Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 9, 2020

J. Rosenberg
Five9
C. Jennings
Cisco Systems
A. Minessale
Signalwire/Freeswitch

July 8, 2019

Real Time Internet Peering Protocol
draft-rosenbergjennings-dispatch-ripp-01

Abstract

This document specifies the Realtime Internet Peering Protocol (RIPP). RIPP is used to provide telephony peering between a trunking provider (such as a telco), and a trunking consumer (such as an enterprise, cloud PBX provider, cloud contact center provider, and so on). RIPP is an alternative to SIP, SDP and RTP for this use case, and is designed as a web application using HTTP/3. Using HTTP/3 allows trunking consumers to more easily build their applications on top of cloud platforms, such as AWS, Azure and Google Cloud, all of which are heavily focused on HTTP based services. RIPP also addresses many of the challenges of traditional SIP-based trunking. Most notably, it mandates secure caller ID via STIR, and provides automated trunk provisioning as a mandatory protocol component. RIPP supports both direct and "BYO" trunk configurations. Since it runs over HTTP/3, it works through NATs and firewalls with the same ease as HTTP does, and easily supports load balancing with elastic cluster expansion and contraction, including auto-scaling - all because it is nothing more than an HTTP application. RIPP also provides built in mechanisms for migrations of calls between RIPP client and server instances, enabling failover with call preservation.

This Internet-Draft will expire on January 9, 2020.

Table of Contents

1. Introduction

1.1. Background

Cloud computing platforms, such as those provided by Amazon, Azure, and Google, have now become mainstream for the development of software applications. These platforms are targeted at enabling web applications, and as such many of their features are based on the usage of HTTP.

One example are HTTP load balancers. Cloud computing platforms provide highly scalable, geographically distributed, redundant load balancers. These load balancers can monitor the state of downstream servers and can uniformly distribute load amongst them. The load balancers can compensate for failure of individual nodes and send new traffic to other nodes.

Autoscaling is another example. The cloud computing platforms can automatically add new instances of a server backend, or remove them, and automatically configure the load balancers to include them in the pool of available servers.

Yet another example is Kubernetes, which allows web-based applications to be deployed into containers (typically Docker), with load balancing, scaling, and HTTP request routing.

Another example are HTTP tracing tools, which facilitate the tracing of requests through distributed microservices. These tools can autogenerate sequence diagrams and facilitate in troubleshooting.

Yet another example are API gateways (such as APIGee and Kong), which provide authentication and authorization, provisioning of applications, rate limiting, analytics, sandboxing for testing, embedded documentation, and so on.

And yet another example are denial-of-service prevention techniques, typically done using BGP peering and re-routing. Though in principle these techniques can work for VoIP, they are deployed in conjunction with the load balancers which represent the entry point into these cloud provider networks. Consequently, the protections these cloud providers offer do not extend to applications which merely use these platforms for virtual machines.

A more recent technology are service meshes, such as Istio, which utilize sidecar HTTP proxies to facilitate inter-service communications. These systems come with robust control planes which enable additional routing features, such as canary deploys, percentage based routing, and so on.

1.2. Problem Statement

Unfortunately, there are many applications being deployed into these cloud platforms which require interconnection with the public switched telephone network (PSTN). Examples of such applications include cloud PBXs, cloud contact centers, cloud meetings applications, and so on. Furthermore, commerce websites would like to allow customers to call into the telephone network for customer support.

In order for these applications to connect to the PSTN, they typically deploy Session Initiation Protocol (SIP) [RFC3261] based servers - SBCs, SIP proxies, and softswitches, to provide this interconnection. Unfortunately, SIP based applications cannot make use of the many capabilities these cloud platforms afford to HTTP based applications. These SIP servers are usually deployed on bare metal or VMs at best. Application developers must build their own load balancing, HA, failover, clustering, security, and scaling technologies, rather than using the capabilities of these platforms.

This has creating a barrier to entry, particularly for applications such as websites which are not expert in VoIP technologies. Furthermore, it has meant that VoIP applications have been unable to take advantage of the many technology improvements that have come to

networking and protocol design since the publication of RFC 3261 in 2002.

In addition, SIP trunking has suffered from complex provisioning operations, oftentimes requiring the exchange of static IPs and ports. These operations are almost never self-service and consequently, SIP trunk turn ups can take weeks. Finally, perhaps the biggest challenge with SIP trunking has been its abuse for injecting robocalls.

1.3. Solution

The goal of RIPP is to enable one administrative domain to send and receive voice calls with another domain. In this regard, RIPP replaces the usage of SIP, SDP offer/answer [RFC3264] and RTP [RFC3550] for this particular use case. RIPP does not actually deprecate or replace SIP itself, as it covers only a small subset of the broader functionality that SIP provides. It is designed to be the minimum protocol required to interconnect voice between a trunking provider and a domain wishing to access trunking services.

In order to make use of new HTTP based technologies as described above, RIPP uses HTTP/3 [I-D.ietf-quic-http], but is not an extension to it. The goal is to ride the coattails of advancement in HTTP based technologies without requiring them to do anything special for the benefit of VoIP. This means that RIPP inherits the benefits of classic HTTP deployments - easy load balancing, easy expansion and contraction of clusters (including auto-scaling), standard techniques for encryption, authentication, and denial-of-service prevention, and so on.

RIPP also includes a built-in mechanism for provisioning, as a mandatory component of the specification. This enables RIPP trunks to be self-provisioned through web portals, and instantly turned on in production. This will help accelerate the adoption of telecommunications services across the web.

1.4. Why Now?

The idea of re-converging HTTP and SIP is certainly not new, and indeed has been discussed in the hallways of IETF for many years. However, several significant limitations made this previously infeasible:

1. HTTP utilized TCP, which meant that it created head-of-line blocking which would delay lost packets rather than just discard them. This will often provide intolerable latency for VoIP.

2. HTTP was request response, allowing the client to send requests and receive a response. There as no way for a server to asynchronously send information to the client in an easy fashion.

HTTP2 [RFC7540] addressed the second of these with the introduction of pushes and long running requests. However, its usage of TCP was still a problem. This has finally been addressed with the arrival of QUIC [I-D.ietf-quic-transport] and HTTP/3. QUIC is based on UDP, and it introduces the concept of a stream that can be set up with zero RTT. These streams are carried over UDP, and though are still reliable, there is no head of line blocking across streams. This change has made it possible for HTTP to support VoIP applications.

2. Solution Requirements

The protocol defined here is based on the following requirements:

REQ1: The solution shall not require extensions or modifications to HTTP/3.

REQ2: The solution shall work with both L4 and L7 HTTP load balancers

REQ3: The solution shall work in ways that are compatible with best practices for load balancers and proxies supporting HTTP/3, and not require any special changes to these load balancers in order to function.

REQ4: The solution should hide the number of servers behind the load balancer, allow the addition or removal of servers from the cluster at will, and not expose any of this information to the peer

REQ5: The solution shall enable the usage of autoscaling technologies used in cloud platforms, without any special consideration for RIPP - its just a web app

REQ6: The solution shall provide call preservation in the face of failures of the server or client. It is acceptable for a brief blip of media due to transient packet loss, but thats it

REQ7: The solution shall support built-in migration, allowing a server to quickly shed load in order to be restarted or upgraded, without any impact to calls in progress

REQ8: The solution will be easy to interoperate with SIP

REQ9: The solution shall be incrementally deployable - specifically it must be designed for easy implementation by SBCs and easy

deployment by PSTN termination and origination providers who do not utilize cloud platforms

REQ10: The solution shall require authentication and encryption, with no opportunity to disable them. Furthermore, it will require secure callerID, with no provision for insecure callerID

REQ11: The solution shall provide low latency for media

REQ12: The solution shall support only audio, but be extensible to video or other media in the future

REQ13: The solution must support secure caller ID out of the gate and not inherit any of the insecure techniques used with SIP

REQ14: The solution shall include mandatory-to-implement provisioning operations

3. Design Approaches

To meet the requirements stated above, RIPP makes several fundamental changes compared to SIP. These changes, and their motivations, are described in the sections below.

3.1. HBH, not E2E

SIP was designed as an end-to-end protocol. As such, it explicitly incorporates features which presume the existence of a network of elements - proxies and registrars in particular. SIP provides many features to facilitate this - Via headers, record-routing, and so on.

HTTP on the other hand - is strictly a hop-by-hop technology. Though it does support the notion of proxies (ala the CONNECT method for reverse proxies), the protocol is fundamentally designed to be between a client and an authoritative server. What happens beyond that authoritative server is beyond the scope of HTTP, and can (and often does) include additional HTTP transactions.

Consequently, in order to reside within HTTP, RIPP follows the same pattern and only concerns itself with HBH behaviours. Like HTTP, a RIPP server can of course act as a RIPP client and further connect calls to downstream elements. However, such behavior requires no additional specification and is therefore not discussed by RIPP.

3.2. Client-Server, not Agent-to-Agent

SIP is based fundamentally on the User Agent, and describes the communications between a pair of user agents. Either user agent can initiate requests towards the other. SIP defines the traditional role of client and server as bound to a specific transaction.

HTTP does not operate this way. In HTTP, one entity is a client, and the other is a server. There is no way for the server to send messages asynchronously towards the client. HTTP/3 does enable two distinct techniques that facilitate server messaging towards the client. But to use them, RIPP must abide by HTTP/3 rules, and that means distinct roles for clients and servers. Clients must always initiate connections and send requests, not servers.

To handle this RIPP, specifies that the domain associated with the caller implements the RIPP client, and the domain receiving the calls is the RIPP server. For any particular call, the roles of client and server do not change. To facilitate calls in either direction, a domain can implement both RIPP client and RIPP server roles. However, there is no relationship between the two directions.

3.3. Signaling and Media Together

One of the most fundamental design properties of SIP was the separation of signalling and media. This was fundamental to the success of SIP, since it enabled high quality, low latency media between endpoints inside of an enterprise or consumer domain.

This design technique is quite hard to translate to HTTP, especially when considering load balancing and scaling techniques. HTTP load balancing is effective because it treats each request/response pair as an independent action which can route to any number of backends. In essence, the request/response transaction is atomic, and consequentially RIPP needs to operate this way as well.

Though SIP envisioned that signalling and media separation would also apply to inter-domain calls, in practice this has not happened. Inter-domain interconnect - used primarily for interconnection with the PSTN - is done traditionally with SBCs which terminate and re-originate media. Since this specification is targeted solely at these peering use cases, RIPP fundamentally combines signalling and media together on the same connection. To ensure low latency, it uses multiple independent request/response transactions - each running in parallel over unique QUIC streams - to transmit media.

3.4. URIs not IPs

SIP is full of IP addresses and ports. They are contained in Via headers, in Route and Record-Route headers. In SDP. In Contact headers. The usage of IPs is one of the main reasons why SIP is so difficult to deploy into cloud platforms. These platforms are based on the behavior of HTTP which has been baed on TCP connections and therefore done most of its routing at the connection layer, and not the IP layer.

Furthermore, modern cloud platforms are full of NATs and private IP space, making them inhospitable to SIP based applications which still struggle with NAT traversal.

HTTP of course does not suffer from this. In general, "addressing", to the degree it exists at all, is done with HTTP URIs. RIPP follows this pattern. RIPP - as a web application that uses HTTP/3 - does not use or convey any IP addresses or ports. Furthermore, the client never provides addressing to the server - all traffic is sent in the reverse direction over the connection.

3.5. OAuth not MTLS or private IP

When used in peering arrangements today, authentication for the SIP connections is typically done using mutual TLS. It is also often the case that security is done at the IP layer, and sometimes even via dedicated MPLS connections which require pre-provisioning. Unfortunately, these techniques are quite incompatible with how modern cloud platforms work.

HTTP - due to its client-server nature, uses asymmetric techniques for authentication. Most notably, certificate based authentication is done by the client to verify that it is speaking to the server it thinks it should be speaking to. For the server to identify the client, modern platforms make use of OAuth2.0. Though OAuth is not actually an authentication protocol, the use of OAuth has allowed authentication to be done out of band via separate identity servers which produce OAuth tokens which can then be used for authentication of the client.

Consequently, RIPP follows this same approach. For each call, one domain acts as the client, and the other, as the server. When acting as a server, the domain authenticates itself with TLS and verifies the client with OAuth tokens. For calls in the reverse direction, the roles are reversed.

To make it possible to easily pass calls in both directions, RIPP allows one domain to act as the customer of another, the trunking

provider. The customer domain authenticates with the provider and obtains an OAuth token using traditional techniques. RIPP then allows the customer domain to automatically create a bearer token for inbound calls and pass it to the provider.

3.6. TLS not SRTP or SIPS

SIP has provided encryption of both signalling and media, through the usage of SIP over TLS and SIPS, and SRTP, respectively. Unfortunately, these have not been widely deployed. The E2E nature of SRTP has made keying an ongoing challenge, with multiple technologies developed over the years. SIP itself has seen greater uptake of TLS transport, but this remains uncommon largely due to the commonality of private IP peering as an alternative.

Because of the HBH nature of RIPP, security is done fundamentally at the connection level - identically to HTTP. Since media is also carrier over the HTTP connection, both signalling and media are covered by the connection security provided by HTTP/3.

Because of the mandatory usage of TLS1.3 with HTTP/3, and the expected widespread deployment of HTTP/3, running VoIP on top of HTTP/3 will bring built-in encryption of media and signalling between peering domains, which is a notable improvement over the current deployment situation. It is also necessary in order to utilize HTTP/3.

Because of this, RIPP does not support SRTP. If a client receives a SIP call with SRTP, it must terminate the SRTP and decrypt media before sending it over RIPP. This matches existing practice in many cases.

3.7. Authenticated CallerID

Robocalling is seeing a dramatic rise in volume, and efforts to combat it continue. One of the causes of this problem is the ease of which SIP enables one domain to initiate calls to another domain without authenticated caller ID.

With RIPP, we remedy this by requiring the client and servers to implement STIR. Since RIPP is meant for peering between providers (and not client-to-server connections), STIR is applicable. RIPP clients must either insert a signed passport, or pass one through if it exists. Similarly, RIPP servers must act as verifying parties and reject any calls that omit a passport.

- CJ - Need to check we have all the things needed in an Passport.

3.8. Calls Separate from Connections

In SIP, there is a fuzzy relationship between calls and connections. In some cases, connection failures cause call terminations, and vice a versa.

HTTP, on the other hand, very clearly separates the state of the resource being manipulated, with the state of the HTTP connection used to manipulate it. This design principle is inherited by RIPP. Consequently, call state on both client and server exist independently from the connections which manipulate them. This allows for greater availability my enabling connections for the same call to move between machines in the case of failures.

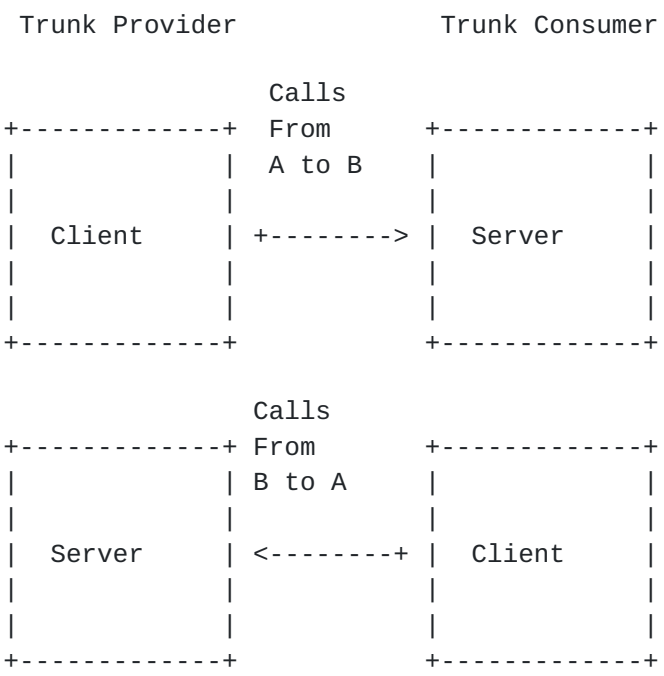
3.9. Path Validation, not ICE

HTTP/3 is designed to work through NAT as a client-server protocol. It has built in techniques for dealing with NAT re-bindings, IP address changes due to a client moving between networks (e.g., wifi to cellular data). It has built in path validation that ensures that HTTP cannot be used for amplification attacks.

SIP has, over the years, solved these problems to some degree, but not efficiently nor completely. To work with HTTP, RIPP must utilize the HTTP approaches for these problems. Consequently, RIPP does not utilize ICE and has no specific considerations for NAT traversal, as these are handled by HTTP/3 itself.

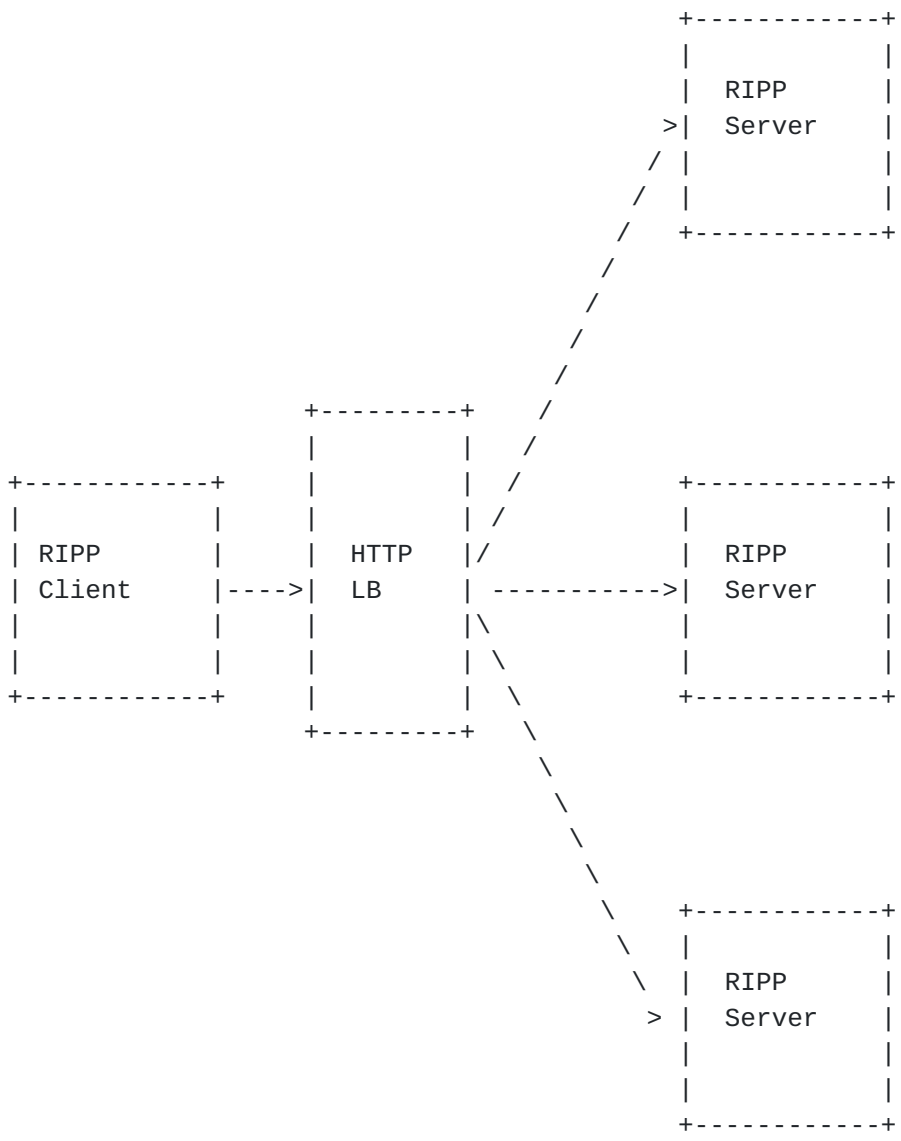
4. Reference Architecture

The RIPP reference architecture is shown in Figure 1.



RIPP is used between a RIPP trunk provider and a RIPP trunk consumer. Both entities implement the RIPP client and RIPP server roles; the latter to receive calls, and the former to send them.

RIPP is also designed such that all communications between the a RIPP client and the RIPP server can easily sit behind a typical HTTP load balancer, as shown below:



Since both the trunk provider and trunk consumer implement the client and server roles, both entities will typically have a load balancer - perhaps a server component, or a cloud-based service, used to receive incoming calls. This is not required, of course. It is worth restating that this load balancer is NOT specific to RIPP - it is any off-the-shelf HTTP load balancer which supports HTTP/3. No specific support for RIPP is required. RIPP is just a usage of HTTP.

Because RIPP clients and servers are nothing more than HTTP/3 applications, the behavior or RIPP is specified entirely by describing how various RIPP procedures map to the core HTTP/3 primitives available to applications - opening connections, closing connections, sending requests and responses, receiving requests and responses, and setting header fields and bodies. That's it.

5. Terminology

This specification follows the terminology of HTTP/3 - specifically:

RIPP Client: The entity that initiates a call, by acting as an HTTP client.

RIPP Server: The entity that receives a call, by acting as an HTTP server.

RIPP Connection: An HTTP connection between a RIPP client and RIPP server.

RIPP Endpoint: Either a RIPP client or RIPP server.

RIPP Peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

This specification defines the following additional terms:

RIPP Trunk: A container for calls between a trunking provider and trunking consumer. A RIPP trunk is identified by a pair of URI - the RIPP Trunk Provider URI (hosted by the trunking provider) and the RIPP Trunk Consumer URI (hosted by the trunking consumer). RIPP trunks act as a unit of policy and capabilities, including rules such as rate limits, allowed phone numbers, and so on.

Call: A VoIP session established by a RIPP client for the purposes of exchanging audio and signalling information. A call is always associated with a RIPP trunk.

Trunking Consumer: An administrative entity that utilizes trunking services from the trunking provider. The relationship between the trunking consumer and trunking provider is static and does not vary from call to call. (e.g., Verizon would be the trunking provider to an enterprise consumer, and the enterprise would be the trunking consumer of Verizon. A trunking consumer implements a RIPP client to initiate calls to the trunking provider, and a RIPP server to receive them.

Trunking Provider: The administrative entity that provides telephony trunking services to the trunking consumer. The relationship between the trunking consumer and trunking provider is static and does not vary from call to call. (e.g., Verizon would be the trunking provider to an enterprise, and the enterprise would be the trunking customer of Verizon. The trunking provider implements a RIPP server to

receive calls from the trunking consumer, and a RIPP client to send calls to the trunking consumer

Trunking Customer: The administrative entity which purchases trunking services from the trunking provider. The trunking customer may be the same as the trunking consumer - such as an enterprise purchasing and then consuming trunking services from a telco. Or, it can be different - such as an enterprise purchasing trunking services from a telco, and then authorizing a cloud PBX or cloud contact center provider to consume those trunking services on their behalf.

RIPP Trunk Provider URI: An HTTP URI hosted by the trunking provider, which represents the RIPP trunk from its perspective.

RIPP Trunk Consumer URI: An HTTP URI hosted by the trunking consumer, which represents the RIPP trunk from its perspective.

Byway: A bidirectional byte stream between a RIPP provider and consumer. A Byway passes its data through a long-running HTTP request and a long-running HTTP response. Byways are used for signalling and media.

6. Overview of Operation

RIPP begins with a configuration phase. This configuration phase occurs when an OAuth2.0 client application (such as a softswitch, cloud PBX, cloud contact center, etc) wishes to enable trunking customers to provision RIPP trunks against a trunking provider. The trunking provider acts as the resource provider in OAuth2.0 parlance. Consequently, The configuration phase is identical to the way in which client applications register with resource providers in OAuth2.0, the details of which are beyond the scope of this specification, but expected to follow existing best practices used by web applications.

The next step is provisioning. Once a trunking customer has obtained access to services from a trunking provider (by purchasing them , for example), the trunking customer can perform provisioning. Provisioning is the process by which a trunking customer connects a RIPP trunk from a trunking provider to trunking consumer. Provisioning is accomplished using OAuth2.0 code authorization techniques. In the case of RIPP, the OAuth resource owner is the trunking customer. The OAuth client is the RIPP implementation within the trunking consumer. The resource server is the RIPP implementation in the trunking provider.

To provision a RIPP trunk, the trunking customer will visit a web page hosted by the trunking consumer, and typically click on a button

labeled with their trunking provider. This will begin the OAuth 2 authorization code flow. The trunking customer will authenticate with the trunking provider. The trunking provider authorizes the access, generates an authorization code, and generates a RIPP trunk provider URI. The provider URI is included in a new OAuth parameter defined by this specification, and is returned as a parameter in the authorization response. The trunking consumer trades the authorization code for a refresh and access token, and stores the provider URI. Finally, the trunking consumer mints a bearer token associated with the new RIPP trunk, and also mints a RIPP trunk consumer URI for receiving calls from the provider on this trunk. Both of these are passed to the trunking provider via a POST operation to /consumerTrunk on the RIPP trunk provider URI.

The usage of the OAuth2.0 flows enables the trunking consumer and trunking customer to be the same (i.e., a cloud PBX provider purchases services from a telco), or different (i.e., an enterprise customer has purchased trunking services from a telco, and wishes to provision them into a cloud contact center that acts as the trunking consumer). The latter is often referred to informally as "BYOSIP" in traditional SIP trunking and is explicitly supported by RIPP using OAuth2.0.

Once provisioned, both sides obtain capability declarations for the RIPP trunk by performing a GET to /capAdv of its peers trunk URI. The capabilities declaration is a simple document, whose syntax is described in Section Section 10. It conveys the receive capabilities of the entity sending it, and includes parameters like maximum bitrate for audio. This process is optional, and each parameter has a default. Either side can update its capabilities for the RIPP trunk at any time, and trigger a fresh GET via an HTTP push. Capability declarations occur outside of a call, are optional, and convey static receive capabilities which are a fixed property of the RIPP trunk. Consequently, capability declaration is significantly different from SDP offer/answer.

Either the trunking consumer or provider can initiate calls by posting to the /calls on RIPP trunk URI of its peer. The request contains the target phone number in the request URI and an Identity header field in the HTTP Request. The Identity header field is identical in syntax and semantics to the SIP Identity header field defined in [RFC8224], just carried in HTTP instead of SIP. This request returns a globally unique call URI in the Location header field of a 201 response sent by the server. Typically the response will also include a session cookie, bound to the call, to facilitate sticky session routing in HTTP proxies. This allows all further signalling and media to reach the same RIPP server that handled the

initial request, while facilitating failover should that server go down.

Once a call has been created, a pair of long-lived HTTP transactions is initiated from the client to the server for purposes of signalling. One is a GET, retrieving call events from its peer. THe other is a PUT, sending call events to its peer. Each of these produces a unidirectional data stream, one in the forwards direction, one in the reverse. These are called byways. HTTP/3 ensures zero RTT for setup of these byways.

Signaling commands are encoded into the signalling byway using streaming JSON in both directions. Each JSON object encodes an event and its parameters. Events are defined for alerting, connected, ended, migrate, keepalive, and transfer-and-takeback.

The media byways carry a simple binary encoding in both directions. Even though data can flow in both directions, a media byway is unidirectional in terms of media transmission. A forward media byway carries media from the client to the server, and a reverse byway carries media from the server to the client. To eliminate HOL blocking for media, a media packet is sent on a media byway when it is first established. After the first packet, the client cannot be sure a subsequent packet will be delayed due to the ordering guarantees provided by HTTP/3 within a stream. To combat this, both sides acknowledge the receipt of each packet using an ACK message sent over the media byways, in the opposite direction of the media. Consequently, in a forward media byway, ACK messages are carried from server to client, and in a reverse media byway, they are carried from client to server. Once a media packet is acknowledged, the media byway can be used once again without fear of HOL blocking. Because each media packet is acknowledged independently, each side can compute statistics on packet losses and delays. Consequently, the equivalent of RTCP sender and receiver reports are not needed.

RIPP defines some basic requirements for congestion control at the client side. Specifically, clients drop media packets if there are too many media byways in the blocked state.

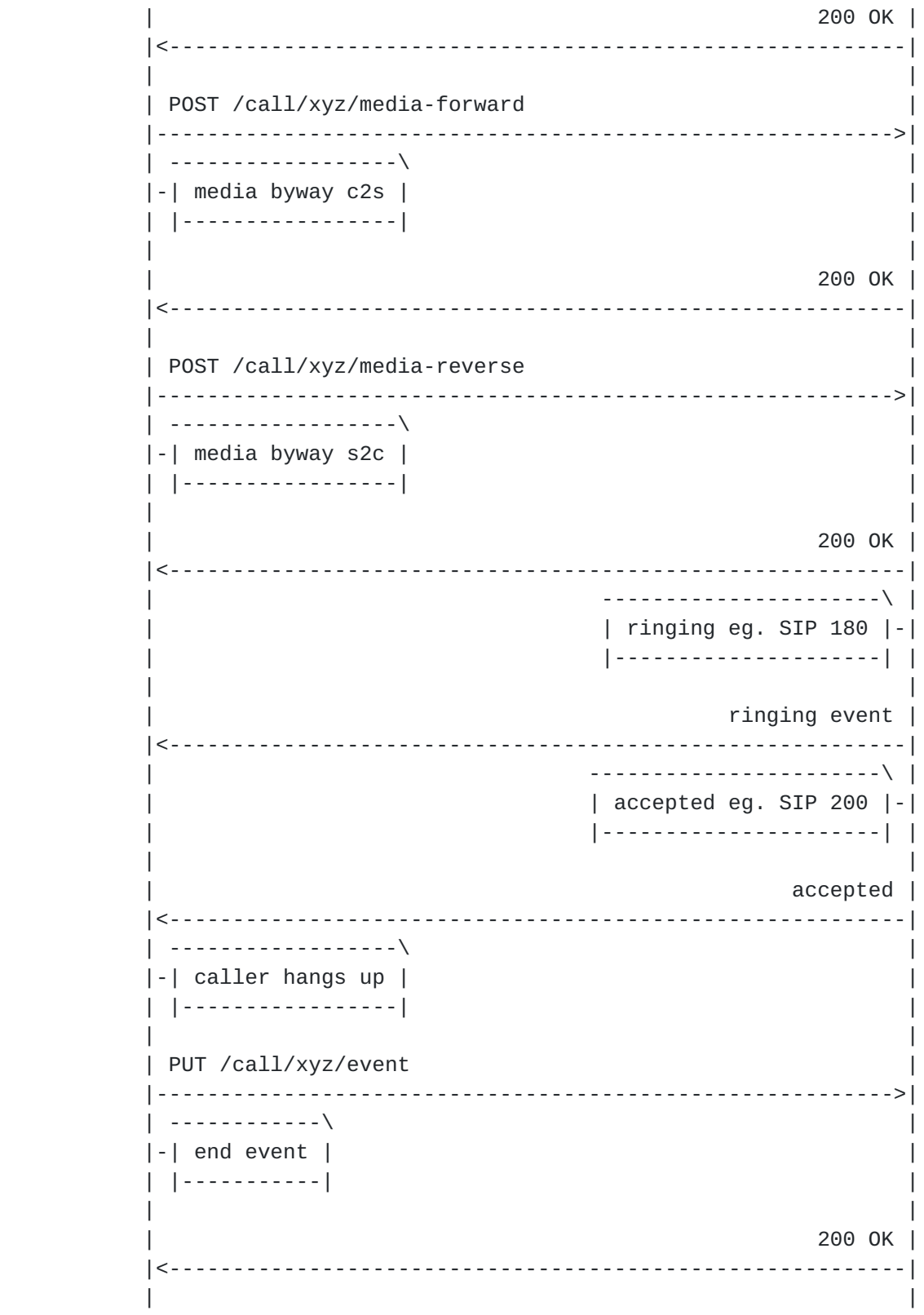
RIPP provides a simple technique for allowing a call to seamlessly migrate from one client instance to another on a different host, or from one server instance to another on a different host. For a client, it need only end the byways in use for the call and re-initiate from a different instance. Similarly, a server can request migration, and this triggers the client to perform this same action. The call state persists independently of the state of the HTTP connection or the byways embedded in HTTP transactions, so that a reconnect can continue where things left off.

Finally, RIPP trunks can be destroyed by a trunking consumer by issuing a DELETE against the RIPP trunk provider URI.

7. Example

This section describes a typical example where one company, Acme, is using a cloud calling service - Webex - and gets PSTN trunking from the provider Comcast.

The sequence diagram for the outbound call flow is here:



The first stage is for Webex to set up their service to be able to work as an OAuth Resource Server, working with Comcast as the

Authorization Server, and to obtain the baseURI that Comcast uses for RIPP authorization. Assume that this is "https://ripp.comcast.com". The next stage is the admin from ACME logs on to their Webex account and selects Comcast as the RIPP provider. This will cause the OAUTH dance and the admin will end up having approved Webex to use Acme's account at Comcast for RIPP. Webex will have received an OAuth access and refresh token from Comcast and be passed the new Provider Trunk URI. At this point, provisioning is complete and calls can start. Assume the provider trunk URI returned is "https://ripp.comcast.com/trunks/123".

Webex will start by setting up for incoming calls at "https://ripp.webex/trunks/abc" with an opaque security token of "secret1234". This is done by making a HTTP PUT to https://ripp.comcast.com/trunks/123/consumerTrunk with a JSON body of:

```
{
"consumerTrunkURI":"https://ripp.webex/trunks/abc " ,
"consumerToken":"secret1234"
}
```

The Comcast server will then find out the advertised capability of the Webex trunk by doing a GET to https://ripp.webex/trunks/abc/capAdv and using the secret1234 as an authorization token. Webex supports the default values but also support G.729 as an additional codec. It returns a JSON body of:

```
{  "audio/g729": true }
```

Similarly, the Webex server will find out the advertised capability of the trunk by doing a GET to https:://ripp.comcast.com/trunks/123/capAdv, using its OAuth token. In this case, the response is empty, indicating that the capabilities are all default.

At this point we are ready for inbound or outbound calls.

7.1. Inbound Call

A PSTN calls arrives at Comcast that is routed to the this trunk via a Comcast SBC that will convert it from SIP to RIPP. The SBC knows which codecs the trunk supports (G.729, Opus and G.711) and can immediately send the SIP answer in a 183. It can then can make an HTTP post to the consumer trunk URI to set up the incoming call. This is does by doing a POST to "https://ripp.webex/trunks/acme123/calls&target=14085551212@e164.arpa" using the authorization token "secret1234". This will return a new call URI for this call of https://ripp.webex/call/xyz.

At this point the SBC can make a long poll GET and PUT to "https://ripp.webex/call/xyz/events" to receive and send signaling events for this call. The SBC will also open a number of media byways by making POST requests to "https://ripp.webex/call/xyz/media-forward" and "https://ripp.webex/call/xyz/media-reverse" to send and receive media.

For each of the media-forward byways, the Comcast SBC will send a BywayPreamble that tells the other side meta data about what will be sent on this byway. For the media-reverse byways, the Webex server will send the BywayPreamble. The BywayPreamble contains the name of the codec, the base sequence number, frameTime, and baseTime. After this BywayPreamble, media frames can be sent that contain a seqOffset number, media length, and then the media data. The receiver compute the time sequence number for the frame by adding the baseSeqNum for the byway to the seqOffset for the frame. The timestamp for the media is computed using the baseTime for the byway plus the packeTime multiplied by the seqNum.

The data from the https://ripp.webex/call/xyz/events request will be an infinite JSON array of Events. When the Webex server answers the call, the event returned would look like:

```
{ "name":"accepted" }
```

7.2. Outbound Call

For Webex to make it outbound call, it is the same as the inbound call other than the provider trunk URI is used. The Webex server would act as a client and do a HTTP POST to "https://ripp.comcast.com/trunks/123/calls&target=14085551212@e164.arpa" to create a call URI of "http\s://ripp.comcast.com/call/c789". From that point the flow is roughly the same as inbound with the client and server roles reversed.

7.3. End of call

If the call is ended on the server side, server sends a terminated event with the ended flag set to true then waits a small time for client to close the connection then closes the connection.

If the call is ended on the client side, the client sends a terminated event with the ended flag set to true and then closes the connection. In either case the even looks like:

```
{ "name":"terminated", "ended": true }
```

8. Detailed Behaviours

This section provides an overview of the operation of RIPP.

8.1. Configuration

RIPP configuration happens when a trunking consumer wishes to be able to provision, on demand, new RIPP trunks with a trunking provider.

One example use case is that of an enterprise, which has deployed an IP PBX of some sort within its data centers. Once deployed, the enterprise needs to enable the PBX to place and receive calls towards the PSTN. The enterprise contracts with a RIPP trunking provider. All of this happens as a precursor to configuration. At the end of the contracting process, the enterprise administrator will visit the configuration web page, and be able to register their enterprise PBX. This process will typically return a client-ID, client-secret, and authorization endpoint URL. The administrator manually enters these into the configuration of their PBX. [[OPEN ISSUE: OpenID connect?]]

As another example use case, a cloud contact center, cloud PBX provider, or any other saas application which wishes to obtain trunking services, can contract with a RIPP trunking provider. In a similar process to the enterprise case above, the administrator obtains a clientID, client-secret, and authorization endpoint URL which are configured into their service.

In the final use case, an enterprise administrator has purchased trunking services from a RIPP trunking provider. They separately have purchased cloud PBX, cloud contact center, or another saas service which requires connectivity to a RIPP trunk. In this case, the cloud PBX, cloud contact center, or other saas service acts as the RIPP trunk consumer. The RIPP trunk consumer would configure itself as a client with a variety of RIPP trunking providers, and for each, obtain the clientID, client-secret and authorization URL. This will allow the customers of the RIPP trunking consumer to provision RIPP trunks automatically, and point them to the RIPP trunking consumer.

8.2. RIPP Trunk Provisioning

Once a RIPP consumer has been configured as an OAuth client application with a RIPP provider, a RIPP customer can provision a RIPP trunk on-demand using a web form. RIPP consumers will typically provide a self-service web form for such provisioning, since self-service and instant provisioning are key goals of RIPP.

The RIPP customer visits this web form, and selects their provider. The RIPP consumer would then initiate an OAuth2.0 authorization code flow. This utilizes the clientID, client-secret and authorization endpoint URL configured previously. The RIPP customer will authenticate to the RIPP provider, and authorize creation of a new RIPP trunk.

Once the RIPP customer authorizes creation of a RIPP trunk, the RIPP provider MUST generate an authorization code and follow the procedures defined in [RFC6749] for the authorization code grant flow. Furthermore, the RIPP provider MUST mint a new URI identifying this new RIPP trunk. This URI MUST contain a path component, and MUST NOT contain any URI parameters. This URI MUST be an HTTPS URI, and HTTP/3 MUST be supported for this URI. The path component MUST be a globally unique identifier for this trunk, and not depend on the authority component as part of the namespace for purposes of uniqueness.

As an example, the following is a valid RIPP trunk URI:

<https://ripp.comcast.com/trunks/6ha937fjjj9>

This URI MUST be returned in the OAuth2.0 parameter "ripp-trunk", and MUST be base64 encoded.

The RIPP consumer MUST follow the procedures defined in [RFC6749] for an OAuth client, trade in its authorization code for both a refresh and access token. The RIPP provider MUST issue both refresh and access tokens. It is expected that the refresh token will last a long time, in order to avoid the resource owner needing to manually re-authorize. The trunk consumer MUST be prepared for its access and refresh tokens to be invalidated at any time. The RIPP consumer MUST extract the "ripp-trunk" OAuth parameter from the authorization response, decode, and persist it.

Once the RIPP consumer has obtained an access token, it MUST initiate an HTTPS PUT request towards /consumerTrunk on the provider trunk URI. This request MUST contain an Authorization header field utilizing the access token just obtained. It MUST include a RIPP provisioning object in the body. This object is specified in Section Section 10.

The RIPP provisioning object MUST contain a RIPP trunk consumer URI and a RIPP bearer token. The RIPP consumer MUST mint an HTTPS URI for the RIPP Trunk consumer URI. This URI MUST support HTTP/3, and MUST implement the behaviours associated with capabilities and new call operations as defined below. This URI MUST have a path

component, MUST NOT contain any URI parameters, and MUST have a path segment which is globally unique.

In addition, the RIPP consumer MUST mint a bearer token to be used by the RIPP provider when performing operations against the RIPP Trunk Client URI. The bearer token MAY be constructed in any way desired by the RIPP consumer. The token and URI SHOULD remain valid for at least one day, however, a security problem could cuase them to be invalidated. The RIPP consumer MUST refresh the provisioning against the RIPP trunk at least one hour in advance of the expiration, in order to ensure no calls are delayed.

At this point, the RIPP trunk is provisioned. Both the RIPP provider and RIPP consumer have a RIPP trunk URI and an Authorization token to be used for placing calls in each direction.

8.3. Capabilities

Once provisioned, each side obtains receive capabilities about the trunk from its peer. To do that, each client performs an HTTP GET to /capAdv on its peer's RIPP trunk URI. The response body MUST be a RIPP capabilities object as defined in Section Section 10.

Once established, either side MAY update the capabilities by sending an HTTP push to trigger its peer to fetch a fresh capability document. Due to race conditions, it is possible that the client may receive calls compliant to the old capabilities document for a brief interval. It MUST be prepared for this.

When the trunk resource is destroyed, its associated capabilities are also destroyed.

The RIPP capabilities document is a list of name-value pairs, which specify a capability. Every capability has a default, so that if no document is posted, or it is posted but a specific capability is not included, the capability for the peer is understood. Capabilities are receive only, and specify what the entity is willing to receive. Capabilities are bound to the RIPP trunk, and are destroyed when the RIPP trunk is destroyed.

This specification defines the following capability set. This set is extensible through an IANA registry.

- max-bitrate: The maximum bitrate for receiving voice. This is specified in bits per second. It MUST be greater than or equal to

  1. Its default is 64000.

- max-samplerate: The maximum sample rate for audio. This is specified in Hz. It MUST be greater than or equal to 8000. Its default is 8000.

- max-samplesize: The maximum sample size for audio. This is specified in bits. It MUST be greater than or equal to 8. The default is 16.

- force-cbr: Indicates whether the entity requires CBR media only. It MUST be either "true" or "false". The default is "false". If "true", the sender MUST send constant rate audio.

- two-channel: Indicates whether the entity supports receiving two audio channels or not. Two channel audio is specifically used for RIPP trunks meant to convey listen-only media for the purposes of recording, similar to SIPREC [RFC7866]. It MUST be either "true" or "false". The default is "false".

- tnt: Indicates whether the entity supports the takeback-and-transfer command. Telcos supporting this feature on a trunk would set it to "true". The value MUST be "true" or "false". The default is "false".

In addition, codecs can be listed as capabilities. This is done by using the media type and subtype, separated by a "/", as the capability name. Media type and subtype values are taken from the IANA registry for RTP payload format media types, as defined in [RFC4855]. The value of the capability is "true" if the codec is supported, "false" if it is not. The default is "false" for all codecs except for "audio/PCMU", "audio/opus", "audio/telephone-event" and "audio/CN", for which the default is "true". Because codec capabilities are receive-only, it is possible, and totally acceptable, for there to be different audio codecs used in each direction.

In general, an entity MUST declare a capability for any characteristic of a call which may result in the call being rejected. This requirement facilitates prevention of call failures, along with clear indications of why calls have failed when they do. For example, if a RIPP trunk provider provisions a trunk without support for G.729, but the consumer configures their to utilize this codec, this will be known as a misconfiguration immediately. This enables validation of trunk configurations in an automated fashion, without placing test calls or calling customer support.

8.4. Initiating Calls

HTTP connections are completely independent of RIPP trunks or calls. As such, RIPP clients SHOULD reuse existing HTTP connections for any request targeted at the same authority to which an existing HTTP connection is open. RIPP clients SHOULD also utilize 0-RTT HTTP procedures in order to speed up call setup times.

To initiate a new call, a RIPP client creates an HTTPS POST request to /calls endpoint on the RIPP trunk URI of its peer. For a trunking consumer, this is the RIPP trunk URI provisioned during the OAuth2.0 flow. For the trunking provider, it is the RIPP trunk consumer URI learned through the provisioning POST operation. This MUST be an HTTP/3 transaction. The client MUST validate that the TLS certificate that is returned matches the authority component of the RIPP trunk URI.

This request MUST contain the token that the client has obtained out-of-band. For the RIPP trunk consumer, this is the OAuth token. For the RIPP trunk provider, it is the bearer token learned through the provisioning POST operation.

The client MUST also add the "target" URI parameter. This parameter MUST be of the form user@domain. If the target is a phone number on the PSTN, this must take the form @e164.arpa, where is a valid E.164 number. RIPP also supports private trunks, in which case the it MUST take the form @, where the number is a non-E164 number scoped to be valid within the domain. This form MUST NOT be used for E.164 numbers. Finally, RIPP can be used to place call to application services - such as a recorder - in which case the parameter would take the form of an RFC822 email address.

The client MUST add an HTTP Identity header field. This header field is defined in Section Section 11 as a new HTTP header field. Its contents MUST be a valid Identity header field as defined by [RFC8224]. This ensures that all calls utilize secure caller ID. A RIPP client MUST NOT place the caller ID in any place except for the Identity header field in this request. Specifically, a "From", "Contact", or "P-Asserted-ID" header field MUST NOT ever appear.

- CJ - I would prefer to add this another way without using a header.

The server MUST validate the OAuth token, MUST act as the verifying party to verify the Identity header field, and then authorize the creation of a new call, and then either accept or reject the request. If accepted, it indicates that the server is willing to create this call. The server MUST return a 201 Created response, and MUST

include a Location header field containing an HTTPS URI which identifies the call that has been created. The URI identifying the call MUST include a path segment which contains a type 4 UUID, ensuring that call identifiers are globally unique.

The server MAY include HTTP session cookies in the 201 response. The client MUST support receipt of cookies [RFC6265]. It MUST be prepared to receive up to 10 cookies per call. The client MUST destroy all cookies associated with a call, when the call has ended. Cookies MUST NOT be larger the 5K.

The usage of an HTTP URI to identify the call itself, combined with session cookies, gives the terminating RIPP domain a great deal of flexibility in how it manages state for the call. In traditional softswitch designs, call and media state is held in-memory in the server and not placed into databases. In such a design, a RIPP server can use the session cookie in combination with sticky session routing in the load balancers to ensure that subsequent requests for the same call go to the same call server. Alternatively, if the server is not using any kind of HTTP load balancer at all, it can use a specific hostname in the URI to route all requests for this call to a specific instance of the server. This technique is particularly useful for telcos who have not deployed HTTP infrastructure, but do have SBCs that sit behind a single virtual IP address. The root URI can use a domain whose A record maps to this IP. Once a call has landed on a particular SBC, the call URI can indicate the specific IP of the SBC.

For example, the RIPP trunk URI for such a telco operator might be:

<https://sbc-farm.comcast.com/trunks/6ha937fjjj9>

which always resolves to 1.2.3.4, the VIP shared amongst the SBC farm. Consequently, a request to this RIPP trunk would hit a specific SBC behind the VIP. This SBC would then create the call and return a call URL which points to its actual IP, using DNS

<https://sbc23.sbc-farm.comcast.com/call/ha8d7f6fso29s88clzopa>

However, the HTTP URI for the call MUST NOT contain an IP address; it MUST utilize a valid host or domain name. This is to ensure that TLS certificate validation functions properly without manual configuration of certificates (a practice which is required still for SIP based peering).

Neither the request, nor the response, contain bodies.

8.5. Establishing the Signaling Byways

To perform signalling for this call, the client MUST initiate an HTTP GET and PUT request towards the call URI that it just obtained, targeted at the /event endpoint.

The signaling is accomplished by a long running HTTP transaction, with a stream of JSON in the PUT request, and a stream of JSON in the GET response.

The body begins with an open curly bracket, and after that is a series of JSON objects, each starting with a curly bracket, and ending with a curly bracket. Consequently, each side MUST immediately send their respective open brackets after the HTTP header fields. We utilize streaming JSON in order to facilitate usage of tools like CURL for signalling operations.

8.6. The Media Sequence

In RIPP, media is represented as a continuous sequence of RIPP media frames embedded in a media byway. Each ripp media frame encodes a variable length sequence number offset, followed by a variable length length field, followed by a codec frame equal to that length. The media byway itself, when created, includes properties that are shared across all media frames within that byway. These parameters include the sequence number base, the timestamp base, the codec type, and the frame size in milliseconds for the codec.

This is a significantly different design than RTP, which conveys many repeated parameters (such as the payload type and timestamp) in every packet. Instead, RIPP extracts information that will be shared across many packets and associates it with the byway itself. This means the media frames only contain the information which varies - the sequence number and length. [[OPEN ISSUE: we could maybe even eliminate the sequence number by computing it from offset in the stream. Worried about sync problems though?]]

Consequently, each media frame has the following properties:

- The sequence number, which is equal to the sequence number base associated with the media byway, PLUS the value of the sequence number offset

- The timestamp, which is equal to the timestamp base from the byway, PLUS the sequence number offset TIMES the frame size in milliseconds. Note that this requires that frame size must remain fixed for all media frames in a byway.

- The codec type, which is a fixed property of the byway. There are no payload type numbers in RIPP.

RIPP does not support gaps in the media sequence due to silence. Something must be transmitted for each time interval. If a RIPP implementation wishes to change codecs, it MUST utilize a different byway for that codec.

8.7. Opening Media Byways

The client bears the responsibility for opening media byways - both forward and reverse. Consequently, the server is strongly dependent on the client opening reverse byways; it cannot send media unless they've been opened.

A client MUST open a new forward byway whenever it has a media frame to send, all existing forward byways (if any) are in the blocked state, and the client has not yet opened 20 byways.

Furthermore, the client MUST keep a minimum of 10 reverse byways open at all times. This ensures the server can send media. The client MUST open these byways immediately, in parallel.

The use of multiple media byways in either direction is essential to low latency operation of RIPP. This is because, as describe below, media frames are sprayed across these byways in order to ensure that there is never head-of-line blocking. This is possible because, in HTTP/3, each transaction is carried over a separate QUIC stream, and QUIC streams run on top of UDP. Furthermore, a QUIC stream does not require a handshake to be established - creation of new QUIC streams is a 0-RTT process.

The requests to create these transactions MUST include Cookie headers for any applicable session cookies.

To open a forward media byway, the client MUST initiate a POST request to the /media-forward endpoint on the call URI, and MUST include a RIPP-Media header field in the request headers. Similarly, to open a reverse media byway, the client MUST initiate a POST request to the /media-reverse endpoint of the call URI. It MUST NOT include a RIPP-Media header field in the request headers. The server MUST include the RIPP-Media header in the response headers. The RIPP-Media header contains the properties for the byway - the sequence number base, the timestamp base, and the name of the codec.

RIPP supports multiple audio channels, meant for SIPREC use cases. Each channel MUST be on a separate byway. When multi-channel audio

is being used, the client MUST include the multi-channel parameter and MUST include the channel number, starting at 1.

All RIPP implementations MUST support G.711 and Opus audio codecs. All implementations MUST support [RFC2833] for DTMF, and MUST support [RFC3389] for comfort noise, for both sending and receiving.

The sequence number space is unique for each direction, channel, and call (as identified by the call URI). Each side MUST start the sequence number at zero, and MUST increment it by one for each subsequent media frame. The sequence number base is represented as a string corresponding to a 32 bit unsigned integer, and the sequence number offset in the media frame is variable length, representing an unsigned integer. Consequently, the sequence number space for a media stream within a call has a total space of 32 bits. With a minimum frame size of 10ms, RIPP can support call durations as long as 11,930 hours. Rollover of the sequence number is not permitted, the client or server MUST end the call before rollover. This means that the combination of call URI, direction (client to server, or server to client), channel number, and sequence number represent a unique identifier for media packets.

8.8. Sending and Receiving Media

The approach for media is media striping.

To avoid HOL blocking, we cannot send a second media packet on a byway until we are sure the prior media packet was received. This is why the client opens multiple media byways.

When either the client or server sends a media frame on a byway, it immediately marks the byway as blocked. At that point, it SHOULD NOT send another media frame on that byway. The client or server notes the sequence number and channel number for that media frame. Once it receives an acknowledgement for that corresponding media frame, it marks the byway as UNBLOCKED. A client or server MAY send a media frame on any unblocked byway.

The sequence number for the media frame is computed based on the rules described above.

Per the logic described above, the client will open additional byways once the number of blocked byways goes above a threshold. If a the number of blocked byways in either direction hits 75% of the total for that direction, this is a signal that congestion has occurred. In such a case, the client or server MUST either drop packets at the application layer, or buffer them for later transmission. [[TODO:

can we play with QUIC priorities to prioritize newer media frames over older?]]

When a client or server receives a media frame, it MUST send an acknowledge message. This message MUST be sent on the same byway on which the media was received. This acknowledgement message MUST contain the full sequence number and channel number for the media packet that was received. It MUST also contain the timestamp, represented as wallclock time, at which the media packet was received.

If the server has marked 75% of the reverse media byways as blocked, it MUST send a signaling event instructing the client to open another reverse media byway. Once this command is received, the client MUST open a new reverse byway, unless the total number of byways has reached 20.

A client MAY terminate media byways gracefully if they have not sent or received packets on that byway for 5 or more seconds. This is to clean up unused byways.

There is no need for sender or receiver reports. The equivalent information is knowable from the application layer acknowledgements.

8.9. Terminating and Re-establishing Connections and Byways

The state of the connection, the QUIC streams, and byways, is separate from the state of the call. The client MAY terminate an HTTP connection or byway at any time, and re-establish it. Similarly, the server or client may end the a byway at any time.

If a byway ends or the connection breaks or is migrated, the client MUST re-initiate the byways immediately, or risk loss of media and signalling events. However, to deal with the fact that re-establishment takes time, both client and server MUST buffer their signalling and media streams for at least 5 seconds, and then once the connections and byways are re-established, it sends all buffered data immediately.

Note that it is the sole responsibility of the client to make sure byways are re-established if they fail unexpectedly.

8.10. Signaling - Events

Signaling is performed by having the client and server exchange events. Each event is a JSON object embedded in the signalling stream, which conveys the event as perceived by the client or server. Each event has a sequence number, which starts at zero for a call,

and increases by one for each event. The sequence number space is unique in each direction. The event also contains a direction field, which indicates whether the event was sent from client to server, or server to client. It also contains a timestamp field, which indicates the time of the event as perceived by the sender. This timestamp is not updated when retransmissions happen; the timestamp exists at the RIPP application layer and RIPP cannot directly observe HTTP retransmits.

It also contains a call field, which contains the URI of the call in question.

Finally, there is an event type field, which conveys the type of event. This is followed by additional fields which are specific to the event type.

This structure means that each event carried in the signalling is totally self-describing, irregardless of the enclosing connection and stream. This greatly facilitates logging, debugging, retransmissions, retries, and other race conditions which may deliver the same event multiple times, or deliver an event to a server which is not aware of the call.

Events are also defined so that the resulting state is uniquely defined by the event itself. This ensures that knowing the most recent event is sufficient to determine the state of the call.

This specification defines the following events:

alerting: Passed from server to client, indicating that the recipient is alerting.

accepted: Passed from server to client, indicating that the call was accepted.

rejected: Passed from server to client, indicating that the call was rejected by the user.

failed: Passed from server to client, indicating that the call was rejected by server or downstream servers, not by the user, but due to some kind of error condition. This event contains a response code and reason phrase, which are identical to the response codes and reason phrases in SIP.

noanswer: Passed from server to client, indicating that the call was delivered to the receiving user but was not answered, and the server or a downstream server timed out the call.

end: initiated by either client or server, it indicates that the call is to be terminated. Note that this does NOT delete the HTTP resource, it merely changes its state to call end. Furthermore, a call cannot be ended with a DELETE against the call URI; DELETE is not permitted and MUST be rejected by the server. The call end event SHOULD contain a reason, using the Reason codes defined for SIP.

- CJ - Not keen on SIP reason codes - they did not conatin enough info for all the Q950 stuff and were not particually extensible. I think it would be better to define a set here with clear mapping to SIP and SIP +Q950 reasons.

migrate: sent from server to client, it instructs the client to terminate the connections and re-establish them to a new URI which replaces the URI for the call. The event contains the new URI to use. This new URI MUST utilize the same path components, and MUST have a different authority component.

open-reverse: sent from server to client, it instructs the client to open an additional set of reverse media byways.

- CJ - would it work to have this all far simplier and just have the trunk cap advertisemtn say how many to open up ?

tnt: send from consumer to provider, it invokes a takeback-and-transfer operation. It includes the phone number to which the call should be transferred. The provide will then transfer the call to the target number. This event is meant to invoke the feature as it has been implemented by the provider. RIPP does not define additional behaviors.

8.11. Call Termination

Signaling allows an application layer call end to be sent. This will also cause each side to terminate the the outstanding transactions using end flags per HTTP/3 specs. However, the opposite is not true - ending of the transactions or connection does not impact the call state.

A server MUST maintain a timer, with a value equal to one second, for which it will hold the call in its current state without any active signalling byway. If the server does not receive a signalling byway before the expiration of this timer, it MUST consider the call as ended. Once the call has ended, the call resource SHOULD be destroyed.

If the server receives a signalling or media byway for a call that is TERMINATED, it MUST reject the transaction with an 404 response code, since the resource no longer exists.

8.12. GET Transactions

A client MAY initiate a GET request against the call URI at any time. This returns the current state of the resource. This request returns the most recent event, either sent by the server or received by the server.

- CJ - lets call this previosEvent as the event part waits till the next event on a long poll. Be good to say something about how this is used as it is not clear to me it is needed.

8.13. Graceful Call Migration: Server

To facilitate operational maintenance, the protocol has built in support for allowing a server instance to drain all active calls to another server instance.

The server can issue a migrate event over the signalling byway, which includes a new call URI that the peer should use. Once received, the client closes all transactions to the current call URI. It then establishes new signalling, media and media control byways to the URI it just received. All media that the client wishes to transmit, but was unable to do so during the migration, is buffered and then sent in a burst once the media byways are re-established. This ensures there is no packet loss (though there will be jitter) during the migration period.

We don't use QUIC layer connection migration, as that is triggered by network changes and not likely to be exposed to applications.

8.14. Graceful Call Migration: Client

Clients can move a call from one client instance to another easily. No commands are required. The client simply ends the in-progress transactions for signalling and media, and then reinitiates them to the existing call URI from whatever server is to take over. Note that the client MUST do this within 1s or the server will end the call.

8.15. Ungraceful Call Migration

Since all media packets are acknowledged at the application layer, it is possible for endpoints to very quickly detect remote failures, network failures, and other related problems.

Failure detection falls entirely at the hands of the client. A failure situation is detected when any one of the following happens:

1. The QUIC connection closes unexpectedly

2. Any outstanding signalling or media byway is reset by the peer

3. No media packets are received from the peer for 1s

4. No acknowledgements are received for packets that have been sent in the last 1s

If the client detects such a failure, it MUST abort all ongoing transactions to the server, terminate the QUIC connection, and then establish a new connection using 0-RTT, and re-establish signalling and media transactions. If this retry fails, the client MUST consider the call terminated. It SHOULD NOT a further attempt to re-establish the call.

- CJ - Note there is no way to know if it can use 0-RTT or not, all depends on cached state so the best it can do is hope it might work.

9. SIP Gateway

RIPP is designed to be easy to gateway from SIP. The expectation is that RIPP will be implemented in SBCs and softswitches. A SIP to RIPP gateway has to be call-stateful, acting as a B2BUA, in order to gateway to RIPP. Furthermore, a SIP to RIPP gateway has to act as a media termination point in SIP. It has to perform any SRTP decryption and encryption, and it must de-packetize RTP packets to extract their timestamps, sequence numbers, and codec types.

SIP to RIPP gateways are not transparent. SIP header fields which are unknown or do not map to RIPP functionality as described here, MUST be discarded.

Any configuration and provisioning for RIPP happens ahead of receipt or transmission of SIP calls. Consequently, the logic described here applies at the point that a gateway receives a SIP INVITE on the SIP side, or receives a POST to the RIPP trunk URI on the RIPP side.

This specification does define some normative procedures for the gateway function in order to maximize interoperability.

9.1. RIPP to SIP

9.2. SIP to RIPP

10. RAML API

```
#%RAML 1.0
---
title: RIPP
baseUri: http://ripp.example.net/{version}
version: v1
protocols: [ HTTPS ]
securedBy: [ oauth_2_0 ]
securitySchemes:
  oauth_2_0: !include securitySchemes/oauth_2_0.raml

types:
    InboundEndpoint:
        type: object
        properties:
            consumerTrunkURI: string
            consumerToken: string
    Event:
        type: object
        properties:
            name:
                enum: [ alerting, accepted, rejected, failed, tnt, migrate,
end,open-reverse ]
            direction:
                enum: [c2s, s2c]
            sequence number:
                type: number
            timestamp:
                type: number
            ended:
                type: boolean
            timeStamp:
                type: datetime
            tntDestination:
                type: string
                note: only in events with name tnt
            migrateToURl:
                type: string
                note: only in events with name migrate
     Advertisement:
        type object
        properties:
            max-bitrate: number
```

```
            max-samplerate: number
```

```
            max-channels: number
            non-e164: boolean
            force-cbr: boolean
            tnt: boolean
    Frame:
        seqNumOffset: number
        dataLen: number
        data: string
    FrameAck:
        seqNum: number
    BywayPreamble:
        baseSeqNum: number
        baseTime: number
        frameTime: number
        codec:
            enum: [ opus, g711, dtmf, cn, ack ]
     BywayMedia:
        mediaFrames: array


/trunks:
   /{trunkID}:
      /consumerTrunk:
        put:
            description: Set the URI and security token for consumer trunk URI
            securedBy: [oauth_2_0]
      /capAdv:
        get:
            description: Get the Capability Advertisement for this trunk
            securedBy: [oauth_2_0]
            responses:
                200:
                    body:
                         application/json:
                            type: Advertisement
      /calls:
        post:
            queryParameters:
              target:
            securedBy: [oauth_2_0]
            description: Create a new call. Returns a Call URI
            responses:
                202:



/call:
    /{callID}:
```

```
        /prevEvent:
            get:
                description: Retreive the previous event from server
                responses:
                    200:
                        body:
                            application/json:
                                type: Event
        /event:
            get:
                description: Wait for next event then retreive the most recent
event from server
                responses:
                    200:
                        body:
                            application/json:
                                type: Event
            put:
                description: Tell server about recent event
                body:
                    application/json:
                        type: Event
                responses:
                    200:
        /media-forward:
            post:
                 description: Starts an infinite flow of media frames from
client to server
                 body:
                    application/octet-stream:
                        type: BywayFlow
                 responses:
                    200:
                        application/octet-stream:
                            type: BywayFlow
        /media-reverse:
            post:
                 description: Starts an infinite flow of media frames from
server to client
                 body:
                    application/octet-stream:
                        type: BywayFlow
                 responses:
                    200:
                        application/octet-stream:
                            type: BywayFlow
```

11. IANA Considerations

12. Security Considerations

13. Acknowledgements

Thank you to Jason Livingood for the detailed review.

14. Informative References

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", draft-ietf-quic-http-20 (work in progress), April 2019.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-20 (work in progress), April 2019.

[RFC2833] Schulzrinne, H. and S. Petrack, "RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals", RFC 2833, DOI 10.17487/RFC2833, May 2000, <https://www.rfc-editor.org/info/rfc2833>.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <https://www.rfc-editor.org/info/rfc3261>.

[RFC3264] Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, DOI 10.17487/RFC3264, June 2002, <https://www.rfc-editor.org/info/rfc3264>.

[RFC3389] Zopf, R., "Real-time Transport Protocol (RTP) Payload for Comfort Noise (CN)", RFC 3389, DOI 10.17487/RFC3389, September 2002, <https://www.rfc-editor.org/info/rfc3389>.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, DOI 10.17487/RFC3550, July 2003, <https://www.rfc-editor.org/info/rfc3550>.

[RFC4855] Casner, S., "Media Type Registration of RTP Payload Formats", RFC 4855, DOI 10.17487/RFC4855, February 2007, <https://www.rfc-editor.org/info/rfc4855>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7866] Portman, L., Lum, H., Ed., Eckel, C., Johnston, A., and A. Hutton, "Session Recording Protocol", RFC 7866, DOI 10.17487/RFC7866, May 2016, <https://www.rfc-editor.org/info/rfc7866>.

[RFC8224] Peterson, J., Jennings, C., Rescorla, E., and C. Wendt, "Authenticated Identity Management in the Session Initiation Protocol (SIP)", RFC 8224, DOI 10.17487/RFC8224, February 2018, <https://www.rfc-editor.org/info/rfc8224>.

Authors' Addresses

Jonathan Rosenberg
Five9

Email: jdrosen@jdrosen.net

Cullen Jennings
Cisco Systems

Email: fluffy@iii.ca

Anthony Minessale
Signalwire/Freeswitch

Email: anthm@signalwire.com