VOIP IN DEPTH

An Introduction to the SIP protocol, Part 1

In our last VoIP installment, we looked at the main reasons why SIP has become a widely adopted protocol, but we left details of the protocol's inner workings fairly vague. This article will drill down into the way the Session Initiation Protocol (SIP) works, and it should serve as a good starting point for really learning SIP. If you haven't already done so, you are encouraged to read the previous article, although it's not a prerequisite. This introduction also covers the latest SIP extensions and changes, so it gives a complete view of the protocol's current state, rather than just the basic, underlying RFC.

Session Initiation Protocol (SIP) is a VoIP signaling protocol. As its name suggests, it has everything to do with setting up sessions, which means it has the responsibility for starting a session after you dial a number (or double-click, in some cases). As such, SIP's role also includes maintaining user registrations with a server, defining session routing, handling various error scenarios, and, of course, modifying and tearing down sessions.

We'll present this introduction in two parts. In the first part, we'll focus on the SIP foundation layers. These layers allow creating a network of SIP servers. In the next article, we will go through the way a phone communicates with the rest of the world using this server network, based on the same foundation layers.

SIP Foundations

Mesage Structures

SIP shares the same message structure as HTTP and RTSP. As a result, most of the text description here would fit these protocols as well. Each message is either a request or a response. All messages are text-based and have the following form:

- First line
- Headers
- Empty line
- Optional body

The first line is different for a request and a response. A request's first line uses the following format:

<*method* (request type)> <*URI* (address/resource)> <*version*>

For example, a SIP request's first line can be:

REGISTER sip:arstechnica.com SIP/2.0

A response comes in the form of:

<version> <code> <reason phrase>

The *version* is the same in all messages. The *code* is a 3-digit value, and the reason phrase could be any text describing the nature of the response. A proper first line in the response could be:

SIP/2.0 200 OK

It's easy to scan the first characters of a message to detect whether it's a request or a response ("/" is not a valid character for the method field, therefore even a generic parser could differentiate a request from a response).

Next come the headers of the message. Some headers contain vital information and thus are mandatory, but many headers are optional. Each header has a name and a value, and some have parameters. For example:

Contact: sip:gilad@voxisoft.com;Expires=2000

Contact is the header name, *sip:gilad@voxisoft.com* is the value, and expires=*2000* is a parameter. Other parameters may appear separated by semicolons.

Some headers may appear just once in a message, and some may appear multiple times. An equivalent approach to using multiple headers is to separate each value-parameter set with a comma. Some headers have a compact form for which the header name is shorter. For example, you can use "m" instead of "Contact".

After the last header, there's an empty line followed by the body of the message. The body could be anything, even binary information. The header *Content-Type* specifies the type of the message body. In order to know the length of the body, you have to look at the *Content-Length* header. (If the transport type is UDP, then the *Content-Length* header is not mandatory. It is, however, mandatory for TCP). One common use of the body is to encapsulate the media negotiation protocol within the SIP message.

That's pretty much what you need to know in order to understand the basic structure of SIP. It is rather straightforward, and anyone who is already familiar with protocols that have similar structure will feel right at home. Of course, we still have to understand what this text means and what a SIP device should do with the messages it sends and receives.

Transport Types

By default, SIP messages are sent on port 5060 if they're unencrypted. Encrypted messages are sent over port 5061. One can specify a port other than the default within the SIP address and override this default value.

SIP mandates support for both UDP and TCP, but it can successfully operate on practically any transport type. It defines different behavior per transport only when the characteristics of the specific transport require it to do so. For example, UDP does not guarantee delivery, so SIP retransmits the messages. In TCP, such retransmission is really unnecessary (and confusing), so no one should retransmit a message. For the most part, other than the transaction layer (detailed in the next sub-section), almost no other component changes its behavior due to the transport.

In fact, because SIP operates hop-by-hop (clients do not usually communicate directly, but rather use proxies along the signaling path to send and receive messages), each hop could change the transport type. So a client may receive a message over TCP even if the original message was sent over UDP. SIP's transport type independence enables the possibility of defining new transport types that were not originally included when SIP itself was first defined. For example, RFC 4158 is a very short RFC that defines SIP over SCTP.

For connection-based transports (e.g., TCP and SCTP), the state of the connections is maintained. Connections are kept open and reused to save time and resources. The recommendation is to keep a connection open for at least 32 seconds after the last message, but in practice it's application-defined. Because there is no defined limit for the number of different SIP messages that one can send on a connection, two devices such as proxies usually have one or very few connections between them.

NAT vs. VoIP

One of the main challenges that VoIP protocols have encountered, and SIP is no exception, is the existence of NAT devices. NATs usually aggregate several IP addresses (in many cases within a private network) to a few external IP addresses, mapping different traffic from different IP addresses to different ports. (This is a very simplistic description of NATs; there are in fact several ways NATs can work, but this is a common one that's easy to describe). In order to map different addresses to IP and ports, NATs usually maintain a dynamic mapping table. If traffic from 172.16.1.1 with port 5060 maps to a public IP address with port 10000, the NAT will keep this mapping as long as it has a flow of packets from that address. If the flow stops, the NAT will remove this mapping after a configurable amount of time to allow another internal IP and port to use the external IP and port combination.

VoIP's problem arises because signaling protocols are as minimalistic as possible in terms of traffic. In order to allow the rest of the world to locate a device, the device first registers by sending a REGISTER request. A response from the registrar accepting this registration will usually tell the device that its registration is valid for a long time, most commonly an hour.

Now, suppose a NAT is located between the client and the server. When the REGISTER request is sent, the NAT maps the device's internal IP and port to an external one. When the response is sent back, the NAT has the proper mapping to the original IP and port. If the client does not send any packets to the server (for example, does not make a call), the NAT may remove the mapping it created during the registration. The outcome of this scenario is that incoming calls cannot reach the client. The proxy server receiving the request to the registered client cannot reach the internal IP address without the NAT mapping.

It's because of this NAT issue that RFC 5626 was introduced. This RFC defines the techniques that a client can use to maintain the NAT mapping. It introduces the concept of a *flow* that should be maintained by the registering client. The client sends two empty lines (carriage return and line feed, or CRLF) on connection-oriented flow (TCP and SCTP) and expects to receive from the server a single empty line as a response. For connection-less transports, the client maintains the flow by using STUN (defined by RFC 5389).

Transaction Layer

The SIP RFC divides the architecture into layers. We actually went through two of the layers in the discussion above: the first was the syntax and encoding layer that defines the message structure, and the second was the transport layer. Now it's time to inspect the contents of the SIP message by taking a look at the transaction layer.

THE SIP LAYERS

Every SIP message is associated with a single transaction. Similar to HTTP, messages are either requests or responses, but unlike HTTP, matching responses to requests is not simple. HTTP uses TCP as its transport, so you can match a response based on the order of the requests. But a SIP transaction can have more than a single response, and, in some cases, more than one request. When a SIP device sends a request, it acts as a user agent client (UAC). The recipient of the request, the one that sends the response, acts as a user agent server (UAS).

The layer above the transaction layer is named "transaction user" or TU. Let's look at a SIP request that a UAC can initiate:

```
REGISTER sip:arstechnica.com SIP/2.0
Via: SIP/2.0/UDP home.mynetwork.org;branch=z9hG4bKmq0Tgb
To: sip:me@arstechnica.com
From: sip:me@arstechnica.com;tag=m25caI4
Call-ID: n35nzlsdjfb3@home.mynetwork.org
CSeq: 153 REGISTER
Contact: sip:me@home.mynetwork.org
Max-Forwards: 70
```

We have already seen that *REGISTER* is the method (type of request), *sip:arstechnica.com* is the request-URI, and *SIP/2.0* is the version. All the headers above are the mandatory. At this point, we'll cover the headers that are important to the transaction layer, and we'll cover the rest of them when we get to the way proxies and registrars work. First, let's examine the Via header.

The Via header has a parameter called "branch" with an odd value. The first 7 letters (z9hG4bK) are fixed, and they help identify this as a SIP transaction based on RFC 3261. These letters, often referred to as the "magic cookie", would not appear with a request that is using the previous SIP RFC, which has different transaction matching rules. We'll only look the cases that have the magic cookie because it's very rare today to encounter an implementation that has not caught up with the latest spec.

After the seven letters, the rest is just a random string. Every time you see a different branch value it's a different transaction; conversely, if both messages have the same branch value, then they should be the same transaction. One exception to this rule is if the method of the CSeq header is different. This is because the CANCEL method uses the same branch value to identify which transaction you should cancel. So, to fully match two messages to the same transaction, both the branch and CSeq method have to match. Naturally, this means that responses to a request will have matching values.

Before moving on, one final note on the Via header. When we refer to this header, we actually refer to the first, or topmost, Via header. Via is one of those headers that can appear multiple times within a message. The reason for this will become clear in the proxy section, but it's important to note that you always match the transaction based on the first Via header and ignore the rest.

The UAS sends a response to an incoming request. SIP responses are divided into 6 different classes, and the first digit of the 3-digit response code identifies each class. A 1xx response means any response in the range of 100 to 199. The response types are:

- 1xx - **Provisional response**, which indicates that the request is handled, but without a final response yet. For example, 180 Ringing is a common provisional response.
- 2xx - **Successful response**. The most common one is 200 OK.
- 3xx - **Redirect response**. A client receiving this response would know the user moved to a different location. For example, a phone may redirect all its calls to a different address by responding back with a 302 Moved Temporarily.
- 4xx - **Client error**, which means that the request cannot be fulfilled and the sender should modify its request. For example, you can send 401 Unauthorized if the request does not contain the correct user credentials.
- 5xx - **Server error**, which usually indicates that the error is not related to the request, but to the state of the server or the server capabilities. For example, you would send 501 Not Implemented when receiving an unknown method.
- 6xx - **Global error**, which indicates the request cannot be fulfilled by any server. It would be rare to receive such responses, as it requires having global knowledge of the network.

SIP dedicates special attention to making sure the response is sent back to the same source IP that sent the request. This is, in fact, one of the roles of the transport layer, not the transaction layer. The transport layer does this by adding a "received" parameter to the top Via header of the request. Later, RFC 3581 defined a new parameter called "rport" to ensure that the response is sent back to the same originating port. Both of these additions were aimed at making SIP work over NAT. SIP's default behavior is to send the response back on the same connection of the request, but in case it fails to do so, it will attempt to open a new connection. Therefore, none of the layers can assume a single transaction uses a single connection.

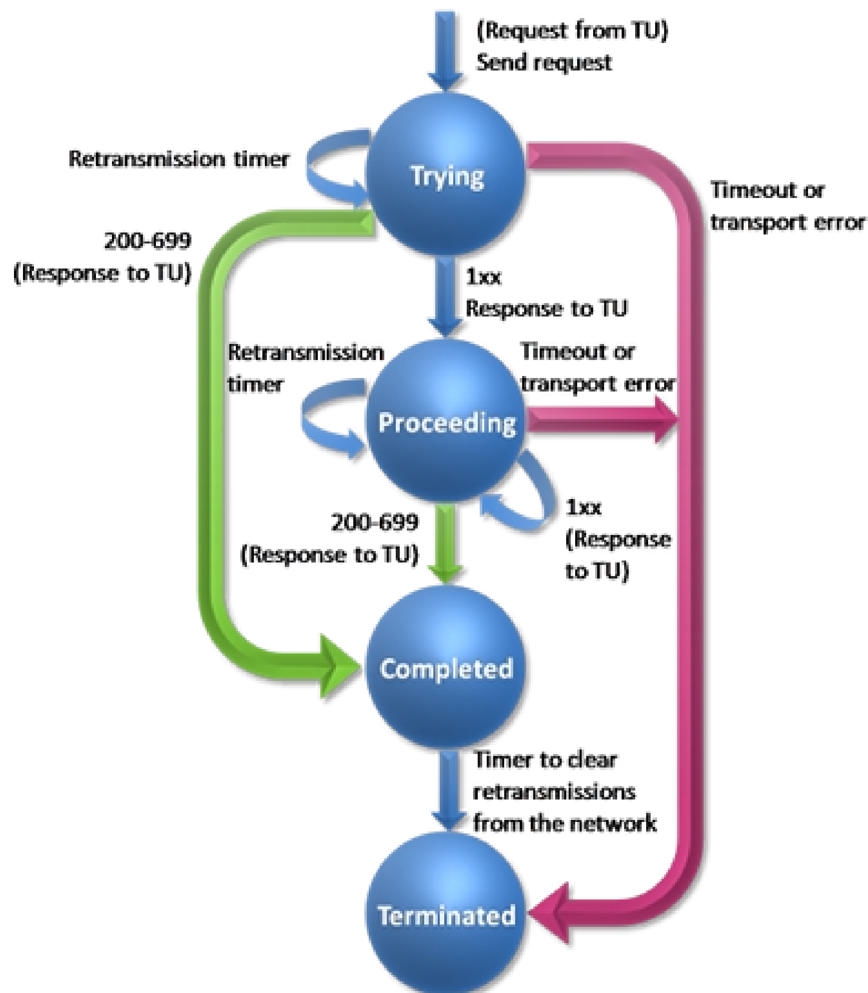
A possible SIP response to the request above is:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP home.mynetwork.org;branch=z9hG4bKmq0Tgb;received=172.16.75.2
To: sip:me@arstechnica.com;tag=q8K2f1zv
From: sip:me@arstechnica.com;tag=m25caI4
Call-ID: n35nzlsdjfb3@home.mynetwork.org
CSeq: 153 REGISTER
Contact: sip:me@home.mynetwork.org;Expires=3600
```

The example shows a successful response, but a UAS may choose to send an error response, such as the well-known "404 not found," in an instance where the user is not known. Both the UAC and UAS maintain a state machine for each transaction, and each state machine has timers. Timers are necessary in case the other side does not respond in time, and they're also required in case the layer above the transaction layer does not send a proper event and leaves the transaction open.

Ultimately, SIP has built each of its layers to be as decoupled as possible from the other layers, and an error in any one layer has minimal impact on the rest. This separation makes it easy for programmers to separate their software into smaller components.

The protocol distinguishes between 4 types of transactions, so it has 4 different types of state machines: client INVITE, client non-INVITE, server INVITE and server non-INVITE. We haven't mentioned the INVITE method yet, and for a good reason. INVITE is a method used to generate a call, and these lower layers do not maintain the call state. However, this transaction is different because calls have a 3-way handshake that affects the state-machine. Let's start with a diagram of the client non-INVITE transaction state-machine:

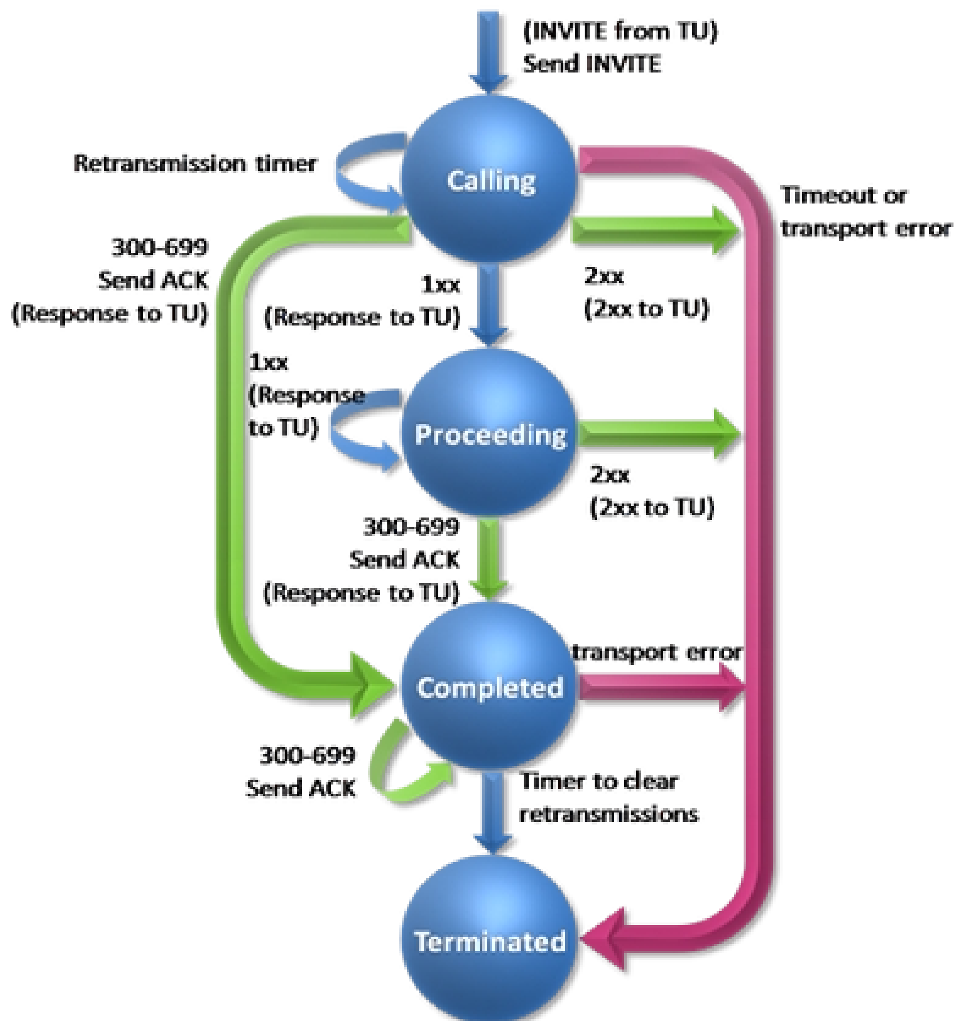


The Non-INVITE client transaction

Most of the timers are for retransmissions in UDP, and they are disabled in TCP. An additional timeout timer exists in case no response is received. Transactions normally exist for 32 seconds until they time out. The equivalent server state-machine is quite similar; it receives a request, sends it to the TU, sends the response back, and handles retransmissions if required. It should be noted that some of the non-INVITE transaction definitions were updated by RFC 4320.

Let's cover the 3-way handshake. The UAC sending the INVITE waits for a response, but this time to complete the handshake it sends an ACK request back to the server. ACK has no response, as it's the 3rd message in the handshake. This fact forces ACK to be an exception to many of the rules.

When a client receives a successful (2xx) response type, it means a call was created and it will send the ACK in a new transaction. A failure response (300-699) means the ACK will be on the same transaction. The reason for this lies in the behavior of the upper layers. We will see that proxies are not aware of a call state, and those that are stateful maintain just the transaction state. There are scenarios in which a proxy would need to ACK a failed response, but it cannot ACK a successful response because that would require understanding call-related information. The INIVITE client state machine is as follows:



The INVITE client transaction

User Location

Let's step out of the SIP layers and see what we have so far: using the layers, we can now create and receive SIP transactions.

One basic requirement in SIP is for phone devices to be able to register their location with a registrar. Using this registration information, we can send a request to a server and this request would reach the intended recipient. Let's go back to the previous example:

```
REGISTER sip:arstechnica.com SIP/2.0
Via: SIP/2.0/UDP home.mynetwork.org;branch=z9hG4bKmq0Tgb
To: sip:me@arstechnica.com
From: sip:me@arstechnica.com;tag=m25caI4
Call-ID: n35nzlsdjfb3@home.mynetwork.org
CSeq: 153 REGISTER
Contact: sip:me@home.mynetwork.org
Max-Forwards: 70
```

We already covered the first two lines, so let's now look at the rest.

- The *Contact* header tells the registrar the actual address of the device; many times you'll see an IP rather than a domain name in this field.
- The *To* header tells the registrar the address of record (usually referred as the AOR). AOR is the public SIP address, the same as my email address.
- The *From* header usually has the same value as the *To*. (It would be different in case someone is registering on behalf of someone else, but this case is rare.)
- Call-ID is an identifier that groups together a series of messages. Two different registrations should have different *Call-ID* values, but re-registrations should have the same Call-ID values. To differentiate the re-registrations, the client would increment the CSeq value.
- We'll leave the Max-Forwards explanation for the last section.

The client may also add an expiration value to the registration, either by adding a new *Expires* header or by adding an *Expires* parameter to the contact header. This is a recommended value to the server, because the server is the party that chooses the expiration time and sends it in the successful response. The expiration time may not be higher than the client-requested value, and if the value is below the minimum acceptable value then the registrar should reject the request.

Registering multiple user devices

A user is not bound to register with only one device, since it's possible that the user owns several SIP-capable devices and wants to be able to use the same SIP address simultaneously. A simple example involves two phones, one a desk phone and the other a cell phone. When someone calls the public SIP address, both phones would ring. To accomplish this, both devices register with the same AOR but different contact values. The registrar receiving these registrations maintains both associations. Requests arriving to the user AOR would fork to both devices. In some cases, it may make sense that both devices would create a session, but most of the time, this would be a call and thus the first device that answers would establish the call. Anyone who has a cell phone and a car phone with the same phone number is quite familiar with this scenario.

The fact that SIP lets several devices register originally worked out well, but then more complex scenarios started to come up. Suppose a user with two devices is in a conversation, and the person on the other end transfers the call to a third party. In SIP, it would mean the third party would send a request to the first user, but if it were to use the AOR, the request would fork to both devices. A device that is not active in the call would not be able to transfer a non-existent call. In this case, you might suggest using the device IP value, but many times, this will be a non-routable IP address, and the only way to reach it is via a server that has access to this private network.

For such scenarios, an extension called GRUU (Globally Routable User Agent URI) was introduced, defined in RFC 5627. When a user registers with a device that supports this extension, a header parameter named "+sip.instance" with a unique identifier is added to the contact header. The registrar creates two GRUUs for that instance: a public GRUU with the identifiable AOR, and a temporary GRUU to be used in case the user wants to make an anonymous call. Both these GRUUs are SIP addresses with a "gr" parameter. Now when there's a call with one of the devices, the contact address would have the GRUU SIP address. A transfer request would use this value, and because the server has a mapping between this GRUU and the actual device instance, the request reaches a single device participating in the call rather than the other devices.

Locating Servers

A SIP client that wishes to register to sip:arstechnica.com needs to resolve this address. SIP uses DNS to do that, but simply resolving the address to an IP is not enough. Therefore, SIP uses DNS to discover everything it needs to send the request. All the procedures I describe in this section are detailed in RFC 3263.

First, the client needs to discover the preferred transport type. For example, a client that supports UDP, TCP and SCTP performs a NAPTR query (defined in RFC 3403) for arstechnica.com. A response for such query may be:

```
;          order pref flags service      regexp  replacement
    IN NAPTR 50   50  "s"  "SIPS+D2T"     ""  _sips._tcp.arstechnica.com.
    IN NAPTR 90   50  "s"  "SIP+D2T"      ""  _sip._tcp.arstechnica.com.
    IN NAPTR 80   50  "s"  "SIP+D2S"      ""  _sip._sctp.arstechnica.com.
    IN NAPTR 100  50  "s"  "SIP+D2U"      ""  _sip._udp.arstechnica.com.
```

That means the server supports TLS, TCP, SCTP and UDP in this order. The *service* value determines the transport type. A client that does not support TLS will choose the second option, "SIP+D2T" which means "SIP over TCP." To use TCP, the client now needs to resolve _sip._tcp.arstechnica.com.

We have the transport type, but now the port is unknown. It is true that the default port is 5060, but this port will only be used if we cannot resolve a different port. To resolve the port, the client performs an SRV query (this type is defined in RFC 2782) for _sip._tcp.arstechnica.com. A possible response to this query may be:

```
;;          Priority Weight Port   Target
     IN SRV  0        1      5060   server1.arstechnica.com
     IN SRV  0        2      5070   server2.arstechnica.com
```

Now it's finally time to resolve the IP address. The client tries to send the request to server1.arstechnica.com, port 5060 via TCP. To find the correct IP address, the client performs an A query on IPv4 or AAAA query on IPv6. If the transaction times out, then the client should not stop at this point. It should try to send a new request to server2.arstechnica.com, port 5070 and transport TCP. If this also fails, then the client stops because it’s the last SRV record. So, SRV records are not just for resolving the port, but are also useful for server redundancy and load balancing.

Naturally, this elaborate process of doing different DNS queries takes time, and will significantly increase latency if you want to send many SIP messages. Hence, the assumption is that the client caches these DNS results. This is a reasonable assumption, since HTTP clients also cache DNS to improve performance. There are other possibilities for overriding these queries. The first is to use a numeric IP address, but this is not recommended for obvious reasons. The other option is to specify the values in the SIP URI itself. One can specify its address as: sip:arstechnica.com:5060;transport=tcp. This is also not recommended, since changing the values requires changing the URI. Furthermore, the server redundancy supported by the SRV records would not be available.

Proxies

Now let's look at SIP proxies to see how the different pieces fall together. You rarely send signaling messages directly between phone devices. Usually, there's at least one, if not several, proxies along the signaling path, and these proxies are designed to be aware of transactions, not calls. (Such a design is more scalable.)

Proxies act as both UAC and UAS. An incoming request goes to the proxy UAS side, and the proxy then creates a new transaction as a UAC. Responses reach the UAC, and the proxy generates responses as a UAS. Therefore, for an incoming transaction and a corresponding outgoing transaction, the transaction layer maintains two state-machines, and it's the proxy's job to manage the interaction of these two transactions.

Some proxies work in conjunction with a registrar and have access to a shared database. It's such a proxy's job to retrieve a user's public AOR and to resolve its registered contact address. Other proxies simply route the messages. I should note at this point that this section focuses on stateful proxies. The standard also defines stateless proxies, so some of the text here would not apply to those server types.

The Via header, forking, loop prevention

When we went through the transport layer, I added a vague description of the top Via header. Now it's time to address this header in more detail.

A SIP message may contain more than a single Via header. When a proxy constructs a request for a new transaction, it takes the existing message and adds an additional Via header above the existing, topmost Via header. This Via header has a new "branch" parameter value, thus signifying that this is a new transaction and that its address is the proxy's UAC-side address. The UAS receiving this request would send the response based on the top Via header, thereby ensuring that the response goes back to the proxy. If the proxy sends back the response, it sends it without its own Via header, because the original transaction is a different one and has a different top Via header with a different "branch" value. Proxies use a similar mechanism with route and record-route headers, but this subject is covered in the next part of the article.

I mentioned previously that a single user can register with multiple devices. In order to send a request to multiple targets, the proxy forks a request. An incoming request may result in several outgoing requests to different targets, each containing a different branch value. Proxies will not forward every failed final response to the UAC because the first final response would cause the UAC to close the transaction. Therefore, the proxy collects the error responses, and if all the outgoing transactions fail, it chooses the most appropriate failed response. A successful response is sent back to the UAC immediately.

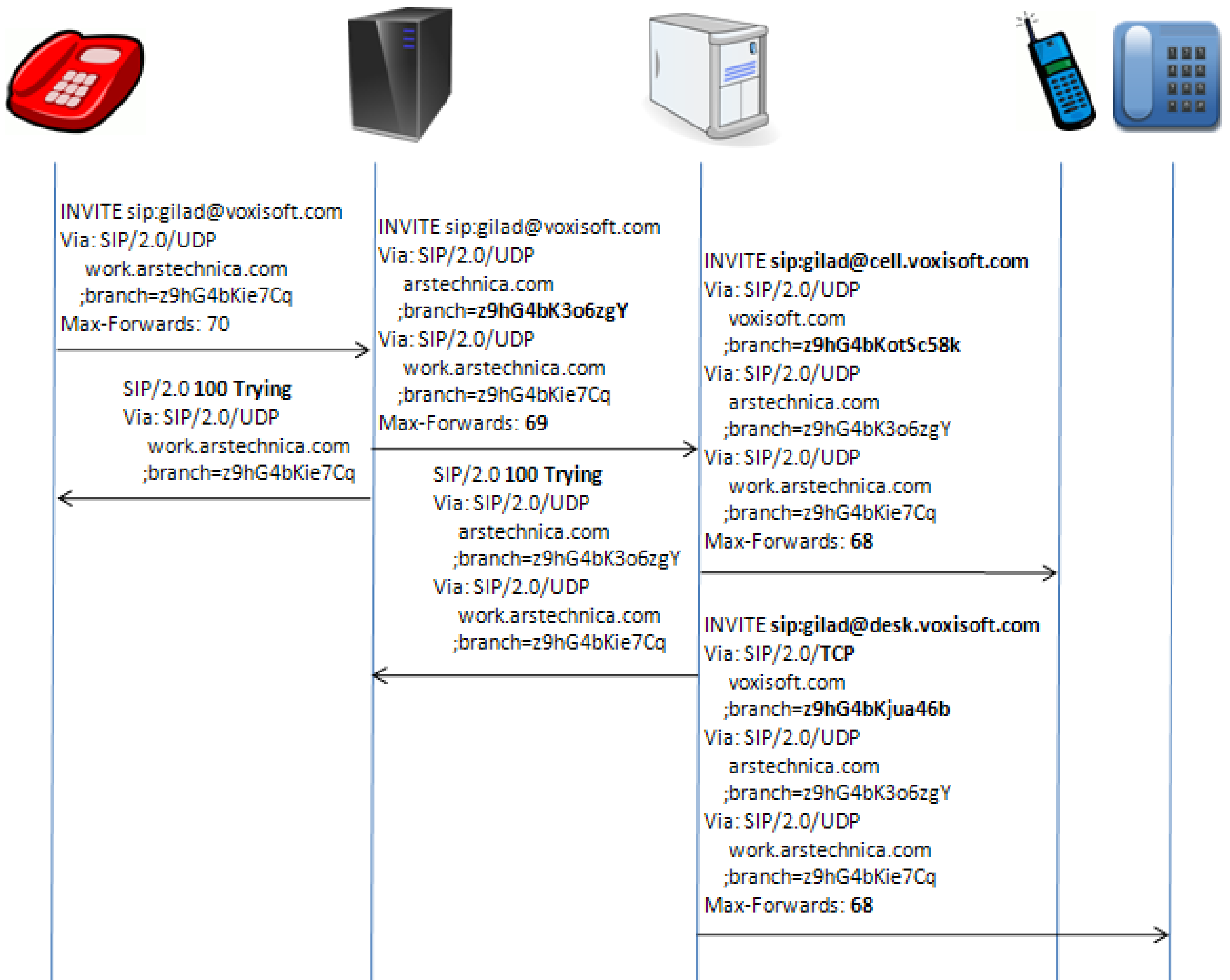
This is why we saw that an ACK request is part of the transaction for a failed response. For INVITE transactions, the proxy sends out an ACK for failed responses and does not wait until the other responses arrive. On the other hand, the proxy cannot ACK a successful 2xx response to INVITE since this means a new call was created, but the caller is not aware of this yet.

To prevent loops, proxies use the Max-Forwards header. This is the last of the mandatory headers that I skipped in earlier sections. The initial request is sent out with a value, most commonly 70, and every proxy that forwards this request deducts 1 from the value in the request that it sends out. If the value of the Max-Forwards reaches 0 before it reaches the final destination, the UAS sends back a 483 (Too Many Hops) response. It should be noted that a possible amplification vulnerability was later discovered for forking proxies. This was addressed by RFC 5393, which changed some loop detection mechanisms and introduced a new header called Max-Breadth that reduces the number of possible forks a message goes through.

Finally, proxies send out a 100 (Trying) provisional response when they receive a request whose response takes more than 200 ms. This prevents the UAC from retransmitting the request, and it also prevents a time-out event. 100 is a response that is generated by the proxy. If we have more than one proxy, the proxy that receives the 100 message will not forward it. This is because it should have sent its own 100 message prior to receiving it.

An example using proxies

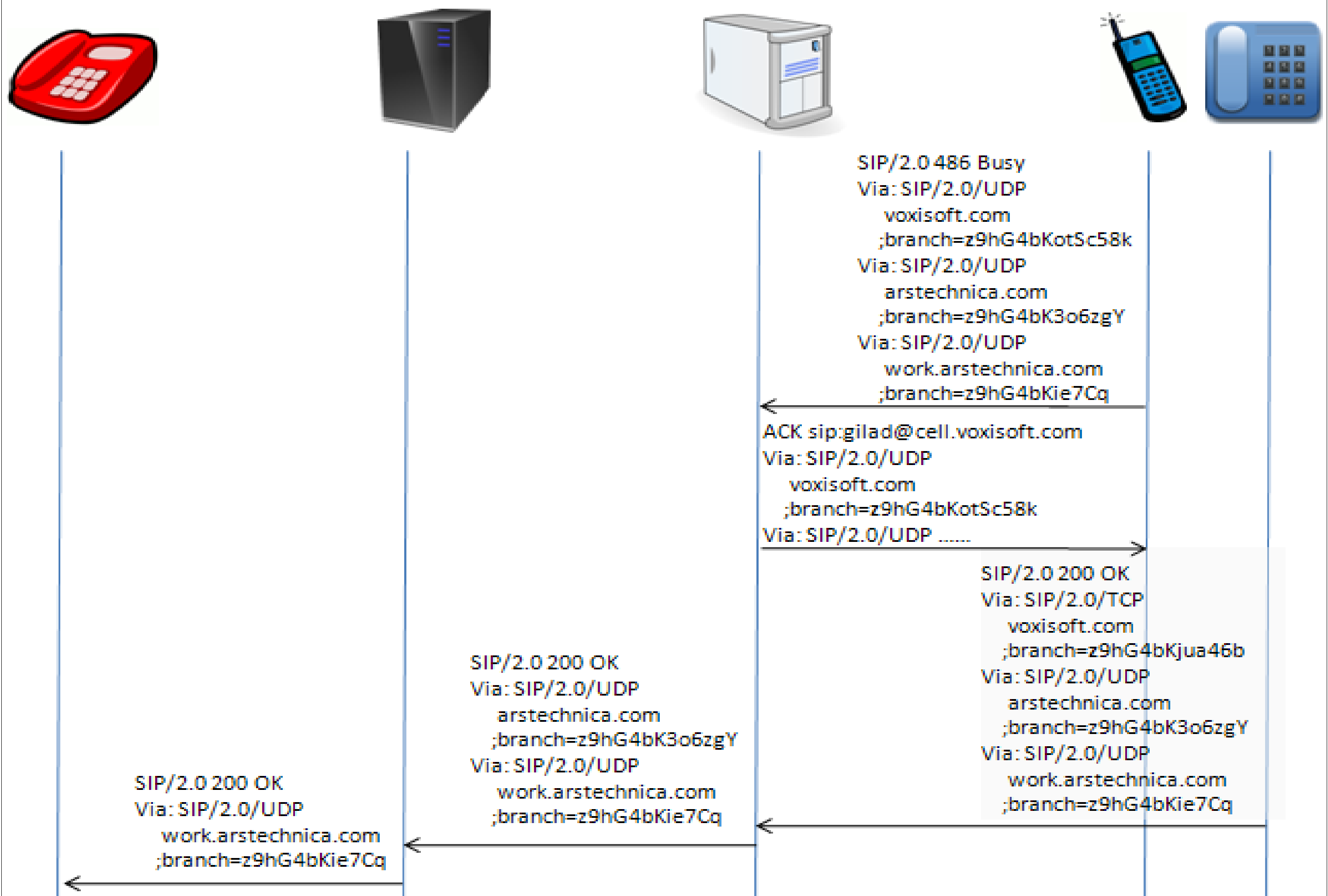
Let's look at an example to wrap up this discussion (non-essential details elided, including the SIP version in the first line of the request and some headers that will be discussed when we cover calls):



This message is sent from Ars Technica's network and reaches the arstechnica.com SIP proxy. The proxy sends out the message to the voxisoft.com SIP proxy and returns a 100 response. Voxisoft's proxy has two registrations and forks the request to two devices (while also sending out a 100 response). Notice that all new requests have a new Via header with a new branch parameter. Also, note that the second message is using TCP as a transport. This is a valid scenario, as we previously discussed, since both transactions have different state machine and one of them may discover a different transport when performing a NAPTR query.

At this point, one device might answer with, for example, 486 (Busy), but the proxy does not forward it because it has another forked message pending; so it just sends an ACK. The ACK has the same branch value since it's the same transaction. ACK is sent only for the INVITE case; if this were a different method then ACK wouldn't be used. The second device sends a

200 OK and this message is sent all the way back to the initiating client. The following illustration shows this process in action:



Finally, the client on the left side sends out an ACK for the 200 OK. This ACK is a new transaction and therefore has a new branch value. The proxies forward the request to the destination, again adding a Via header for each hop. This time ACK does not fork; we will see this mechanism in the next article.

Summary

In this article, we've covered the foundation layers of SIP, including its message structure, transport layer and transaction layer. We've also covered the way SIP registrars and proxies work based on these layers. The discussion so far should give you a good foundation for understanding this protocol. In the next part of this series of articles, we'll complete this discussion by going through the definitions of SIP calls and additional services.