```
/*-------------------------------------------------------------------------\
|                                                                          |
|   clone_audio_video_payloads()                                           |
|                                                                          |
|   Copy payloads for media types ACU_AUDIO and ACU_VIDEO                  |
|                                                                          |
\-------------------------------------------------------------------------*/
ACU_PAYLOAD *clone_audio_video_payloads(const ACU_PAYLOAD *payloads)
{
   ACU_PAYLOAD *new_payload = NULL;

   // Although there may be several payloads this function is recursive
   // so only enough memory for one payload needs to be allocated here.
   new_payload = (ACU_PAYLOAD*)malloc(sizeof(ACU_PAYLOAD));
   memset(new_payload, 0, sizeof(ACU_PAYLOAD));

   if(NULL != payloads->next)
   {
      // The next payload in the list is not NULL so it needs to be copied first.
      new_payload->next = clone_audio_video_payloads(payloads->next);
   }

   // Copy payload attributes
   new_payload->payload.audio_video.clock_rate     = payloads->payload.audio_video.clock_rate;
   new_payload->payload.audio_video.packet_length  = payloads->payload.audio_video. packet_length;
   if(NULL != payloads->payload.audio_video.payload_specific_options)
   {
      ACU_CHAR *options;
      options = (ACU_CHAR*)malloc(strlen(payloads->payload.audio_video.payload_specific_options)+1);
      strcpy(options, payloads->payload.audio_video.payload_specific_options);
      new_payload->payload.audio_video.payload_specific_options = options;
   }

   if(NULL != payloads->payload.audio_video.rtp_payload_name)
   {
      ACU_CHAR *payload_name;
      payload_name = (ACU_CHAR *)malloc(strlen(payloads->payload.audio_video.rtp_payload_name) + 1);
      strcpy(payload_name, payloads->payload.audio_video.rtp_payload_name);
      new_payload->payload.audio_video.rtp_payload_name = payload_name;
   }
   new_payload->payload.audio_video.rtp_payload_number = payloads->payload.audio_video.rtp_payload_number;

   return new_payload;
}
```

A.5 clone_image_payloads

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   clone_image_payloads()                                                |
|                                                                         |
|   Copy payloads for media type ACU_IMAGE                                |
|                                                                         |
\-------------------------------------------------------------------------*/
ACU_PAYLOAD *clone_image_payloads(const ACU_PAYLOAD *payloads)
{
   ACU_PAYLOAD *new_payload = NULL;

   // Although there may be several payloads this function is recursive
   // so only enough memory for one payload needs to be allocated here.
   new_payload = (ACU_PAYLOAD*)malloc(sizeof(ACU_PAYLOAD));
   memset(new_payload, 0, sizeof(ACU_PAYLOAD));

   if(NULL != payloads->next)
   {
      // The next payload in the list is not NULL so it needs to be copied first.
      new_payload->next = clone_image_payloads(payloads->next);
   }

   // Copy payload attributes
   if(NULL != payloads->payload.image.image_payload_name)
   {
      ACU_CHAR* image_payload_name;
      image_payload_name = (ACU_CHAR*)malloc(strlen(payloads->payload.image.image_payload_name) + 1);
      strcpy(image_payload_name, payloads->payload.image.image_payload_name);
      new_payload->payload.image.image_payload_name = image_payload_name;
   }

   return new_payload;
}
```

A.6 clone_control_payloads

```
/*-------------------------------------------------------------------------------\
|                                                                               |
|   clone_control_payloads()                                                     |
|                                                                               |
|   Copy payloads for media type ACU_CONTROL                                     |
|                                                                               |
\-------------------------------------------------------------------------------*/
ACU_PAYLOAD *clone_control_payloads(const ACU_PAYLOAD *payloads)
{
   ACU_PAYLOAD *new_payload = NULL;

   // Although there may be several payloads this function is recursive
   // so only enough memory for one payload needs to be allocated here.
   new_payload = (ACU_PAYLOAD*)malloc(sizeof(ACU_PAYLOAD));
   memset(new_payload, 0, sizeof(ACU_PAYLOAD));

   if(NULL != payloads->next)
   {
      // The next payload in the list is not NULL so it needs to be copied first.
      new_payload->next = clone_control_payloads(payloads->next);
   }

   // Copy payload attributes
   new_payload->payload.control.dummy = payloads->payload.control.dummy;

   return new_payload;
}
```

A.7 clone_misc_attributes

```
/*-------------------------------------------------------------------------\
|                                                                          |
|   clone_misc_attributes()                                                |
|                                                                          |
|   Copy ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE linked list                     |
|                                                                          |
\-------------------------------------------------------------------------*/
ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE *clone_misc_attributes(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE *misc_attributes)
{
   ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE *new_attribute;

   if(NULL == misc_attributes)
   {
      // There are no miscellaneous attributes
      return NULL;
   }

   // Although there may be several attributes this function is recursive
   // so only enough memory for one attribute needs to be allocated here.
   new_attribute = (ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE*)malloc(sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
   memset(new_attribute, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));

   if(NULL != misc_attributes->next)
   {
      // The next attribute in the list is not NULL so it needs to be copied first.
      new_attribute->next = clone_misc_attributes(misc_attributes->next);
   }

   // Copy miscellaneous attribute
   if(NULL != misc_attributes->attribute)
   {
      new_attribute->attribute = (ACU_CHAR*)malloc(strlen(misc_attributes->attribute) + 1);
      strcpy(new_attribute->attribute, misc_attributes->attribute);
   }

   return new_attribute;
}
```

A.8 free_media_offer

Use this function to free any dynamically allocated ACU_MEDIA_OFFER_ANSWER structures.

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   free_media_offer()                                                    |
|                                                                         |
|   Free all dynamically allocated memory in a ACU_MEDIA_OFFER_ANSWER structure |
|                                                                         |
\-------------------------------------------------------------------------*/
void free_media_offer(ACU_MEDIA_OFFER_ANSWER** stored_offer)
{
   // Free contents of global connection address
   free((*stored_offer)->connection_address.address);
   (*stored_offer)->connection_address.address = NULL;

   // Free the raw SDP char* buffer
   free((*stored_offer)->raw_sdp);
   (*stored_offer)->raw_sdp = NULL;

   // Free any media descriptions. free_media_descriptions is defined in section 0
   free_media_descriptions(&((*stored_offer)->media_descriptions));

   // Free the ACU_MEDIA_OFFER_ANSWER structure
   free(*stored_offer);
   *stored_offer = NULL;
}
```

A.9 free_media_descriptions

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   free_media_descriptions()                                             |
|                                                                         |
|   Free all dynamically allocated memory in a media description linked list    |
|                                                                         |
\-------------------------------------------------------------------------*/
void free_media_descriptions(ACU_MEDIA_DESCRIPTION **media_descriptions)
{
   if(NULL == *media_descriptions)
   {
      // There is no media description to free
      return;
   }

   if(NULL != (*media_descriptions)->next)
   {
      // The next media description in the list is not NULL so needs to be freed first.
      free_media_descriptions(&((*media_descriptions)->next));
   }

   // Free the local connection address
   free((*media_descriptions)->connection_address.address);
   (*media_descriptions)->connection_address.address = NULL;

   // Free any miscellaneous attributes. free_misc_attributes is defined in section 0
   free_misc_attributes(&((*media_descriptions)->miscellaneous_attributes));

   // Free all payloads. free_payloads is defined in section 0
   free_payloads(&((*media_descriptions)->payloads), (*media_descriptions)->media_type);

   // Free the transport char* buffer
   free((*media_descriptions)->transport);
   (*media_descriptions)->transport = NULL;

   // Free the ACU_MEDIA_DESCRIPTION structure
   free(*media_descriptions);
   *media_descriptions = NULL;
}
```

A.10 free_payloads

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   free_payloads()                                                       |
|                                                                         |
|   Check media type and free corresponding payload structures            |
|                                                                         |
\-------------------------------------------------------------------------*/
void free_payloads(ACU_PAYLOAD **payloads, const ACU_MEDIA_TYPES media_type)
{
   switch(media_type)
   {
   case ACU_AUDIO:
   case ACU_VIDEO:
      // free_audio_video_payloads is defined in section 0
      free_audio_video_payloads(payloads);
      break;
   case ACU_IMAGE:
      // free image_payloads is defined in section 0
      free_image_payloads(payloads);
      break;
   case ACU_CONTROL:
   case ACU_TEXT:
   case ACU_APPLICATION:
      // No dynamically allocated memory here.
      // Only need to free the pointer to the payload.
      free(*payloads);
      *payloads = NULL;
      break;
   default:
      break;
   }
}
```

A.11 free_audio_video_payloads

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   free_audio_video_payloads()                                           |
|                                                                         |
|   Free payloads for media types ACU_AUDIO and ACU_VIDEO                 |
|                                                                         |
\-------------------------------------------------------------------------*/
void free_audio_video_payloads(ACU_PAYLOAD **payloads)
{
   if(NULL == *payloads)
   {
      // There are no payloads to free
      return;
   }

   if(NULL != (*payloads)->next)
   {
      // The next payload in the list is not NULL so needs to be freed first.
      free_audio_video_payloads(&((*payloads)->next));
   }

   // Free any dynamically allocated elements of the ACU_PAYLOAD.audio_video structure
   free((*payloads)->payload.audio_video.rtp_payload_name);
   (*payloads)->payload.audio_video.rtp_payload_name = NULL;
   free((*payloads)->payload.audio_video.payload_specific_options);
   (*payloads)->payload.audio_video.payload_specific_options = NULL;
   free(*payloads);
   *payloads = NULL;
}
```

A.12 free_image_payloads

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   free_image_payloads()                                                 |
|                                                                         |
|   Free payloads for media type ACU_IMAGE                                |
|                                                                         |
\-------------------------------------------------------------------------*/
void free_image_payloads(ACU_PAYLOAD **payloads)
{
   if(NULL == *payloads)
   {
      // There are no payloads to free
      return;
   }

   if(NULL != (*payloads)->next)
   {
      // The next payload in the list is not NULL so needs to be freed first.
      free_image_payloads(&((*payloads)->next));
   }

   // Free any dynamically allocated elements of the ACU_PAYLOAD.image structure
   free((*payloads)->payload.image.image_payload_name);
   (*payloads)->payload.image.image_payload_name = NULL;
   free(*payloads);
   *payloads = NULL;
}
```

A.13 free_misc_attributes

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   free_misc_attributes()                                                |
|                                                                         |
|   Free any ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE structures                 |
|                                                                         |
\-------------------------------------------------------------------------*/
void free_misc_attributes(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE **misc_attributes)
{
   if(NULL == *misc_attributes)
   {
      // There are no attributes to free
      return;
   }

   if(NULL != (*misc_attributes)->next)
   {
      // The next attribute in the list is not NULL so needs to be freed first.
      free_misc_attributes(&((*misc_attributes)->next));
   }

   // Free other elements of the ACU_MISCELLANEOUS_ATTRIBUTE structure
   free((*misc_attributes)->attribute);
   (*misc_attributes)->attribute = NULL;
   free(*misc_attributes);
   *misc_attributes = NULL;
}
```

A.14 free_string_list

This function is only used in `Redirector.exe` but will be useful for any application that needs to handle 3xx responses.

```
void free_string_list(ACU_STRING_LIST** list)
{
   if(NULL == *list)
   {
      return;
   }

   if(NULL != (*list)->next)
   {
      free_string_list(&((*list)->next));
   }

   // Free the current ACU_STRING_LIST structure
   free((*list)->string);
   (*list)->string = NULL;
   free(*list);
   *list = NULL;
}
```

Appendix B: Miscellaneous helper functions

These functions are used in the example code provided with this guide. They are intended as illustrative and will not be suitable for all applications.

B.1 populate_media_offer

```
/*-------------------------------------------------------------------------\
|                                                                          |
|   populate_media_offer()                                                 |
|                                                                          |
|   For the purposes of these examples any media offer made by the application  |
|   will contain just the one codec and a telpehone-event. The codec used will  |
|   be G.711 as support for this is mandatory for all SIP User Agents.          |
|                                                                          |
\-------------------------------------------------------------------------*/
ACU_MEDIA_OFFER_ANSWER *populate_media_offer(void)
{
   ACU_MEDIA_OFFER_ANSWER *offer = (ACU_MEDIA_OFFER_ANSWER*)malloc(sizeof(ACU_MEDIA_OFFER_ANSWER));
   ACU_PAYLOAD *codec, *telephone_event;

   // Set the global connection address details. This may be superceded by the local
   // connection address(es) given in each media description. Here we use the global address.
   memset(offer, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));
   offer->connection_address.address = (char*)malloc(strlen(LOCAL_CONNECTION_ADDRESS) + 1);
   strcpy(offer->connection_address.address, LOCAL_CONNECTION_ADDRESS);
   offer->connection_address.address_type = ACU_IPv4;

   // Set just one media description. Again, there may be more.
   offer->media_descriptions = (ACU_MEDIA_DESCRIPTION*)malloc(sizeof(ACU_MEDIA_DESCRIPTION));
   memset(offer->media_descriptions, 0, sizeof(ACU_MEDIA_DESCRIPTION));
   offer->media_descriptions->media_direction   = ACU_SEND_RECV;
   offer->media_descriptions->media_type        = ACU_AUDIO;
   offer->media_descriptions->port              = LOCAL_RTP_PORT;  // Usually provided by the media server

   // Usually an application will present a choice of codecs to the far end.
   // Here we use G.711 as support for it is mandatory.
   codec = (ACU_PAYLOAD*)malloc(sizeof(ACU_PAYLOAD));
   memset(codec, 0, sizeof(ACU_PAYLOAD));
   codec->payload.audio_video.clock_rate           = ACU_EIGHT_K;
   codec->payload.audio_video.rtp_payload_number   = ACU_PCMA_PAYLOAD_NUMBER;
   codec->payload.audio_video.rtp_payload_name     = (ACU_CHAR*)malloc(strlen(ACU_PCMA) + 1);
   strcpy(codec->payload.audio_video.rtp_payload_name, ACU_PCMA);

   // Add a telephone event.
   telephone_event = (ACU_PAYLOAD*)malloc(sizeof(ACU_PAYLOAD));
   memset(telephone_event, 0, sizeof(ACU_PAYLOAD));
   telephone_event->payload.audio_video.clock_rate               = ACU_EIGHT_K;
   telephone_event->payload.audio_video.rtp_payload_number       = 101;
   telephone_event->payload.audio_video.payload_specific_options  = (ACU_CHAR*)malloc(strlen("0-15") + 1);
   strcpy(telephone_event->payload.audio_video.payload_specific_options, "0-15");
   telephone_event->payload.audio_video.rtp_payload_name         = (ACU_CHAR*)malloc(strlen(ACU_TELEPHONE_EVENT));
   strcpy(telephone_event->payload.audio_video.rtp_payload_name, ACU_TELEPHONE_EVENT);

   codec->next = telephone_event;
```

```
   offer->media_descriptions->payloads = codec;

   return offer;
}
```

B.2 populate_media_answer

This function will populate the media answer to be used in `sip_accept` or `sip_incoming_ringing`. It will iterate through all of the media descriptions (each one corresponding to a single media stream) and accept or reject each stream based on an arbitrary list of payload types corresponding to the media type of the particular stream.

In a real application, the constant LOCAL_CONNECTION_ADDRESS would be supplied by the media server, as would the local connection port.

```
/*-------------------------------------------------------------------------\
|                                                                          |
|   populate_media_answer()                                                |
|                                                                          |
|   Whether or not a particular media stream is accepted by the application is  |
|   down to local policy and/or whether a particular codec (or modem etc.) is   |
|   supported by the media server being used.  Here, each media stream will be  |
|   tested against an arbitrary set of media and accepted or declined on that   |
|   basis alone. Previous versions of this guide assumed one audio stream only. |
|                                                                          |
\-------------------------------------------------------------------------*/
ACU_MEDIA_OFFER_ANSWER *populate_media_answer(ACU_MEDIA_OFFER_ANSWER* stored_offer)
{
   // Copy the original offer. This will be modified to indicate which media
   // settings to use and the connection information relating to this end point
   ACU_MEDIA_OFFER_ANSWER *answer = clone_media_offer(stored_offer);
   ACU_INT found_valid_answer = 0;

   answer->connection_address.address_type = ACU_IPv4;
   if(answer->connection_address.address)
   {
      // We do not want to use the remote connection details.
      free(answer->connection_address.address);
   }
   // These examples will use only the session level connection address. If there are multiple
   // media streams to be accepted and these streams are to be processed at differing IP addresses
   // the application would need to populate the connection_address structures in each element of
   // the ACU_MEDIA_DESCRIPTION linked list.
   answer->connection_address.address = (char*)malloc(strlen(LOCAL_CONNECTION_ADDRESS) + 1);
   strcpy(answer->connection_address.address, LOCAL_CONNECTION_ADDRESS);

   // We do not want to use raw SDP here.
   if(answer->raw_sdp)
   {
      free(answer->raw_sdp);
      answer->raw_sdp = 0;
   }

   // Iterate through each media description and return true if a valid answer has been found
   found_valid_answer = modify_media_descriptions(answer->media_descriptions);
```

```
    if(!found_valid_answer)
    {
       // No valid answer has been found. Free any remaining dynamically allocated memory.
       free_media_offer(&answer);
       answer = 0;
    }

    return answer;
}
```

B.3 modify_media_descriptions

To illustrate the use of the ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE structure, this application focuses on the ‘a=crypto’ attribute used with secure RTP. This, and the following SRTP helper functions, uses the following structure to store SRTP information:

```
typedef struct _SRTP_ATTRIBUTE
{
   // a=crypto:<tag> <crypto suite> inline:<key param>
   ACU_UINT tag;
   ACU_UINT tag_length;  // defined by the crypto suite
   ACU_CHAR crypto_suite[32];
   ACU_CHAR remote_key_param[41];
   ACU_CHAR local_key_param[41];
} SRTP_ATTRIBUTE;
```

There are other optional parameters in the ‘a=crypto’ attribute. These examples do not cover their use.

```
/*-------------------------------------------------------------------------\
|                                                                          |
|   modify_media_descriptions()                                            |
|                                                                          |
|   This function takes an ACU_MEDIA_DESCRIPTION provided in an offer and  |
|   modifies it for use as an answer. The various media types offered will be |
|   checked against an arbitrary list and accepted or declined accordingly. |
|   The function recurses through all the media_descriptions in the linked list.|
|   Returns true if a single valid stream is found                         |
|                                                                          |
\-------------------------------------------------------------------------*/
ACU_INT modify_media_descriptions(ACU_MEDIA_DESCRIPTION *media_description)
{
   static ACU_UINT local_port = 2000;
   ACU_PAYLOAD *acceptable_payload = 0;
   ACU_PAYLOAD *telephone_event = 0;
   ACU_INT found_valid_answer = 0;
   SRTP_ATTRIBUTE srtp_attributes;

   if(NULL == media_description)
   {
      return 0;
   }

   // Recurse through all offered media streams
   found_valid_answer = modify_media_descriptions(media_description->next);

   // Remove any connection information. In this example, all media streams
   // use the same IP address. This is set at the session level. There is no
   // need for a connection address at the media description level. This may
   // not be the case in general.
   if(media_description->connection_address.address)
   {
      free(media_description->connection_address.address);
      media_description->connection_address.address = 0;
   }
   media_description->connection_address.address_type = 0;
```

```
   // Set the media direction
   switch(media_description->media_direction)
   {
   case ACU_SEND_RECV:
      media_description->media_direction = ACU_SEND_RECV;
      break;
   case ACU_SEND_ONLY:
      media_description->media_direction = ACU_RECV_ONLY;
      break;
   case ACU_RECV_ONLY:
      media_description->media_direction = ACU_SEND_ONLY;
      break;
   case ACU_INACTIVE:
      media_description->media_direction = ACU_INACTIVE;
      break;
   default:
      break;
   }

   // Look for an 'a=crypto:...' line in the SDP. This attribute is used to defined
   // any secure RTP parameters.
   check_for_srtp(media_description->miscellaneous_attributes, &srtp_attributes);

   // These examples are not interested in any other miscellaneous attributes.
   free_misc_attributes(&media_description->miscellaneous_attributes);

   if(strcmp("", srtp_attributes.local_key_param))
   {
      // There was a valid 'a=crypto:...' line in the offer. Add to answer.
      media_description->miscellaneous_attributes =
(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE*)malloc(sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
      memset(media_description->miscellaneous_attributes, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
      media_description->miscellaneous_attributes->attribute = (char*)malloc(strlen("crypto:") +
         (srtp_attributes.tag / 10) +
         strlen(srtp_attributes.crypto_suite) +
         strlen("inline:") +
         strlen(srtp_attributes.local_key_param) +
         4);
      sprintf(media_description->miscellaneous_attributes->attribute, "crypto:%d %s inline:%s",
         srtp_attributes.tag,
         srtp_attributes.crypto_suite,
         srtp_attributes.local_key_param);

      // Transport type for SRTP is RTP/SAVP
      if(media_description->transport)
      {
         free(media_description->transport);
      }
      media_description->transport = (char*)malloc(strlen("RTP/SAVP") + 1);
      strcpy(media_description->transport, "RTP/SAVP");
   }
   else
   {
```

```
      // Either:
      // 1. no SRTP attribute has been found or, if one was,
      // 2. it was in some way mal-formed.
      // If 1 is the case, the application is free to do as it chooses - local
      // policy may require SRTP which would result in the call being rejected.
      // Otherwise it would continue as usual.
      // If 2 is the case the call should be rejected if SRTP is required.
      // Endpoints that are unaware of SRTP would ignore these attributes and
      // continue as normal. In this case, the UAC would disconnect the call
      // if SRTP is required.
      // For the purposes of these examples, the call will continue as normal.
   }

   // Attempt to find a compatible payload for this media type
   get_acceptable_payload(media_description, &acceptable_payload, &telephone_event);

   if(acceptable_payload)
   {
      // In these examples, an arbitrary port number is used. In general,
      // this would be provided by the media server being used.
      local_port -= 2;
      media_description->port = local_port;
      free_payloads(&media_description->payloads, media_description->media_type);
      media_description->payloads = acceptable_payload;
      if(telephone_event)
      {
         media_description->payloads->next = telephone_event;
      }
      found_valid_answer = 1;
   }
   else
   {
      // No matching payload found in this media stream.
      // Streams are rejected in SDP by setting the port to zero.
      media_description->port = 0;
   }

   return found_valid_answer;
}
```

B.4 get_acceptable_payload

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   get_acceptable_payload()                                              |
|                                                                         |
|   This function returns the first supported payload (as defined by an   |
|   arbitrary list and a telephone event if one is present. These examples|
|   illustrate the use of the media types, audio, video and image. Other types|
|   are not covered.                                                      |
|                                                                         |
\-------------------------------------------------------------------------*/
void get_acceptable_payload(ACU_MEDIA_DESCRIPTION *media_description,
                            ACU_PAYLOAD **acceptable_payload,
                            ACU_PAYLOAD **telephone_event)
{
   ACU_PAYLOAD *temp_payload;
   ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE* maxBitRate;
   ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE* maxBuffer;

   switch(media_description->media_type)
   {
   case ACU_IMAGE:
      *acceptable_payload = (ACU_PAYLOAD*)malloc(sizeof(ACU_PAYLOAD));
      memset(*acceptable_payload, 0, sizeof(ACU_PAYLOAD));
      (*acceptable_payload)->payload.image.image_payload_name = (char*)malloc(strlen("t38") + 1);
      strcpy((*acceptable_payload)->payload.image.image_payload_name, "t38");

      // There now follow an arbitrary set of attributes defining T.38 behaviour
      // This list is far from exhaustive.
      maxBitRate = (ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE*)malloc(sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
      maxBuffer = (ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE*)malloc(sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));

      memset(maxBitRate, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
      memset(maxBuffer, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));

      maxBitRate->attribute = (char*)malloc(strlen("T38MaxBitRate:9600") + 1);
      maxBuffer->attribute = (char*)malloc(strlen("T38FaxMaxBuffer:1024") + 1);

      strcpy(maxBitRate->attribute, "T38MaxBitRate:9600");
      strcpy(maxBuffer->attribute, "T38FaxMaxBuffer:1024");

      maxBitRate->next = maxBuffer;
      maxBuffer->next = 0;

      media_description->miscellaneous_attributes = maxBitRate;
      return;
   case ACU_AUDIO:
   case ACU_VIDEO:
      for(temp_payload = media_description->payloads; NULL != temp_payload; temp_payload = temp_payload->next)
      {
         if(audio_video_payload_is_supported(temp_payload))
         {
            if(!(*telephone_event) &&
                              !strcmp(temp_payload->payload.audio_video.rtp_payload_name, "telephone-event"))
```

```
                {
                   *telephone_event = (ACU_PAYLOAD*)malloc(sizeof(ACU_PAYLOAD));
                   memset(*telephone_event, 0, sizeof(ACU_PAYLOAD));

                   (*telephone_event)->payload.audio_video.rtp_payload_name =
                                                                  (char*)malloc(strlen("telephone-event") + 1);
                   strcpy((*telephone_event)->payload.audio_video.rtp_payload_name, "telephone-event");

                   // telephone-event uses a dynamic payload number. It must be the same in both directions.
                   (*telephone_event)->payload.audio_video.rtp_payload_number =
                                                          temp_payload->payload.audio_video.rtp_payload_number;
                   (*telephone_event)->payload.audio_video.packet_length =
                                                          temp_payload->payload.audio_video.packet_length;
                   (*telephone_event)->payload.audio_video.clock_rate = temp_payload->payload.audio_video.clock_rate;

                   // In this example, only DTMF digits (0-15) are shown as supported. This will often be the
                   // case but may vary. For example (amongst many others) the FAX tones CNG & CED may also be defined.
                   (*telephone_event)->payload.audio_video.payload_specific_options =
                                                                        (char*)malloc(strlen("0-15") + 1);
                   strcpy((*telephone_event)->payload.audio_video.payload_specific_options, "0-15");
                }
                else if(!(*acceptable_payload))
                {
                   // Found supported codec in this stream.
                   *acceptable_payload = (ACU_PAYLOAD*)malloc(sizeof(ACU_PAYLOAD));
                   memset(*acceptable_payload, 0, sizeof(ACU_PAYLOAD));

                   if(temp_payload->payload.audio_video.rtp_payload_name)
                   {
                      (*acceptable_payload)->payload.audio_video.rtp_payload_name =
                                     (char*)malloc(strlen(temp_payload->payload.audio_video.rtp_payload_name) + 1);
                      strcpy((*acceptable_payload)->payload.audio_video.rtp_payload_name,
                                                          temp_payload->payload.audio_video.rtp_payload_name);
                   }
                   (*acceptable_payload)->payload.audio_video.rtp_payload_number =
                                                          temp_payload->payload.audio_video.rtp_payload_number;
                   (*acceptable_payload)->payload.audio_video.packet_length =
                                                          temp_payload->payload.audio_video.packet_length;
                   (*acceptable_payload)->payload.audio_video.clock_rate =
                                                          temp_payload->payload.audio_video.clock_rate;
                   if(temp_payload->payload.audio_video.payload_specific_options)
                   {
                      (*acceptable_payload)->payload.audio_video.payload_specific_options =
                              (char*)malloc(strlen(temp_payload->payload.audio_video.payload_specific_options) + 1);
                      strcpy((*acceptable_payload)->payload.audio_video.payload_specific_options,
temp_payload->payload.audio_video.payload_specific_options);
                   }
                }
             }
             if(*acceptable_payload && *telephone_event)
             {
                break;
             }
          }
```

```
            break;
         default:
            break;
         }
      }
```

B.5 audio_video_payload_is_supported

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   audio_video_payload_is_supported()                                    |
|                                                                         |
|   Return TRUE if the supplied payload is supported by this application.  |
|   Return FALSE otherwise.                                               |
|   For the purposes of these examples, an arbitrary set of payloads are  |
|   defined as being supported. This will differ from application to      |
|   application according to the media server's capabilities and the      |
|   application's requirements.                                           |
|                                                                         |
\-------------------------------------------------------------------------*/
ACU_INT audio_video_payload_is_supported(ACU_PAYLOAD *payload)
{
   ACU_INT result = 0;
   // Arbitrary list of supported codecs with static payload numbers:
   // PCMU, G723, PCMA, G729, H263 (video)
   static const ACU_INT supported_static_payloads[5] = {0, 4, 8, 18, 34};
   // Arbitrary list of supported codecs with dynamic payload numbers:
   static const char *supported_dynamic_payloads[2] = {
      "telephone-event",
      "AMR"};
   ACU_UINT i;

   if(payload->payload.audio_video.rtp_payload_number < 96)
   {
      // Static payload number - just need to compare the numbers
      for(i = 0; i < 5; i++)
      {
         if(supported_static_payloads[i] == payload->payload.audio_video.rtp_payload_number)
         {
            return 1;
         }
      }
   }
   else
   {
      // Dynamic payload number - need to compare payload names
      for(i = 0; i < 2; i++)
      {
         if(!strcmp(supported_dynamic_payloads[i], payload->payload.audio_video.rtp_payload_name))
         {
            return 1;
         }
      }
   }

   return 0;
}
```

B.6 check_for_srtp

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   check_for_srtp()                                                      |
|   This function stores the details of the first 'a=crypto:...' attribute (if  |
|   present). In general, there may be an arbitrary amount of 'a=crypto:...'    |
|   lines contained in an SDP body and an application is free to choose         |
|   whichever one it likes although, according to RFC 4568, it should accept    |
|   the most preferred (topmost in the list) that it is capable of accepting.   |
|   The 3 crypto suites defined in RFC 4568 (also supported by Prosody X and    |
|   Prosody S) are:                                                       |
|      AES_CM_128_HMAC_SHA1_80                                            |
|      AES_CM_128_HMAC_SHA1_32                                            |
|      F8_128_HMAC_SHA1_80                                                |
|   It is assumed that there is only one SRTP key for each 'a=crypto:...' line  |
|   in the SDP. This may not be the case although often there will only be one  |
|   such 'a=' line.                                                       |
|                                                                         |
|   These examples are intended primarily to demonstrate the use of the         |
|   ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE structure. The error checking is not      |
|   exhaustive and the reader is encouraged to refer to RFC 4568 for more       |
|   detail on SRTP with respect to SDP.                                   |
|                                                                         |
\-------------------------------------------------------------------------*/
void check_for_srtp(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE *miscellaneous_attributes, SRTP_ATTRIBUTE *srtp_attributes)
{
   memset(srtp_attributes, 0, sizeof(SRTP_ATTRIBUTE));

   if(miscellaneous_attributes)
    {
      ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE *ma = miscellaneous_attributes;
      while(ma)
      {
         if(strstr(ma->attribute, "crypto"))
         {
            // Found an 'a=crypto:...' line. Check that it contains a valid crypto suite
            // and create an answer.
            if(srtp_extract_crypto_suite(ma->attribute,
                                         srtp_attributes->crypto_suite,
                                        &srtp_attributes->tag_length))
            {
               // These examples will extract the mandatory parts of an 'a=crypto:...' line
               // There are other optional parameters. For more information see RFC 4568. The
               // mandatory parameters are: tag, crypto suite and key params. It is the key
               // params that contain the optional information.
               if(!srtp_extract_tag(ma->attribute, &srtp_attributes->tag))
               {
                  // The tag parameter in the crypto attribute is mal-formed. For the
                  // purposes of these examples, the call will go ahead as normal. In general,
                  // an SRTP aware endpoint should reject the call here as secure RTP has been
                  // requested and cannot be supplied. The UAC should disconnect the call
                  // anyway as the response will not contain an 'a=crypto:...' line.
                  return;
               }
```

```
                    if(!srtp_extract_key_param(ma->attribute, srtp_attributes->remote_key_param))
                    {
                       // The key parameter is mal-formed. See comment for srtp_extract_tag.
                       return;
                    }
                    // All mandatory parameters have been found and are valid.
                    // For the purposes of these examples, a hard coded, base 64 encoded key is used.
                    // A realistic application would need to use a suitable algorithm to generate
                    // the combined key and salt and subsequently encode into base 64 notation.
                    strcpy(srtp_attributes->local_key_param, "mdws+SMU52xuhMEb14yOOHFIS5XZg/X9RxdyO/J+");
                    break;
                 }
                 else
                 {
                    // A valid crypto suite has not been found. See comment for srtp_extract_tag.
                 }
              }
              ma = ma->next;
           }
        }
     }
```

B.7 srtp_extract_tag

```
/*-------------------------------------------------------------------------------\
|                                                                               |
|   srtp_extract_tag()                                                          |
|   This function finds the tag which identifies one offered crypto suite from  |
|   another within a single media description. This tag will be used by the     |
|   answerer to identify which offered crypto suite is to be used. It is        |
|   assumed that the string 'crypto_attribute' starts 'crypto:..." and that     |
|   the rest is in a valid form - there is no error checking.                   |
|                                                                               |
\-------------------------------------------------------------------------------*/
ACU_INT srtp_extract_tag(const char *crypto_attribute, ACU_UINT *tag)
{
   ACU_INT rc = 1;
   char *p;
   char *temp = (char*)malloc(strlen(crypto_attribute) + 1);
   strcpy(temp, crypto_attribute);

   // The tag parameter lies between the first ':' and the next whitespace.
   // Set p to point to the first character after the first ':'.
   p = strstr(temp, ":") + 1;
   // NULL terminate at the first white space. p will now point to a string
   // containing the tag parameter.
   *strstr(p, " ") = '\0';

   // Convert to int
   *tag = atoi(p);
   // Check validity
   if(!tag && ('0' != *p))
   {
      // Invalid tag parameter. Must be an integer.
      rc = 0;
   }

   free(temp);

   return rc;
}
```

B.8 srtp_extract_crypto_suite

```
/*-------------------------------------------------------------------------\
|                                                                         |
|   srtp_extract_crypto_suite()                                           |
|   This function finds the crypto suite which is to be used in the SRTP  |
|   session. It returns 1 if a valid crypto suite (as defined in RFC 4568) has  |
|   been found and 0 otherwise. It also populates the SRTP authentication tag   |
|   length.                                                               |
|                                                                         |
\-------------------------------------------------------------------------*/
ACU_INT srtp_extract_crypto_suite(const char *crypto_attribute, char *crypto_suite, ACU_UINT *tag_length)
{
   ACU_INT rc = 0;
   char *p;
   char *temp = (char*)malloc(strlen(crypto_attribute) + 1);
   strcpy(temp, crypto_attribute);

   // The crypto suite parameter is located between the first two white spaces.
   // Set p to point to the first character after the first white space.
   p = strstr(temp, " ") + 1;
   // NULL terminate at the next white space. p will now point to a string
   // containing the crypto suite parameter.
   *strstr(p, " ") = '\0';

   // Check that the crypto suite is valid. These 3 crypto suites are defined
   // in RFC 4568. Others may be defined in additional documents.
   if((!strcmp("AES_CM_128_HMAC_SHA1_80", p)) ||
      (!strcmp("AES_CM_128_HMAC_SHA1_32", p)) ||
      (!strcmp("F8_128_HMAC_SHA1_80", p)))
   {
      // Found valid crypto suite
      strcpy(crypto_suite, p);
      *tag_length = atoi(p + strlen(p) - 2);
      rc =  1;
   }

   free(temp);

   return rc;
}
```

B.9 srtp_extract_key_param

```
/*-------------------------------------------------------------------------------\
|                                                                               |
|   srtp_extract_key_param()                                                    |
|   This function will extract the 40 byte, base 64 encoded key from a char*    |
|   obtained from an ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE structure. It will       |
|   ignore any optional parameters that may be present and return 0 if the key  |
|   is not 40 bytes in length.                                                  |
|                                                                               |
\-------------------------------------------------------------------------------*/
ACU_INT srtp_extract_key_param(const char *crypto_attribute, char *key_param)
{
   ACU_INT rc = 1;
   char *p, *q;
   char *temp = (char*)malloc(strlen(crypto_attribute) + 1);
   strcpy(temp, crypto_attribute);

   // The key parameter is located after 'inline:' and continues either
   // to the end of the string or to the first instance of a '|'.
   // Set p to point to the first character after 'inline:'
   p = strstr(temp, "inline:") + 7;
   q = strstr(p, "|");
   if(q)
   {
      // For the puposes of these examples, optional paramters are being
      // ignored. These are delimeted by a '|'. In general, an application
      // would need to check these parameters for the purposes of
      // configuring the media. See RFC 4568 for more details.
      *q = '\0';
   }
   if(40 != strlen(p))
   {
      // All 3 crypto suites use a 128 bit key combined with a 112 bit salt.
      // In base 64 notation, this will always be represented by a 40 byte string.
      rc = 0;
   }
   else
   {
      strcpy(key_param, p);
   }

   free(temp);

   return rc;
}
```

Contact us

Phone

+44 (0)1908 273800 (UK)
+1(781) 352 3550 (USA)

Email

Info@aculab.com

Sales@aculab.com

Support@aculab.com

Socials

Certificate number IS 722024
ISO 27001:2013

Certificate number FS722030
ISO 9001:2015