VOIP IN DEPTH

An Introduction to the SIP protocol, Part 2

In Part 1 of our SIP primer, I covered the SIP foundation layers starting from the message structure and ending with the SIP transactions. We saw how phone registrations and proxies could work using these layers. This second part completes the discussion by covering the way SIP defines calls, and in general, any type of communication. Naturally, this installment is built on the previous part, and therefore you should read Part 1, or at least have some prior knowledge, before proceeding with Part 2. Similar to the previous installment, I will also refer here to the latest specs that influenced the basic SIP scenarios.

SIP Dialogs

The following INVITE message does not just start a new transaction, it is also a request to start a new dialog.

```
INVITE sip:hannibal@arstechnica.com SIP/2.0
Via: SIP/2.0/UDP home.mynetwork.org;branch=z9hG4bK8uf35f
To: Jon Stokes <sip:hannibal@arstechnica.com>
From: Gilad <sip:gilad@voxisoft.com>;tag=n23ycs
Call-ID: nbo34tsggvsqap@home.mynetwork.org
CSeq: 59164 INVITE
Contact: sip:gilad@voxisoft.com
Max-Forwards: 70
```

A 2xx response opens the dialog. The client that originally sent the INVITE would send an ACK request to confirm that it received the 2xx response. However, we'll see that this is specific for INVITE and not for dialogs in general. SIP dialogs are not limited for the INVITE method only. Extensions may also use define methods that initiate a dialog.

Clients are expected to maintain the state of the dialogs. (As we saw in the first part, proxies along the signaling path do not maintain dialog state). Each dialog holds the following information:

- Call-Id
- Local tag
- Remote tag
- Local URI
- Remote URI
- Remote target
- Route-set
- Local CSeq
- Remote CSeq
- A boolean flag called "secure"

The first three values identify the dialog. The dialog initiator chooses a Call-Id and places the value in its header. The initiator also chooses a random local tag and places it as a parameter of the "From" header (the "To" header remains without a tag). The device that accepts this request refers to the tag in the "From" header of the request as the dialog's remote tag. The receiver then creates an additional tag and places it as a parameter in the "To" header of the response. The initiator sees the tag value in the "To" header and refers to it as the dialog's remote tag.

When one party sends a dialog request, several different 2xx responses may arrive. This multiple-response situation occurs when a proxy forks the request and several devices answer. Proxies cannot interrupt with 2xx responses, as they are not aware of dialogs. Hence, all these responses propagate back to the one who sent the request. When you receive several of these responses, effectively, several dialogs were created based on that single request. These dialogs each have a different identifier, even at the source, as the remote tag is unique for each of these dialogs. Any subsequent requests on a specific dialog contain the same identifiers as the ones established in the handshake process.

The contact of the request becomes the other end's "remote target." However, the initial request URI is not necessarily the initiator's remote target. When it receives the 2xx response, it also receives the actual remote target via the response's contact header. Thus, if one sent a request to an AOR, the response may come back with a contact address that is specific for the device, at least for the lifetime of the dialog. ACK, and all subsequent requests, would place in its request URI the dialog's remote target. Therefore, if a proxy previously forked a request to an AOR, it would not do that for subsequent requests, as this time the request URI is different.

Dialogs also hold a route-set. This route-set is a list of SIP URIs and its goal is to contain all the proxies that route all requests on the dialog. The proxies themselves build the route-set, but do not store it internally. Each proxy that routes the first request sends not only an additional "Via" header, but also a "Record-Route" header. When the request has reached its destination, it has a list of URIs within the route-set. Before sending a positive response to the request, the device stores the list in its internal dialog state and sends the same headers, in the same order on the response.

Responses are routed based on their "Via" headers (which are also copied as-is from the request to the response), and thus proxies do not add or remove any response "Record-Route" header. The initiator that also maintains its own dialog internal state also stores this list of URIs, but this time in reverse order (since the first proxy that added this header actually has the last of the headers in the response). Subsequent requests have this route-set copied to a "Route" header. "Route" is different than "Record-Route"; it tells proxies to route the request to a specific destination and not base it upon any other internal routing rules (as they did the first time).

A proxy that has its own address in the top route header would remove itself from the request it sends out, and would resolve the IP address of the outgoing request from the next route header. If it doesn't have an additional route header, it would send it based on the request URI.

f you look at a route-set, you would notice all routes have an "lr" parameter. This parameter states that this is a loose route, effectively meaning that the proxy is RFC 3261 compliant. Proxies that comply with previous RFC have strict route rules and must have their own address in the request URI. Thus, for backwards compatibility, one must change the request URI if the proxy does not specify it supports loose route.

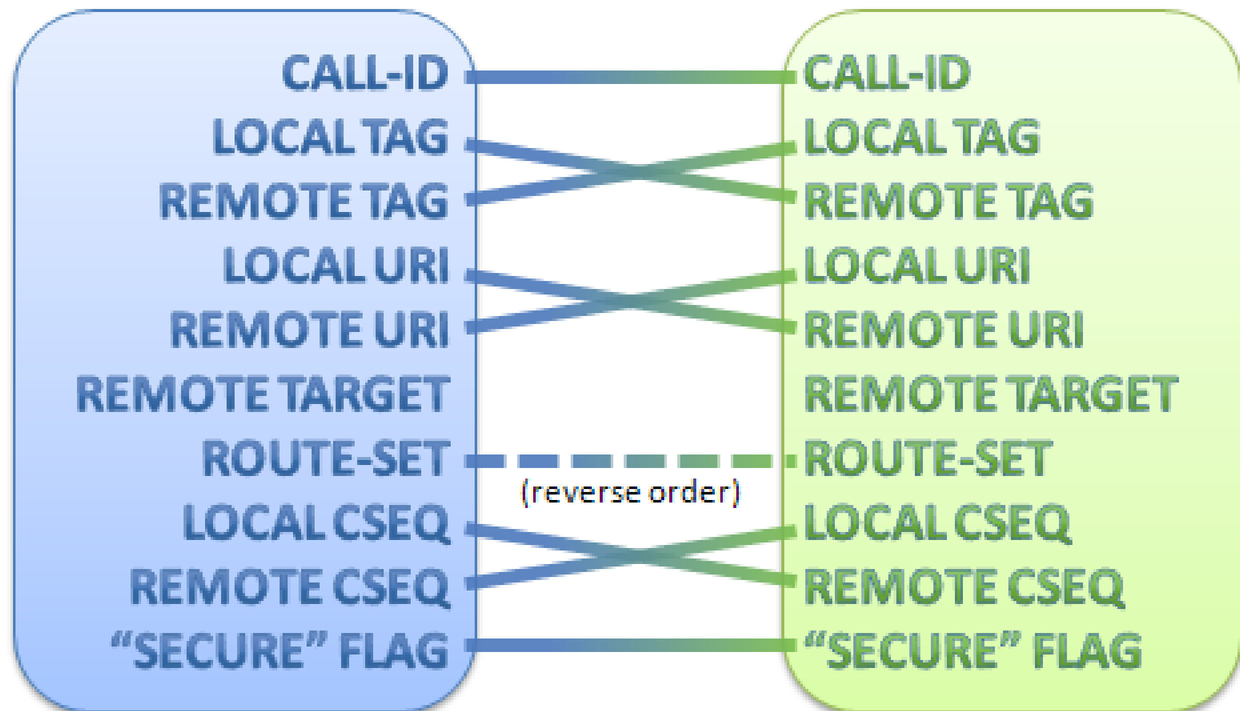
Whenever one of the two dialog participants sends a request, it places the local tag in the "From" header of the request and the remote tag in the "To" header. When a response is sent, this is reversed, the local tag is placed in the "To" header and the remote tag goes to the "From" header. Because one endpoint's local tag is the other's remote tag, the "From" and "To" tag parameters look the same. The same idea goes for the URIs in the "From" and "To" headers. These are mapped to "local URI" and "remote URI" in a similar way.

The party creating a dialog chooses its first CSeq value, which becomes its local CSeq value, and the other's remote CSeq value. As previously discussed, the response includes the same CSeq value and therefore the other participant's CSeq value only becomes known when it sends the first request. Thus, when a dialog has been established, only one of the CSeq fields has value. Someone sending a request on a dialog should first increment its local CSeq value by one and then send the request using this local CSeq value. This helps to know the order of the requests on a given dialog.

The last item in the dialog internal state is the secure flag. This flag simply indicates whether a device should generate requests with encryption (for SIP that means TLS). The addresses of the remote target and the contact start with "sips:" and not "sip:".

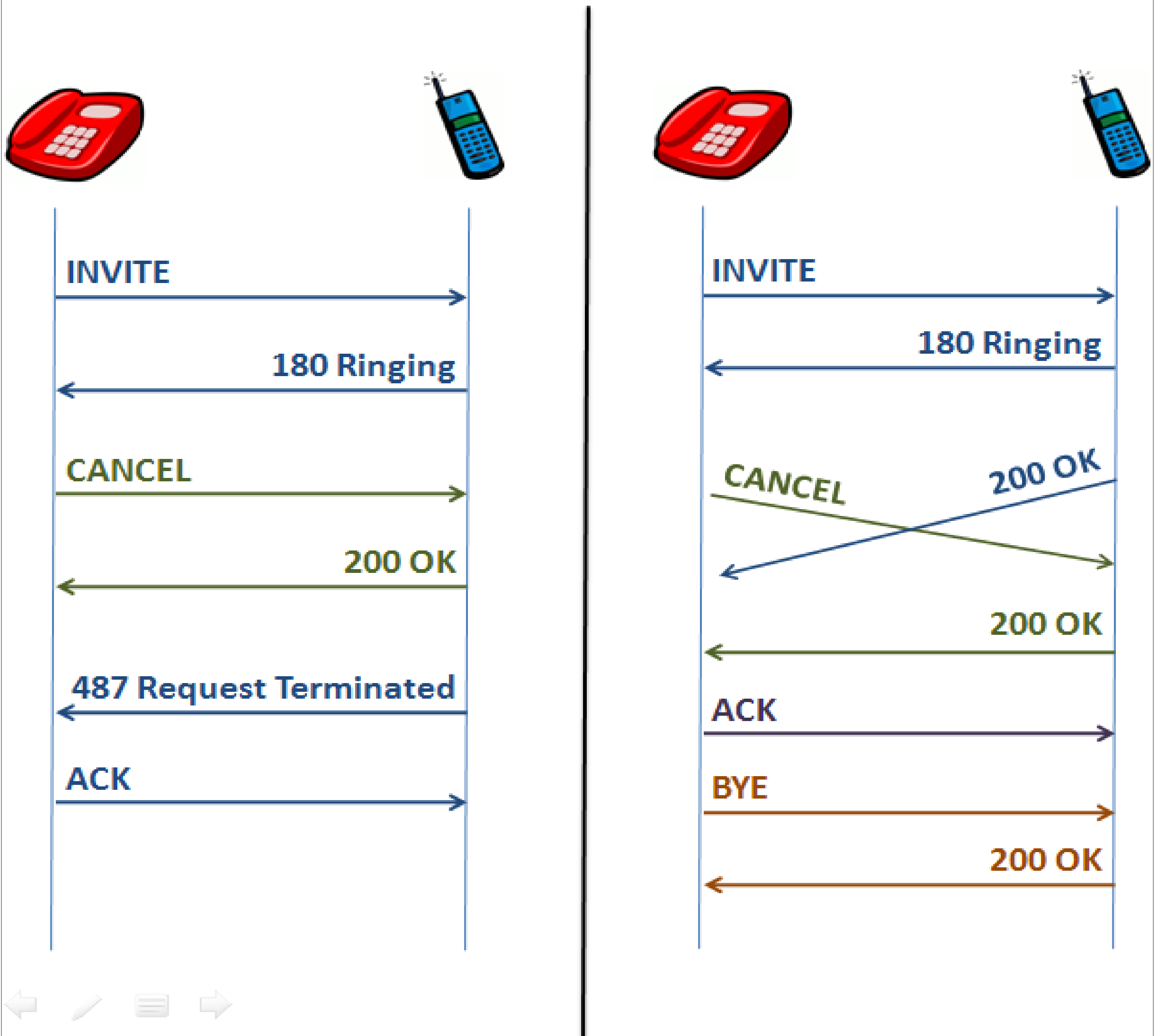
I

As you can see, both endpoints hold the same information in different fields. The following illustration shows the relationship between the fields on both ends:



There must be a way to close the dialog, not just open it. For INVITE dialogs, the way you close a dialog is to send a BYE request. Obviously, any of the two dialog participants may send BYE, which correspond to one hanging up its phone. As with any request, you must respond to the BYE, but in this case, even an error response (or timeout of the transaction) would prompt the sender of the BYE to close the dialog. This is to prevent from the remote end from forcing a device to keep a dialog open with its state.

Another scenario that SIP must support is to be able to cancel a call attempt. This happens when one party does not pick up the phone. The CANCEL method is used in this situation, but this method has many quirks. CANCEL is unique due to the fact that it has the same branch value in the "Via" header as the INVITE transaction, but in the CSeq part its method is CANCEL and not INVITE. This differentiates the two transactions, as both values identify a transaction. CANCEL also oddly does not have multiple "Via" headers. Each proxy receiving a CANCEL request would issue its own CANCEL request without the previous "Via" headers. Most importantly, CANCEL does not guarantee that the request has been cancelled, even though most of the time you would get a 200 response for it. A successful response on CANCEL means that it reached its destination. Only when you receive a 487 (Request Terminated) response does it mean the CANCEL request was honored. Particularly, it's quite possible that a 2xx response for the INVITE request was sent prior to receiving the CANCEL request, and thus one would receive this response to the INVITE even though the CANCEL was sent. If this is the case, the only way to close the dialog at that point is to ACK the 2xx response and send a BYE request.



Cancel Scenarios

Please note that not all requests are part of a dialog. A common misconception is that REGISTER creates a new dialog. REGISTER is just a transaction that does not initiate a dialog or take place within any dialog. This is despite the fact that you would usually find subsequent registrations with the same Call-id and CSeq value incremented by one. An outcome of this is that a REGISTER request does not create a bidirectional communication between the registering device and the registrar, and therefore, based on the REGISTER message alone, the registrar cannot notify the registering device that a registration has expired prior to the original expiration time.

Signaling and media

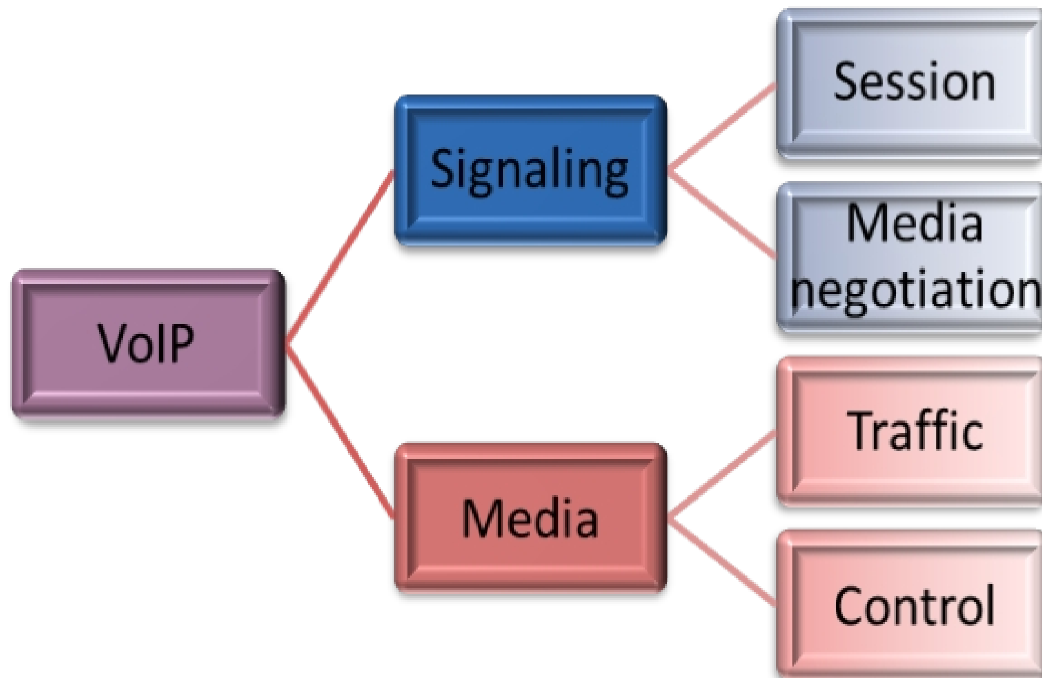
So far, we've focused most of our attention on the signaling part, but having a signaling protocol just to control the session is rather useless without the means to send the contents of the call. In a voice call, this is known as speech, and that’s the media part. Before being able to send and receive media, the parties must negotiate the media properties. Why do you need media negotiation? For voice, the reason is that there are many different ways to represent the contents and compress it. This is similar to having several formats to play sound on your desktop (WAV, MP3 and Ogg), but in this case, the devices choose the format for the conversation. Furthermore, VoIP does not necessarily mean it has actual voice.

From day one, VoIP has been aiming to not just replace the standard telephone system, but also enhance it, so it's possible to negotiate a media type that has nothing to do with sound. Naturally, there is a protocol to negotiate the media type and representation, but you would usually find this negotiation encapsulated within the signaling messages and not sent separately. The reason for this is simple—as you create a session or modify a session, the media properties are also negotiated. Thus we have the signaling divided to two parts: session and media negotiation.

The media is also separated into two parts: contents and control. Once the session participants agree on the media attributes, they can start sending the contents. Because VoIP operates on an IP network, the media is divided into packets. For voice, it's common that each packet represents 20ms of sound. This means that in a conversation with two participants, you would send a packet every 20ms, or 50 packets per second. There are ways to improve that rate by actually detecting voice activity and refraining from sending packets when the conversation contains only silence. This usually reduces the traffic by 50 percent. 20ms is not a necessity, and some choose to use 30ms or even 10ms, but 20ms is by far most common.

As media packets are sent and received, the parties would like to get some feedback. Since media contents are sent at a high rate, it makes no sense to acknowledge each packet that was received, so instead, each end party sends a report detailing some key statistics such as how many media packets it has sent and how many it has received. This helps the devices verify that the network is actually transferring all the packets and that the quality is acceptable.

There are several ways to calculate quality; most of these methods compare the time packets arrived to the time they should have arrived, and they also detect and count the number of packets that were lost.



VOIP Elements

Offer-answer

The default media negotiation protocol is the Session Description Protocol or SDP. This is defined by RFC 4566. SDP is not used solely by SIP and thus has some fields that are irrelevant for SIP's case.

```
v=0
o=me 634962690 634962690 IN IP4 home.mynetwork.org
s=-
c=IN IP4 home.mynetwork.org
t=0 0
m=audio 28534 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

We'll cover only those headers important to this discussion:

- **'m'-line**: This defines a media stream, and each media stream has a type. In this case, it's an audio stream. The number that follows the type indicates the listening port. The port number is followed by the protocol; in this case, one uses RTP, probably the most common protocol to use for audio/video. A list of possible codecs follows the "RTP/AVP" text to signify which codecs are supported. We mentioned that the media has the content, sent over RTP in this case, as well as the control that provides information, such as statistics. RTCP provides the media control and, by default, you receive the RTCP on the RTP port + 1. For that reason RTP usually uses even port numbers and RTCP uses odd port numbers.
- **'a'-line**: This is an attribute. Attributes can be anything that describe either the whole SDP (appearing before the 'm'-line) or the 'm'-line itself. The 'a'-line in this example provides an attribute called rtpmap that matches a numeric value of the codec in the 'm'-line to the actual registered codec value. Another attribute, for example, may set the RTCP port value explicitly.
- **'c'-line**: This is the IP address where one wishes to receive media. This does not have to be equal to the IP address receiving the signaling. It is also possible to have a specific IP address for a media stream by adding a 'c'-line after the corresponding 'm'-line.

RFC 3264 defines the SDP offer-answer model for SIP. The media properties are negotiated as the call is set up, during the call three-way handshake. The simplest scenario is to have the offer in the INVITE request and the answer in the 200 OK of the INVITE. The offer includes all the media streams to set up, and each stream has the offered codecs and other attributes. The answer must have the same number of 'm' lines that were in the offer. If one of the media streams is not accepted, the answer will have '0' in the port number, thus disabling it. The answer also chooses only the codecs it supports out of the proposed codecs. Naturally, if the device receiving the offer cannot establish a communication, it will reject the INVITE request.

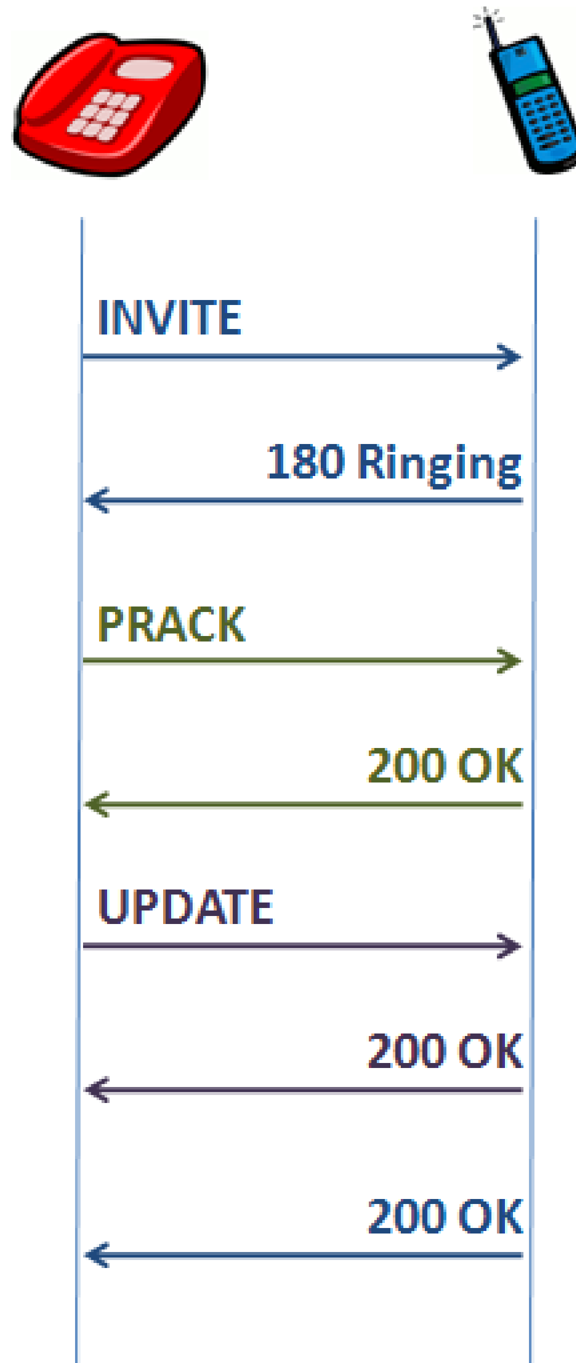
A different scenario is to have the INVITE request without any SDP. In this case, the offer is expected to be in the 2xx response of the INVITE, and the answer to this offer would be in the ACK. This time, if media communication cannot be established, you cannot just ignore the 2xx response and cannot respond to it negatively. It should ACK the 2xx response with all media lines disabled, and then immediately send a BYE.

Things can become more complex in scenarios other than those outlined above. For example, you might use PRACK (defined in RFC 3262), which is a provisional response acknowledgement. There are some cases in which we would like to guarantee the delivery of a provisional response different than 100, such as 180 ringing (you may recall that TCP isn't enough to know the message reached the other end, as a proxy along the path may send the response over UDP). For that reason, a device sending a reliable provisional response would receive a PRACK, acknowledging the response, and because PRACK is a request, it would respond to that request with 200 OK. This does not mean a call was established--just that one party has received the PRACK; the INVITE transaction is still pending a final response. However, the provisional response may include an SDP answer to the original offer. Furthermore, a PRACK may have a new offer. To make things more complex, even after the PRACK offer-answer, you might send an UPDATE that includes, once again, a new offer.

Why make it so complex? The former scenario usually involves early media, such as ring-back tone (the music you hear when calling a person subscribed to this service) or

interworking with PSTN. The later scenario may involve resource reservation. These scenarios, by definition, require setting and changing the media properties as the call begins, and this forces SIP to take a complex path to support them.

The following flow demonstrates a complex INVITE scenario. For clarity purposes, each transaction has a different color.



After the initial negotiation, you might also change the media attributes. For example, during a voice call, one party may ask to receive a fax, or to add or remove video. To do that, the

party sends an INVITE request on the dialog (contrary to a new INVITE, this INVITE already has a tag parameter in the "To" header). Thus, the INVITE looks like any request that you send within the existing dialog, but it can contain SDP that changes the media behavior. This request type is often referred as re-INVITE.

Sometimes you might use UPDATE rather than re-INVITE. The reason is that re-INVITE is also a target refresh request. We have seen that the device stores the dialog remote target; because the remote target is set at the initial negotiation, a device would ignore the “Contact” header value of a request on the existing dialog. This is not true for target refresh requests that may prompt a change in the remote target. UPDATE may contain SDP with a new offer, similar to re-INVITE, but it is not a target refresh and thus affects only the media negotiation.

A very common scenario for changing media in an active call is placing the call on hold. Hold, in SIP/SDP terms, means that the one placing the call on hold no longer receives media, but it may decide to send media. For that reason, each media stream has an attribute that can be send-receive, send-only, receive-only or inactive. By default, a stream is send-receive. When you place a stream on hold, you change the stream to send-only and its peer changes it to receive-only. If both participants place the stream on hold, it becomes inactive.

Older hold scenario examples did not use these attributes, but rather changed the IP address to 0.0.0.0. This may seem equivalent to "inactive," but it also means that the RTCP cannot be used either. That means that devices that use the RTP/RTCP to know the call is still active would not receive any packets and thus would have no indication of the status of the call.

The variety of different cases and quirks, some of which are described above, have created many interoperability questions. In fact, some questions remain unanswered even today. Nevertheless, to cope with this, the IETF provided SDP offer/answer examples in RFC 4317 and, perhaps more importantly, there is a draft that attempts to provide guidance on many of these questions. So, if you're looking for cases in which SIP did not manage to keep things simple, offer-answer is definitely ranked high.

Extensions

We mentioned numerous times that SIP supports extensions. Obviously, different implementations will support different extensions. It's a question of network policy as to whether specific extensions are mandatory, allowed, or forbidden. For that reason, SIP uses option-tags. An option-tag is a short string that indicates support for a specific extension. When an RFC defines a new extension it may define a new option-tag and register its name in IANA's registry. For example, PRACK has an option-tag named "100rel." If a client sending an INVITE supports PRACK, it will include the following header:

Supported: 100rel

A device that receives this INVITE request and wants to send a provisional response reliably using this extension can now send it with the following header:

Require: 100rel

Since the original UAC indicated it supports "100rel," it is expected to send PRACK for that provisional response. The INVITE could have also included the 100rel in the "Require" header. If the recipient does not support this extension, it should send back a 420 (Bad Extension) response with the names of the extensions it does not support in the "Unsupported" header. Headers that include these option-tags may appear multiple times, or include several comma-separated values and therefore one may indicate several extensions in a single message.

Extensions may also define new methods. To indicate the supported methods, you place this in the "Allow" header. Furthermore, to indicate which message bodies are supported, their names are included in the "Accept" header. To figure out, prior to a session, the capabilities of a device, it’s possible to send an OPTIONS request. The request includes in the "Accept" header the message body it wishes to query (obviously most commonly this would be "application/sdp"). A successful response would include the following:

- The option-tags in the "Supported" header.
- Methods supported in the "Allow" header (except a proxy generates the response in which case proxies should not limit the method type).
- The body type in the "Accept" header.
- Languages supported (for text such as response reason phrase) in the "Accept-Language" header.
- Body encoding (such as compression via gzip) in the "Accept-Encoding" header. It is not common practice to compress the body.
- The body itself indicating its capabilities. For SDP, this would include the different media types in each "m"-line and the codecs with their capabilities in corresponding attributes.

Extending SIP

SIP does not stop with the INVITE request. RFC 3265 defines a SIP-based infrastructure for

event notification, and one of the most common event notifications is presence (i.e., you have a list of contacts in a presence client, and you get notifications when a contact goes online, offline, leaves to lunch, etc.). Therefore, the presence client would send a SUBSCRIBE request to a presence server. Once it receives a 2xx response to the SUBSCRIBE request, a new dialog is formed. The server can now generate a NOTIFY request on the dialog that indicates the user status has changed.

The IETF knew that there might be many scenarios aside from presence that include this mechanism, so they defined this infrastructure on top of SIP. For example, when a phone is busy, you can use a call completion feature to be notified when the person is available for the call. This, once again, involves subscribing to an event and receiving status notifications.

To support a specific event, you should implement an event package that defines the event name, parameters, status types, and bodies a message may include. A SUBSCRIBE request would include the event name in the "Event" header, so that a single user can send out multiple SUBSCRIBE requests, even to the same target. To check which event packages are supported, a new header called "Allow-Events" was defined. Subscriptions have an expiration time, and a subscriber that wishes to keep the subscription open must send another SUBSCRIBE request within the dialog before it expires. This dialog closes when the notifier sends a NOTIFY request with a terminated status.

There are numerous examples of event-notification extensions, such as RFC 3856 defining presence, RFC 3680 that sends a notification for user registrations and RFC 4730 defining signaling DTMF events. One event-notification extension, which is more related to calls, is call transfer. To this end, RFC 3515 defines the REFER method.

Suppose participants A and B have a call, and participant A wants to transfer the call to participant C. A would send out a REFER request to B with C's address. B sends out an INVITE request to C, specifying it was referred by A. REFER uses implicit subscription, which helps A know whether the transfer has completed successfully. B then sends NOTIFY

requests to A, indicating the status of the transfer. This is illustrated in the following flow:

A different possible transfer scenario is to send a REFER request from A to C. C would then send an INVITE to B. The INVITE would include the dialog details of A and B to specify which dialog is being replaced. It is also possible for A to have two concurrent INVITE dialogs, one

with B and one with C prior to the REFER; this is also known as attended transfer.

Let's assume one sends a REFER within an existing dialog. Often, that makes sense because A probably has an INVITE dialog with B and thus, it may send a REFER on the existing dialog instead of creating a new one. In the REFER diagram above, we have colored each dialog usage with a different color. You can see that despite the fact that BYE was sent, we still have communication between the two parties. If the REFER was sent on the same dialog as the INVITE, the NOTIFY and 200 OK are also sent on this dialog. This is because the subscription still exists.

Having call and subscription on the same dialog may seem harmless at first, but sometimes it's not trivial. For example, what should you do when an error response arrives? Some responses affect just the transaction, some affect just the specific usage (INVITE or REFER), and some, such as 404 not found, may affect both INVITE and REFER (suddenly the user does not exist, although the dialog was established). In addition, the dialog remains open even after a BYE request (if the REFER notifications are still active), but many implementations tend to close the dialog when seeing a BYE simply because this multiple usage on a single dialog was not clearly defined. RFC 5057 attempts to address that by defining the desirable behavior of multiple usages on a single dialog. It also discourages using this ability since it recognizes many of the complications. In fact, RFCs published after the REFER RFC do not define multiple usages or implicit subscriptions.

Readers who want to learn more in-depth details about the event notifications are encouraged to open the RFC, as its language is rather clear. I should mention that there is a new draft updating RFC 3265, which addresses many issues that have come up in recent years. Some changes clarify the text, others alter some definitions (e.g., the dialog is now created only when the NOTIFY transaction completes). Other changes also discourage multiple usages on a single dialog.

Finale

Part 1 focused on the SIP foundations and showed the protocol's simplicity. Part 2 has described the more complex areas of SIP that require more attention to detail. If you've survived this primer then you should have a good baseline for understanding this protocol. It might be a good idea to look at RFC 5359, which provides excellent examples to many SIP services scenarios. If you want to extend your knowledge of SIP even further, I would recommend reading the base SIP RFC, as well as following the many extensions, some of which are mentioned in this primer. Finally, the IETF forums are very friendly to any person who has a question, including newcomers. You can find many answers in the forum's archive, and you can post any question on any related matter.