

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2020

M. Cavage
Oracle
M. Sporny
Digital Bazaar
October 20, 2019

Signing HTTP Messages
draft-cavage-http-signatures-12

Abstract

When communicating over the Internet using the HTTP protocol, it can be desirable for a server or client to authenticate the sender of a particular message. It can also be desirable to ensure that the message was not tampered with during transit. This document describes a way for servers and clients to simultaneously add authentication and message integrity to HTTP messages by using a digital signature.

Feedback

This specification is a joint work product of the W3C Digital Verification Community Group [1] and the W3C Credentials Community Group [2]. Feedback related to this specification should be logged in the issue tracker [3] or be sent to public-credentials@w3.org [4].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 22, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction	3
1.1. Using Signatures in HTTP Requests	4
1.2. Using Signatures in HTTP Responses	4
2. The Components of a Signature	4
2.1. Signature Parameters	5
2.1.1. keyId	5
2.1.2. signature	5
2.1.3. algorithm	5
2.1.4. created	6
2.1.5. expires	6
2.1.6. headers	6
2.2. Ambiguous Parameters	6
2.3. Signature String Construction	7
2.4. Creating a Signature	9
2.5. Verifying a Signature	9
3. The 'Signature' HTTP Authentication Scheme	10
3.1. Authorization Header	10
3.1.1. Initiating Signature Authorization	11
3.1.2. RSA Example	11
3.1.3. HMAC Example	12
4. The 'Signature' HTTP Header	12
4.1. Signature Header	12
4.1.1. RSA Example	13
4.1.2. HMAC Example	14
5. References	14
5.1. Normative References	14
5.2. Informative References	14
5.3. URIs	15
Appendix A. Security Considerations	16
Appendix B. Extensions	16
Appendix C. Test Values	17
C.1. Default Test	18
C.2. Basic Test	18
C.3. All Headers Test	19

Appendix D. Acknowledgements	19
Appendix E. IANA Considerations	20
E.1. Signature Authentication Scheme	20
E.2. HTTP Signatures Algorithms Registry	20
Authors' Addresses	21

1. Introduction

This protocol extension is intended to provide a simple and standard way for clients to sign HTTP messages.

HTTP Authentication [RFC2617] defines Basic and Digest authentication mechanisms, TLS 1.2 [RFC5246] defines cryptographically strong transport layer security, and OAuth 2.0 [RFC6749] provides a fully-specified alternative for authorization of web service requests. Each of these approaches are employed on the Internet today with varying degrees of protection. However, none of these schemes are designed to cryptographically sign the HTTP messages themselves, which is required in order to ensure end-to-end message integrity. An added benefit of signing the HTTP message for the purposes of end-to-end message integrity is that the client can be authenticated using the same mechanism without the need for multiple round-trips.

Several web service providers have invented their own schemes for signing HTTP messages, but to date, none have been standardized. While there are no techniques in this proposal that are novel beyond the previous art, it is useful to standardize a simple and cryptographically strong mechanism for digitally signing HTTP messages.

This specification presents two mechanisms with distinct purposes:

1. The "Signature" scheme which is intended primarily to allow a sender to assert the contents of the message sent are correct and have not been altered during transmission or storage in a way that alters the meaning expressed in the original message as signed. Any party reading the message (the verifier) may independently confirm the validity of the message signature. This scheme is agnostic to the client/server direction and can be used to verify the contents of either HTTP requests, HTTP responses, or both.
2. The "Authorization" scheme which is intended primarily to allow a sender to request access to a resource or resources by proving that they control a secret key. This specification allows for this both with a shared secret (using HMAC) or with public/private keys. The "Authorization" scheme is typically used in authentication processes and not directly for message signing.

As a consequence 'Authorization' header is normally generated (and the message signed) by the HTTP client and the message verified by the HTTP server.

1.1. Using Signatures in HTTP Requests

It is common practice to protect sensitive website and API functionality via authentication mechanisms. Often, the entity accessing these APIs is a piece of automated software outside of an interactive human session. While there are mechanisms like OAuth and API secrets that are used to grant API access, each have their weaknesses such as unnecessary complexity for particular use cases or the use of shared secrets which may not be acceptable to an implementer. Shared secrets also prohibit any possibility for non-repudiation, while secure transports such as TLS do not provide for this at all.

Digital signatures are widely used to provide authentication and integrity assurances without the need for shared secrets. They also do not require a round-trip in order to authenticate the client, and allow the integrity of a message to be verified independently of the transport (e.g. TLS). A server need only have an understanding of the key (e.g. through a mapping between the key being used to sign the content and the authorized entity) to verify that a message was signed by that entity.

When optionally combined with asymmetric keys associated with an identity, this specification can also enable authentication of a client and server with or without prior knowledge of each other.

1.2. Using Signatures in HTTP Responses

HTTP messages are routinely altered as they traverse the infrastructure of the Internet, for mostly benign reasons. Gateways and proxies add, remove and alter headers for operational reasons, so a sender cannot rely on the recipient receiving exactly the message transmitted. By allowing a sender to sign specified headers, and recipient or intermediate system can confirm that the original intent of the sender is preserved, and including a Digest header can also verify the message body is not modified. This allows any recipient to easily confirm both the sender's identity, and any incidental or malicious changes that alter the content or meaning of the message.

2. The Components of a Signature

There are a number of components in a signature that are common between the 'Signature' HTTP Authentication Scheme and the

'Signature' HTTP Header. This section details the components of the digital signature parameters common to both schemes.

2.1. Signature Parameters

The following section details the Signature Parameters.

2.1.1. keyId

REQUIRED. The 'keyId' field is an opaque string that the server can use to look up the component they need to validate the signature. It could be an SSH key fingerprint, a URL to machine-readable key data, an LDAP DN, etc. Management of keys and assignment of 'keyId' is out of scope for this document. Implementations MUST be able to discover metadata about the key from the 'keyId' such that they can determine the type of digital signature algorithm to employ when creating or verifying signatures.

2.1.2. signature

REQUIRED. The 'signature' parameter is a base 64 encoded digital signature, as described in RFC 4648 [RFC4648], Section 4 [5]. The client uses the 'algorithm' and 'headers' Signature Parameters to form a canonicalized 'signing string'. This 'signing string' is then signed using the key associated with the 'keyId' according to its digital signature algorithm. The 'signature' parameter is then set to the base 64 encoding of the signature.

2.1.3. algorithm

RECOMMENDED. The 'algorithm' parameter is used to specify the signature string construction mechanism. Valid values for this parameter can be found in the HTTP Signatures Algorithms Registry [6] and MUST NOT be marked "deprecated". Implementers SHOULD derive the digital signature algorithm used by an implementation from the key metadata identified by the 'keyId' rather than from this field. If 'algorithm' is provided and differs from the key metadata identified by the 'keyId', for example 'rsa-sha256' but an EdDSA key is identified via 'keyId', then an implementation MUST produce an error. Implementers should note that previous versions of the 'algorithm' parameter did not use the key information to derive the digital signature type and thus could be utilized by attackers to expose security vulnerabilities.

2.1.4. created

RECOMMENDED. The 'created' field expresses when the signature was created. The value MUST be a Unix timestamp integer value. A signature with a 'created' timestamp value that is in the future MUST NOT be processed. Using a Unix timestamp simplifies processing and avoids timezone management required by specifications such as RFC3339. Subsecond precision is not supported. This value is useful when clients are not capable of controlling the 'Date' HTTP Header such as when operating in certain web browser environments.

2.1.5. expires

OPTIONAL. The 'expires' field expresses when the signature ceases to be valid. The value MUST be a Unix timestamp integer value. A signature with an 'expires' timestamp value that is in the past MUST NOT be processed. Using a Unix timestamp simplifies processing and avoid timezone management existing in RFC3339. Subsecond precision is allowed using decimal notation.

2.1.6. headers

OPTIONAL. The 'headers' parameter is used to specify the list of HTTP headers included when generating the signature for the message. If specified, it SHOULD be a lowercased, quoted list of HTTP header fields, separated by a single space character. If not specified, implementations MUST operate as if the field were specified with a single value, '(created)', in the list of HTTP headers. Note:

1. The list order is important, and MUST be specified in the order the HTTP header field-value pairs are concatenated together during Signature String Construction (Section 2.3) used during signing and verifying.
2. A zero-length 'headers' parameter value MUST NOT be used, since it results in a signature of an empty string.

2.2. Ambiguous Parameters

If any of the parameters listed above are erroneously duplicated in the associated header field, then the the signature MUST NOT be processed. Any parameter that is not recognized as a parameter, or is not well-formed, MUST be ignored.

2.3. Signature String Construction

A signed HTTP message needs to be tolerant of some trivial alterations during transmission as it goes through gateways, proxies, and other entities. These changes are often of little consequence and very benign, but also often not visible to or detectable by either the sender or the recipient. Simply signing the entire message that was transmitted by the sender is therefore not feasible: Even very minor changes would result in a signature which cannot be verified.

This specification allows the sender to select which headers are meaningful by including their names in the `'headers'` Signature Parameter. The headers appearing in this parameter are then used to construct the intermediate Signature String, which is the data that is actually signed.

In order to generate the string that is signed with a key, the client MUST use the values of each HTTP header field in the `'headers'` Signature Parameter, in the order they appear in the `'headers'` Signature Parameter. It is out of scope for this document to dictate what header fields an application will want to enforce, but implementers SHOULD at minimum include the `'(request-target)'` and `'(created)'` header fields if `'algorithm'` does not start with `'rsa'`, `'hmac'`, or `'ecdsa'`. Otherwise, `'(request-target)'` and `'date'` SHOULD be included in the signature.

To include the HTTP request target in the signature calculation, use the special `'(request-target)'` header field name. To include the signature creation time, use the special `'(created)'` header field name. To include the signature expiration time, use the special `'(expires)'` header field name.

1. If the header field name is `'(request-target)'` then generate the header field value by concatenating the lowercased `:method`, an ASCII space, and the `:path` pseudo-headers (as specified in HTTP/2, Section 8.1.2.3 [7]). Note: For the avoidance of doubt, lowercasing only applies to the `:method` pseudo-header and not to the `:path` pseudo-header.
2. If the header field name is `'(created)'` and the `'algorithm'` parameter starts with `'rsa'`, `'hmac'`, or `'ecdsa'` an implementation MUST produce an error. If the `'created'` Signature Parameter is not specified, or is not an integer, an implementation MUST produce an error. Otherwise, the header field value is the integer expressed by the `'created'` signature parameter.

3. If the header field name is ``(expires)`` and the ``algorithm`` parameter starts with ``rsa``, ``hmac``, or ``ecdsa`` an implementation MUST produce an error. If the ``expires`` Signature Parameter is not specified, or is not an integer, an implementation MUST produce an error. Otherwise, the header field value is the integer expressed by the ``created`` signature parameter.
4. Create the header field string by concatenating the lowercased header field name followed with an ASCII colon ``:``, an ASCII space `` ``, and the header field value. Leading and trailing optional whitespace (OWS) in the header field value MUST be omitted (as specified in RFC7230 [RFC7230], Section 3.2.4 [8]).
 1. If there are multiple instances of the same header field, all header field values associated with the header field MUST be concatenated, separated by a ASCII comma and an ASCII space ``,``, and used in the order in which they will appear in the transmitted HTTP message.
 2. If the header value (after removing leading and trailing whitespace) is a zero-length string, the signature string line correlating with that header will simply be the (lowercased) header name, an ASCII colon ``:``, and an ASCII space `` ``.
 3. Any other modification to the header field value MUST NOT be made.
 4. If a header specified in the headers parameter is malformed or cannot be matched with a provided header in the message, the implementation MUST produce an error.
5. If value is not the last value then append an ASCII newline ``\n``.

To illustrate the rules specified above, assume a ``headers`` parameter list with the value of ``(request-target) (created) host date cache-control x-emptyheader x-example`` with the following HTTP request headers:

```
GET /foo HTTP/1.1
Host: example.org
Date: Tue, 07 Jun 2014 20:51:35 GMT
X-Example: Example header
           with some whitespace.
X-EmptyHeader:
Cache-Control: max-age=60
Cache-Control: must-revalidate
```

For the HTTP request headers above, the corresponding signature string is:

```
(request-target): get /foo
(created): 1402170695
host: example.org
date: Tue, 07 Jun 2014 20:51:35 GMT
cache-control: max-age=60, must-revalidate
x-emptyheader:
x-example: Example header with some whitespace.
```

2.4. Creating a Signature

In order to create a signature, a client MUST:

1. Use the 'headers' and 'algorithm' values as well as the contents of the HTTP message, to create the signature string.
2. Use the key associated with 'keyId' to generate a digital signature on the signature string.
3. The 'signature' is then generated by base 64 encoding the output of the digital signature algorithm.

For example, assume that the 'algorithm' value is "hs2019" and the 'keyId' refers to an EdDSA public key. This would signal to the application that the signature string construction mechanism is the one defined in Section 2.3: Signature String Construction [9], the signature string hashing function is SHA-512, and the signing algorithm is Ed25519 as defined in RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519. The result of the signature creation algorithm should result in a binary string, which is then base 64 encoded and placed into the 'signature' value.

2.5. Verifying a Signature

In order to verify a signature, a server MUST:

1. Use the received HTTP message, the 'headers' value, and the Signature String Construction (Section 2.3) algorithm to recreate the signature.
2. The 'algorithm', 'keyId', and base 64 decoded 'signature' listed in the Signature Parameters are then used to verify the authenticity of the digital signature. Note: The application verifying the signature MUST derive the digital signature algorithm from the metadata associated with the 'keyId' and MUST NOT use the value of 'algorithm' from the signed message.

If a header specified in the 'headers' value of the Signature Parameters (or the default item '(created)' where the 'headers' value is not supplied) is absent from the message, the implementation MUST produce an error.

For example, assume that the 'algorithm' value was "hs2019" and the 'keyId' refers to an EdDSA public key. This would signal to the application that the signature string construction mechanism is the one defined in Section 2.3: Signature String Construction [10], the signature string hashing function is SHA-512, and the signing algorithm is Ed25519 as defined in RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519. The result of the signature verification algorithm should result in a successful verification unless the headers protected by the signature were tampered with in transit.

3. The 'Signature' HTTP Authentication Scheme

The "Signature" authentication scheme is based on the model that the client must authenticate itself with a digital signature produced by either a private asymmetric key (e.g., RSA) or a shared symmetric key (e.g., HMAC).

The scheme is parameterized enough such that it is not bound to any particular key type or signing algorithm.

3.1. Authorization Header

The client is expected to send an Authorization header (as defined in RFC 7235 [RFC7235], Section 4.1 [11]) where the "auth-scheme" is "Signature" and the "auth-param" parameters meet the requirements listed in Section 2: The Components of a Signature.

The rest of this section uses the following HTTP request as an example.

```
POST /foo HTTP/1.1
Host: example.org
Date: Tue, 07 Jun 2014 20:51:35 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18
```

```
{"hello": "world"}
```

Note that the use of the 'Digest' header field is per RFC 3230 [RFC3230], Section 4.3.2 [12] and is included merely as a demonstration of how an implementer could include information about

the body of the message in the signature. The following sections also assume that the "rsa-key-1" keyId asserted by the client is an identifier meaningful to the server.

3.1.1. Initiating Signature Authorization

A server may notify a client when a resource is protected by requiring a signature. To initiate this process, the server will request that the client authenticate itself via a 401 response [13] code. The server may optionally specify which HTTP headers it expects to be signed by specifying the 'headers' parameter in the WWW-Authenticate header. For example:

```
HTTP/1.1 401 Unauthorized
Date: Thu, 08 Jun 2014 18:32:30 GMT
Content-Length: 1234
Content-Type: text/html
WWW-Authenticate: Signature
    realm="Example",headers="(request-target) (created)"

...
```

3.1.2. RSA Example

The authorization header and signature would be generated as:

```
Authorization: Signature keyId="rsa-key-1",algorithm="hs2019",
    headers="(request-target) (created) host digest content-length",
    signature="Base64(RSA-SHA512(signing string))"
```

The client would compose the signing string as:

```
(request-target): post /foo\n
(created): 1402174295
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

Note that the '\n' symbols above are included to demonstrate where the new line character should be inserted. There is no new line on the final line of the signing string. Each HTTP header above is displayed on a new line to provide better readability of the example.

For an RSA-based signature, the authorization header and signature would then be generated as:

```
Authorization: Signature keyId="rsa-key-1",algorithm="hs2019",
headers="(request-target) (created) host digest content-length",
signature="Base64 (RSA-SHA512(signing string)) "
```

3.1.3. HMAC Example

For an HMAC-based signature without a list of headers specified, the authorization header and signature would be generated as:

```
Authorization: Signature keyId="hmac-key-1",algorithm="hs2019",
headers="(request-target) (created) host digest content-length",
signature="Base64 (HMAC-SHA512(signing string)) "
```

The only difference between the RSA Example and the HMAC Example is the digital signature algorithm that is used. The client would compose the signing string in the same way as the RSA Example above:

```
(request-target): post /foo\n
(created): 1402174295
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

4. The 'Signature' HTTP Header

The "Signature" HTTP Header provides a mechanism to link the headers of a message (client request or server response) to a digital signature. By including the "Digest" header with a properly formatted digest, the message body can also be linked to the signature. The signature is generated and verified either using a shared secret (e.g. HMAC) or public/private keys (e.g. RSA, EC). This allows the receiver and/or any intermediate system to immediately or later verify the integrity of the message. When the signature is generated with a private key it can also provide a measure of non-repudiation, though a full implementation of a non-repudiable statement is beyond the scope of this specification and highly dependent on implementation.

The "Signature" scheme can also be used for authentication similar to the purpose of the 'Signature' HTTP Authentication Scheme (Section 3). The scheme is parameterized enough such that it is not bound to any particular key type or signing algorithm.

4.1. Signature Header

The sender is expected to transmit a header (as defined in RFC 7230 [RFC7230], Section 3.2 [14]) where the "field-name" is "Signature", and the "field-value" contains one or more "auth-param"s (as defined

in RFC 7235 [RFC7235], Section 4.1 [15]) where the "auth-param" parameters meet the requirements listed in Section 2: The Components of a Signature.

The rest of this section uses the following HTTP request as an example.

```
POST /foo HTTP/1.1
Host: example.org
Date: Tue, 07 Jun 2014 20:51:35 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18

{"hello": "world"}
```

The following sections assume that the "rsa-key-1" keyId provided by the signer is an identifier meaningful to the server.

4.1.1. RSA Example

The signature header and signature would be generated as:

```
Signature: keyId="rsa-key-1",algorithm="hs2019",
  created=1402170695, expires=1402170995,
  headers="(request-target) (created) (expires)
  host date digest content-length",
  signature="Base64(RSA-SHA256(signing string))"
```

The client would compose the signing string as:

```
(request-target): post /foo\n
(created): 1402170695
(expires): 1402170995
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

Note that the '\n' symbols above are included to demonstrate where the new line character should be inserted. There is no new line on the final line of the signing string. Each HTTP header above is displayed on a new line to provide better readability of the example.

For an RSA-based signature, the authorization header and signature would then be generated as:

```
Signature: keyId="rsa-key-1",algorithm="hs2019",created=1402170695,  
  headers="(request-target) (created) host digest content-length",  
  signature="Base64 (RSA-SHA512(signing string))"
```

4.1.2. HMAC Example

For an HMAC-based signature without a list of headers specified, the authorization header and signature would be generated as:

```
Signature: keyId="hmac-key-1",algorithm="hs2019",created=1402170695,  
  headers="(request-target) (created) host digest content-length",  
  signature="Base64 (HMAC-SHA512(signing string))"
```

The only difference between the RSA Example and the HMAC Example is the signature algorithm that is used. The client would compose the signing string in the same way as the RSA Example above:

```
(request-target): post /foo\n  
(created): 1402170695  
host: example.org\n  
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n  
content-length: 18
```

5. References

5.1. Normative References

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <<https://www.rfc-editor.org/info/rfc7235>>.

5.2. Informative References

- [RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, DOI 10.17487/RFC2617, June 1999, <<https://www.rfc-editor.org/info/rfc2617>>.

- [RFC3230] Mogul, J. and A. Van Hoff, "Instance Digests in HTTP", RFC 3230, DOI 10.17487/RFC3230, January 2002, <<https://www.rfc-editor.org/info/rfc3230>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/info/rfc6234>>.
- [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>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.

5.3. URIs

- [1] <https://w3c-dvcg.github.io/>
- [2] <https://w3c-ccg.github.io/>
- [3] <https://github.com/w3c-dvcg/http-signatures/issues>
- [4] <mailto:public-credentials@w3.org>
- [5] <https://tools.ietf.org/html/rfc4648#section-4>
- [6] [#hsa-registry](#)
- [7] <https://tools.ietf.org/html/rfc7540#section-8.1.2.3>
- [8] <https://tools.ietf.org/html/rfc7230#section-3.2.4>
- [9] [#canonicalization](#)

- [10] #canonicalization
- [11] <https://tools.ietf.org/html/rfc7235#section-2.1>
- [12] <https://tools.ietf.org/html/rfc3230#section-4.3.2>
- [13] <https://tools.ietf.org/html/rfc7235#section-3.1>
- [14] <https://tools.ietf.org/html/rfc7230#section-3.2>
- [15] <https://tools.ietf.org/html/rfc7235#section-4.1>
- [16] <https://web-payments.org/specs/source/http-signatures-audit/>
- [17] <https://web-payments.org/specs/source/http-signature-nonces/>
- [18] <https://web-payments.org/specs/source/http-signature-trailers/>
- [19] <https://www.iana.org/assignments/http-auth-scheme-signature>
- [20] <https://www.iana.org/assignments/http-authschemes>
- [21] <https://www.iana.org/assignments/shm-algorithms>
- [22] #canonicalization
- [23] #canonicalization
- [24] #canonicalization
- [25] #canonicalization
- [26] #canonicalization

Appendix A. Security Considerations

There are a number of security considerations to take into account when implementing or utilizing this specification. A thorough security analysis of this protocol, including its strengths and weaknesses, can be found in Security Considerations for HTTP Signatures [16].

Appendix B. Extensions

This specification was designed to be simple, modular, and extensible. There are a number of other specifications that build on this one. For example, the HTTP Signature Nonces [17] specification details how to use HTTP Signatures over a non-secured channel like

HTTP and the HTTP Signature Trailers [18] specification explains how to apply HTTP Signatures to streaming content. Developers that desire more functionality than this specification provides are urged to ensure that an extension specification doesn't already exist before implementing a proprietary extension.

If extensions to this specification are made by adding new Signature Parameters, those extension parameters MUST be registered in the Signature Authentication Scheme Registry. The registry will be created and maintained at (the suggested URI) <https://www.iana.org/assignments/http-auth-scheme-signature> [19]. An example entry in this registry is included below:

Signature Parameter: nonce
Reference to specification: [HTTP_AUTH_SIGNATURE_NONCE], Section XYZ.
Notes (optional): The HTTP Signature Nonces specification details how to use HTTP Signatures over a unsecured channel like HTTP.

Appendix C. Test Values

WARNING: THESE TEST VECTORS ARE OLD AND POSSIBLY WRONG. THE NEXT VERSION OF THIS SPECIFICATION WILL CONTAIN THE PROPER TEST VECTORS.

The following test data uses the following RSA 2048-bit keys, which we will refer to as 'keyId=Test' in the following samples:

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDCFENGw33yGiHy92pDjZQh1OC3
6rPJj+CvfSC8+q28hxA161QFNucl3wuCTUcq0Qd2qsBe/2hFyc2DCJJg0h1L78+6
Z4UMR7EOcpfdUE9Hf3m/hs+FUR45uBJeDK1HSFHD8bHKD6kv8FPGfJTotc+2xjJw
oYi+lhqplfIekaxsyQIDAQAB
-----END PUBLIC KEY-----
```

```
-----BEGIN RSA PRIVATE KEY-----
MIICXgIBAAKBgQDCFENGw33yGiHy92pDjZQh1OC36rPJj+CvfSC8+q28hxA161QF
NUcl3wuCTUcq0Qd2qsBe/2hFyc2DCJJg0h1L78+6Z4UMR7EOcpfdUE9Hf3m/hs+F
UR45uBJeDK1HSFHD8bHKD6kv8FPGfJTotc+2xjJwoYi+lhqplfIekaxsyQIDAQAB
AoGBAJR8ZkCUvx5kzv+utdl7T5MnordT1TvoXXJGXX7ZZ+UuvMNUCdn2QPc4sBiA
QWvLwlclSKt5DsKZ8UETpYPy8pPYnnDEz2dDYiaew9+xEpubyeW2oH4Zx71wqBtOK
kqwrXa/pzdpiucRRjk6vE6YY7EBBs/g7uanVpGibOVAESqH1AkeA7DkjVH28WDUg
flnqvfn2Kj6CT7nIcE3jGJsZZ7z1ZmBmHFDONMLUrXR/Zm3pR5m0tCmBqa5RK95u
412jtlldPIwJBANJT3v8pnkth48bQo/fKel6uEYyboRtA5/uHuHkZ6FQF7OUkGogc
mSJluOdc5t6hI1VsLn0QZEjQZMEOWr+wKSMCQQCC4kXJESHAve77oP6HtG/IiEn7
kpyUXRNVfsDE0czpJJBvL/aRFUJxuRK91jhjC68sA7NsKMG5OXb5I5Jj36xAkEA
gIT7aFOYBFwGgQAQkWNKLvySgKbAZRTElBacpHMuQdl1DfdntvAyqpAZ01Y0RKmW
G6aFKaqQfOXKCyWoUiVknQJAXrlgySFci/2ueKlIE1QqIiLSZ8V80lpFLRnb1pzI
7UlyQXnTAEFYM560yJlZUpOb1V4cScGd365tiSMvxLOvTA==
-----END RSA PRIVATE KEY-----
```

All examples use this request:

```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Sun, 05 Jan 2014 21:31:40 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18

{"hello": "world"}
```

C.1. Default Test

If a list of headers is not included, the date is the only header that is signed by default for rsa-sha256. The string to sign would be:

```
date: Sun, 05 Jan 2014 21:31:40 GMT
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
signature="SjWJWbWN7i0wzBvtPl8rbASWz5xQW6mcJmn+ibttBqtifLN7Sazz
6m79cNfwwb8DMJ5couls7uEGKKCs+FLEEdV5lp7q25WqS+lavg7T8hc0GppauB
6hbgEKTwbldHYGETbGmtdHgVCk9SuS13F0hZ8FD0k/50xEPXe5WozsbM="
```

The Signature header would be:

```
Signature: keyId="Test",algorithm="rsa-sha256",
signature="SjWJWbWN7i0wzBvtPl8rbASWz5xQW6mcJmn+ibttBqtifLN7Sazz
6m79cNfwwb8DMJ5couls7uEGKKCs+FLEEdV5lp7q25WqS+lavg7T8hc0GppauB
6hbgEKTwbldHYGETbGmtdHgVCk9SuS13F0hZ8FD0k/50xEPXe5WozsbM="
```

C.2. Basic Test

The minimum recommended data to sign is the (request-target), host, and date. In this case, the string to sign would be:

```
(request-target): post /foo?param=value&pet=dog
host: example.com
date: Sun, 05 Jan 2014 21:31:40 GMT
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
  headers="(request-target) host date",
  signature="qdx+H7PHHDZgy4y/Ahn9Tny9V3GP6YgBPYUXMmoxWtLbHpUnXS
2mg2+SbrQDMCJypxBLSPQR2aAjn7ndmw2iicw3HMbe8VfEdKFYRqzic+efkb3
nndiv/xlxSHDJWeSWkx3ButlYSuBskLu6kd9Fswtemr3lgdDEmn04swr2Os0="
```

C.3. All Headers Test

A strong signature including all of the headers and a digest of the body of the HTTP request would result in the following signing string:

```
(request-target): post /foo?param=value&pet=dog
host: example.com
date: Sun, 05 Jan 2014 21:31:40 GMT
content-type: application/json
digest: SHA-256=X48E9qOokqgrvdtS8nOJRJN3OWDUoyWxBf7kbu9DBPE=
content-length: 18
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
  created=1402170695, expires=1402170699,
  headers="(request-target) (created) (expires)
  host date content-type digest content-length",
  signature="vSdrb+dS3EceC9bcwHSo4MlyKS59iF1rhgYkz8+oVLEEzmYZZvRs
8rgOp+63LEM3v+MFHB32NfpB2bEKBIvB1q52LaEUHFv120V01IL+TAD48XaERZF
ukWgHoBTLmHYS2Gb51gWxpeIq8knRmPnYePbF5MokR0Zkly4zKH7s1dE="
```

The Signature header would be:

```
Signature: keyId="Test",algorithm="rsa-sha256",
  created=1402170695, expires=1402170699,
  headers="(request-target) (created) (expires)
  host date content-type digest content-length",
  signature="vSdrb+dS3EceC9bcwHSo4MlyKS59iF1rhgYkz8+oVLEEzmYZZvRs
8rgOp+63LEM3v+MFHB32NfpB2bEKBIvB1q52LaEUHFv120V01IL+TAD48XaERZF
ukWgHoBTLmHYS2Gb51gWxpeIq8knRmPnYePbF5MokR0Zkly4zKH7s1dE="
```

Appendix D. Acknowledgements

The editor would like to thank the following individuals for feedback on and implementations of the specification (in alphabetical order): Mark Adamcin, Mark Allen, Paul Annesley, Karl Boehlmark, Stephane Bortzmeyer, Sarven Capadisli, Liam Dennehy, ductm54, Stephen Farrell, Phillip Hallam-Baker, Eric Holmes, Andrey Kislyuk, Adam Knight, Dave Lehn, Dave Longley, James H. Manger, Ilari Liusvaara, Mark Nottingham, Yoav Nir, Adrian Palmer, Lucas Pardue, Roberto Polli,

Julian Reschke, Michael Richardson, Wojciech Ryski, Adam Scarr,
Cory J. Slep, Dirk Stein, Henry Story, Lukasz Szewc, Chris Webber,
and Jeffrey Yasskin

Appendix E. IANA Considerations

E.1. Signature Authentication Scheme

The following entry should be added to the Authentication Scheme Registry located at <https://www.iana.org/assignments/http-authschemes> [20]

Authentication Scheme Name: Signature
Reference: [RFC_THIS_DOCUMENT], Section 2.
Notes (optional): The Signature scheme is designed for clients to authenticate themselves with a server.

E.2. HTTP Signatures Algorithms Registry

The following initial entries should be added to the Canonicalization Algorithms Registry to be created and maintained at (the suggested URI) <https://www.iana.org/assignments/shm-algorithms> [21]:

Editor's note: The references in this section are problematic as many of the specifications that they refer to are too implementation specific, rather than just pointing to the proper signature and hashing specifications. A better approach might be just specifying the signature and hashing function specifications, leaving implementers to connect the dots (which are not that hard to connect).

Algorithm Name: hs2019
Status: active
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3:
Signature String Construction [22]
Hash Algorithm: RFC 6234 [RFC6234], SHA-512 (SHA-2 with 512-bits of digest output)
Digital Signature Algorithm: Derived from metadata associated with 'keyId'. Recommend support for RFC 8017 [RFC8017], Section 8.1: RSASSA-PSS, RFC 6234 [RFC6234], Section 7.1: SHA-Based HMACs, ANSI X9.62-2005 ECDSA, P-256, and RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519.

Algorithm Name: rsa-sha1
Status: deprecated, SHA-1 not secure.
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3:
Signature String Construction [23]

Hash Algorithm: RFC 6234 [RFC6234], SHA-1 (SHA-1 with 160-bits of digest output)
Digital Signature Algorithm: RFC 8017 [RFC8017], Section 8.2: RSASSA-PKCS1-v1_5

Algorithm Name: rsa-sha256
Status: deprecated, specifying signature algorithm enables attack vector.
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3:
Signature String Construction [24]
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)
Digital Signature Algorithm: RFC 8017 [RFC8017], Section 8.2: RSASSA-PKCS1-v1_5

Algorithm Name: hmac-sha256
Status: deprecated, specifying signature algorithm enables attack vector.
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3:
Signature String Construction [25]
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)
Message Authentication Code Algorithm: RFC 6234 [RFC6234],
Section 7.1: SHA-Based HMACs

Algorithm Name: ecdsa-sha256
Status: deprecated, specifying signature algorithm enables attack vector.
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3:
Signature String Construction [26]
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)
Digital Signature Algorithm: ANSI X9.62-2005 ECDSA, P-256

Authors' Addresses

Mark Cavage
Oracle
500 Oracle Parkway
Redwood Shores, CA 94065
US

Phone: +1 415 400 0626
Email: mcavage@gmail.com
URI: <https://www.oracle.com/>

Manu Sporny
Digital Bazaar
203 Roanoke Street W.
Blacksburg, VA 24060
US

Phone: +1 540 961 4469
Email: msporny@digitalbazaar.com
URI: <https://manu.sporny.org/>

ASAP
Internet-Draft
Expires: July 31, 2021

K. Inamdar
S. Narayanan
C. Jennings
Cisco Systems
January 27, 2021

Automatic Peering for SIP Trunks
draft-kinamdar-dispatch-sip-auto-peer-05

Abstract

This draft specifies a configuration workflow to enable enterprise Session Initiation Protocol (SIP) networks to solicit the capability set of a SIP service provider network. The capability set can subsequently be used to configure features and services on the enterprise edge element, such as a Session Border Controller (SBC), to ensure smooth peering between enterprise and service provider networks.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 31, 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Conventions and Terminology	3
3. Reference Architecture	3
4. Configuration Workflow	5
5. Overview of Operations	6
6. HTTP Transport	8
6.1. HTTP Methods	8
6.2. Integrity and Confidentiality	8
6.3. Authenticated Client Identity	8
6.4. Encoding the Request	8
6.5. Generating the response	9
6.6. Identifying the Request Target	9
7. State Deltas	9
8. Encoding the Service Provider Capability Set	10
9. Data Model for Capability Set	10
9.1. Tree Diagram	10
9.2. YANG Model	12
9.3. Node Definitions	17
9.4. Extending the Capability Set	24
10. Example Capability Set Document Encoding	25
10.1. XML Capability Set Document	26
11. Example Exchange	27
12. Security Considerations	28
13. References	28
13.1. Normative References	28
13.2. Informative References	28
14. Acknowledgments	29
15.1. URIs	29
Authors' Addresses	31

1. Introduction

The deployment of a Session Initiation Protocol [RFC 3261 [1]] (SIP)-based infrastructure in enterprise and service provider communication networks is increasing at a rapid pace. Consequently, direct IP peering between enterprise and service provider networks is quickly replacing traditional methods of interconnection between enterprise and service provider networks. Currently published standards provide a strong foundation over which direct IP peering can be realized. However, given the sheer number of these standards, it is often not clear which behavioral subsets, extensions to baseline protocols and operating principles ought to be implemented by service provider and enterprise networks to ensure successful peering.

The SIP Connect technical recommendations [2] aim to solve this problem by providing a master reference that promotes seamless peering between enterprise and service provider SIP networks. However, despite the extensive set of implementation rules and operating guidelines, interoperability issues between service provider and enterprise networks persist. This is in large part because service providers and equipment manufacturers aren't required to enforce the guidelines of the technical specifications and have a fair degree of freedom to deviate from them. Consequently, enterprise administrators usually undertake a fairly rigorous regimen of testing, analysis and troubleshooting to arrive at a configuration block that ensures seamless service provider peering. However, this workflow complements the SIP Connect technical recommendations, in that both endeavours aim to promote/achieve interop between the enterprise and service provider.

Another set of interoperability problems arise when enterprise administrators are required to translate a set of technical recommendations from service providers to configuration blocks across one or more devices in the enterprise, which is usually an error prone exercise. Additionally, such technical recommendations might not be nuanced enough to intuitively allow the generation of specific configuration blocks.

This draft introduces a mechanism using which an enterprise network can solicit a detailed capability set from a SIP service provider; the detailed capability set can subsequently be used by automaton or an administrator to generate configuration blocks across one or more devices within the enterprise to ensure successful service provider peering.

2. Conventions and Terminology

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [3]

3. Reference Architecture

Figure 1 illustrates a reference architecture that may be deployed to support the mechanism described in this document. The enterprise network consists of a SIP-PBX, media endpoints and a Session Border Controller [RFC 7092 [4]]. It may also include additional components such as application servers for voicemail, recording, fax etc. At a high level, the service provider consists of a SIP signaling entity (SP-SSE), a media entity and a HTTPS [RFC 7231 [5]] server.

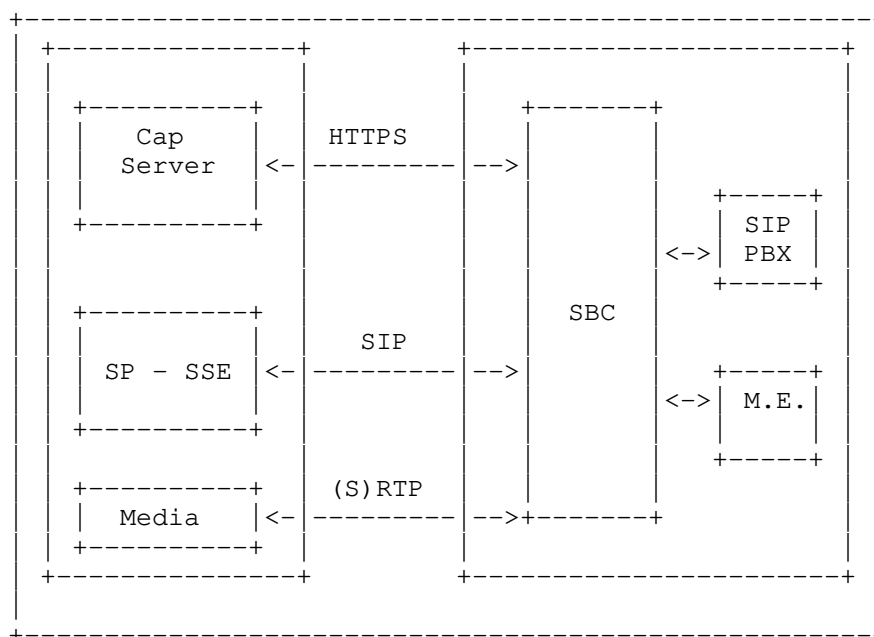


Figure 1: Reference Architecture

This draft makes use of the following terminology:

- o **Enterprise Network:** A communications network infrastructure deployed by an enterprise which interconnects with the service provider network over SIP. The enterprise network could include devices such as application servers, endpoints, call agents and edge devices, among others.
- o **Edge Device:** A device that is the last hop in the enterprise network and that is the transit point for traffic entering and leaving the enterprise. An edge device is typically a back-to-back user agent (B2BUA) [RFC 7092 [6]] such as a Session Border Controller (SBC).
- o **Service Provider Network:** A communications network infrastructure deployed by service providers. In the context of this draft, the service provider network is accessible over SIP for the establishment, modification and termination of calls and accessible over HTTPS for the transfer of the capability set document. The service provider network is also referred to as a SIP Service Provider (SSP) or Internet Telephony Service Provider (ITSP) network.

- o Call Control: Call Control within a telephony networks refers to software that is responsible for delivering its core functionality. Call control not only provides the basic functionality of setting up, sustaining and terminating calls, but also provides the necessary control and logic required for additional services within the telephony network.
- o Capability Server: A server hosted in the service provider network, such that this server is the target for capability set document requests from the enterprise network.
- o Capability Set: This specification uses the term capability set (or capability set document) to refer collectively to a set of characteristics within the service provider network, which when communicated to the enterprise network, provides the enterprise network the information required to interconnect with the service provider network. The various parameters that constitute the capability set relate to characteristics that are specific to signalling, media, transport and security. Certain aspects of interconnecting with service providers are out of scope of the capability set. For example, the access technology used to interconnect with service provider networks.

4. Configuration Workflow

A workflow that facilitates an enterprise network to solicit the capability set of a SIP service provider ought to take into account the following considerations:

- o The configuration workflow must be based on a protocol or a set of protocols commonly used between enterprise and service provider telephony networks.
- o The configuration workflow must be flexible enough to allow the service provider network to dynamically offload different capability sets to different enterprise networks based on the identity of the enterprise network.
- o Capability set documents obtained as a result of the configuration workflow must be conducive to easy parsing by automaton. Subsequently, automaton may be used for generation of appropriate configuration blocks.

Taking the above considerations into account, this document proposes a Hypertext Transfer Protocol (HTTP)-based workflow using which the enterprise network can solicit and ultimately obtain the service provider capability set. The enterprise network creates a well

formed HTTPS GET request to solicit the service provider capability set. Subsequently, the HTTPS response from the SIP service provider includes the capability set. The capability set is encoded in either XML or JSON, thus ensuring that the response can be easily parsed by automaton.

There are alternative mechanisms using which the SIP service provider can offload its capability set. For example, the Session Initiation Protocol (SIP) can be extended to define a new event package [RFC 6665 [7]], such that the enterprise network can establish a SIP subscription with the service provider for its capability set; the SIP service provider can subsequently use the SIP NOTIFY request to communicate its capability set or any state deltas to its baseline capability set.

This mechanism is likely to result in a barrier to adoption for SIP service providers and enterprise networks as equipment manufacturers would have to first add support for such a SIP extension. A HTTPS-based approach would be relatively easier to adopt as most edge devices deployed in enterprise networks today already support HTTPS; from the perspective of service provider networks, all that is required is for them to deploy HTTPS servers that function as capability servers. Additionally, most SIP service providers require enterprise networks to register with them (using a SIP REGISTER message) before any other SIP methods that initiate subscriptions (SIP SUBSCRIBE) or calls (SIP INVITE) are processed. As a result, a SIP-based framework to obtain a capability set would require operational changes on the part of service provider networks.

Yet another example of an alternative mechanism would be for service providers and enterprise equipment manufacturers to agree on YANG models [RFC 6020 [8]] that enable configuration to be pushed over NETCONF [RFC 6241 [9]] to enterprise networks from a centralised source hosted in service provider networks. The presence of proprietary software logic for call and media handling in enterprise devices would preclude the generation of a "one-size-fits-all" YANG model. Additionally, service provider networks pushing configuration to enterprises devices might lead to the loss of implementation autonomy on the part of the enterprise network.

5. Overview of Operations

To solicit the capability set of a SIP service provider, the edge element in an enterprise network generates a well-formed HTTPS GET request. There are two reasons why it makes sense for the enterprise edge element to generate the HTTPS request:

1. Edge elements are devices that normalise any mismatches between the enterprise and service provider networks in the media and signaling planes. As a result, when the capability set is received from the SIP service provider network, the edge element can generate appropriate configuration blocks (possibly across multiple devices) to enable interconnection.
2. Given that edge elements are configured to "talk" to networks external to the enterprise, the complexity in terms of NAT traversal and firewall configuration would be minimal.

The HTTPS GET request is targeted at a capability server that is managed by the SIP service provider such that this server processes, and on successfully processing the request, includes the capability set document in the response. The capability set document is constructed according to the guidelines of the YANG model described in this draft. The capability set document included in a successful response is formatted in either XML or JSON. The formatting depends on the value of the "Accept" header field of the HTTPS GET request. More details about the formatting of the HTTPS request and response are provided in Section 6.

There could be situations wherein an enterprise telephony network interconnects with its SIP service provider such that traffic between the two networks traverses an intermediary SIP service provider network. This could be a result of interconnect agreements between the terminating and transit SIP service provider networks. In such situations, the capability set provided to the enterprise network by its SIP service provider must account for the characteristics of the transit SIP service provider network from a signalling and media perspective. For example, if the terminating SIP service provider network supports the G.729 codec and the transit SIP service provider network does not, G.729 must not be advertised in the capability set. As another example, if the transit SIP service provider network doesn't support a SIP extension, for instance, the SIP extension for Reliable Provisional Responses as defined in RFC 3262, the terminating SIP service provider network must not advertise support for this extension in the capability set provided to the enterprise network. How a terminating SIP service provider obtains the characteristics of the intermediary SIP service provider network is out of the scope of this document; however, one method could be for the terminating SIP service provider to obtain the characteristics of the intermediary SIP service provider by leveraging the YANG model introduced in this document.

Figure 1 provides a reference architecture in which this workflow may be implemented. The architecture depicted in Figure 1 consists of an enterprise telephony network and a SIP service provider network, such

that the enterprise network attempts to provision SIP trunking services for the first time. For the sake of simplicity, the enterprise and service provider networks are decomposed into their core constituent elements.

6. HTTP Transport

This section describes the use of HTTPS as a transport protocol for the peering workflow. This workflow is based on HTTP version 1.1, and as such is compatible with any future version of HTTP that is backward compatible with HTTP 1.1.

6.1. HTTP Methods

The workflow defined in this document leverages the HTTPS GET method and its corresponding response(s) to request for and subsequently obtain the service provider capability set document. The HTTPS POST method and its corresponding response(s) is also used for client authentication.

6.2. Integrity and Confidentiality

Peering requests and responses are defined over HTTPS. However, due to the sensitive nature of information transmitted between client and server, it is required to secure HTTP using Transport Layer Security [RFC 5246 [10]]. The enterprise edge element and capability server MUST be compliant with RFC 7235 [11]. The enterprise edge element and capability server MUST support the use of the https uri scheme as defined in RFC 7230 [12].

6.3. Authenticated Client Identity

It is only required for the SIP service provider to authenticate the client (enterprise edge element). How the SIP service provider authenticates the client is out of the scope of this document, however, methods such as HTTP Digest Authentication may be used.

6.4. Encoding the Request

The edge element in the enterprise network generates a HTTPS GET request such that the request-target is obtained using the procedure outlined in section 6.6 The MIME types for the capability set document defined in this draft are "application/peering-info+json" and "application/peering-info+xml". Accordingly, the Accept header field value MUST be restricted only to these MIME types. It is possible that the edge element supports responses formatted in both JSON and XML. In such situations, the edge element might generate a

HTTPS GET request such that the Accept header field includes both MIME types along with the corresponding "qvalue" for each MIME type.

The generated HTTPS GET request must not use the "Expect" and "Range" header fields. The requests must also not use any conditional request.

6.5. Generating the response

Capability servers include the capability set documents in the body of a successful response. Capability set documents MUST be formatted in XML or JSON. For requests that are incorrectly formatted, the capability server must generate a "400 Bad Request" response. If the client (enterprise edge element) includes any other MIME types in Accept header field other than "application/peering-info+json" or "application/peering-info+xml", the capability set must reject the request with a "406 Not Acceptable" response.

The capability server can respond to client requests with redirect responses, specifically, the server can respond with the following redirect responses:

1. 301 Moved Temporarily
2. 302 Found
3. 307 Temporary Redirect

The server SHOULD include the Location header field in such responses.

6.6. Identifying the Request Target

HTTPS GET requests from enterprise edge elements MUST carry a valid request-target. The enterprise edge element might obtain the URL of the resource hosted on the capability server in one of two ways:

1. Manual Configuration
2. Discovery [TBD]

7. State Deltas

Given that the service provider capability set is largely expected to remain static, the work needed to implement an asynchronous push mechanism to encode minor changes in the capability set document (state deltas) is not commensurate with the benefits. Rather, enterprise edge elements can poll capability servers at pre-defined

intervals to obtain the full capability set document. It is recommended that capability servers are polled every 24 hours.

8. Encoding the Service Provider Capability Set

In the context of this draft, the capability set of a service provider refers collectively to a set of characteristics which when communicated to an enterprise network, provides it with sufficient information to directly peer with the service provider network. The capability set document is not designed to encode extremely granular details of all features, services, and protocol extensions that are supported by the service provider network. For example, it is sufficient to encode that the service provider uses T.38 relay for faxing, it is not required to know the value of the "T38FaxFillBitRemoval" parameter.

The parameters within the capability set document represent a wide array of characteristics, such that these characteristics collectively disseminate sufficient information to enable direct IP peering between enterprise and service provider networks. The various parameters represented in the capability set are chosen based on existing practises and common problem sets typically seen between enterprise and service provider SIP networks.

9. Data Model for Capability Set

This section defines a YANG module for encoding the service provider capability set. Section 9.1 provides the tree diagram, which is followed by a description of the various nodes within the module defined in this draft.

9.1. Tree Diagram

This section provides a tree diagram [RFC 8340 [13]] for the "ietf-capability-set" module. The interpretation of the symbols appearing in the tree diagram is as follows:

- o Brackets "[" and "]" enclose list keys.
- o Abbreviations before data node names: "rw" means configuration (read-write), and "ro" means state data (read-only).
- o Symbols after data node names: "?" means an optional node, "!" means a presence container, and "*" denotes a list and leaf-list.
- o Parentheses enclose choice and case nodes, and case nodes are also marked with a colon (":").

- o Ellipsis ("...") stands for contents of subtrees that are not shown.

The data model for the peering capability document has the following structure:

```

+--rw peering-response
  +--rw variant          string
  +--rw transport-info
    +--rw transport?     enumeration
    +--rw registrar*     host-port
    +--rw registrarRealm? string
    +--rw callControl*   host-port
    +--rw dns*           inet:ip-address
    +--rw outboundProxy? host-port
  +--rw call-specs
    +--rw earlyMedia?    boolean
    +--rw signalingForking? boolean
    +--rw supportedMethods? string
      +--rw numRange
        +--rw numRangeType* string
        +--rw count*        int32
        +--rw value*        string
  +--rw media
    +--rw mediaTypeAudio
      +--rw mediaFormat* string
    +--rw fax
      +--rw protocol*   enumeration
    +--rw rtp
      +--rw RTPTrigger? boolean
      +--rw symmetricRTP? boolean
    +--rw rtcp
      +--rw symmetricRTCP? boolean
      +--rw RTCPfeedback? boolean
  +--rw dtmf
    +--rw payloadNumber? int8
    +--rw iteration?     boolean
  +--rw security
    +--rw signaling
      +--rw type*        string
      +--rw version*     string
    +--rw mediaSecurity
      +--rw keyManagement? string
    +--rw certLocation   string
  +--rw extensions?     string

```

9.2. YANG Model

This section defines the YANG module for the peering capability set document. It imports modules (ietf-yang-types and ietf-inet-types) from [RFC 6991 [14]].

```

module ietf-sip-auto-peering {
  namespace "urn:ietf:params:xml:ns:ietf-sip-auto-peering";
  prefix "peering";

  description
    "Data model for transmitting peering parameters from SP to Enterprise";

  revision 2019-05-06 {
    description "Initial revision of peering-response doc.";
  }

  import ietf-inet-types {
    prefix "inet";
  }

  typedef ipv4-address-port {
    type string {
      pattern '(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.){3}'
        + '([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])'
        + ':(^)([1-9]|[1-5]?[0-9]{2,4}|6[1-4][0-9]{3}|65[1-4][0-9]{2}|655[1-2][
0-9]|6553[1-5])$';
    }
    description "The ipv4-address-port type represents an IPv4 address in
dotted-quad notation followed by a port number.";
  }

  typedef ipv6-address-port {
    type string {
      pattern '((:[0-9a-fA-F]{0,4}) : ([0-9a-fA-F]{0,4}) : ) {0,5}'
        + '((( [0-9a-fA-F]{0,4}) : )? (:[0-9a-fA-F]{0,4}) ) |'
        + '(((25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9])\.){3}'
        + '(25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9]))'
        + ':(^)([1-9]|[1-5]?[0-9]{2,4}|6[1-4][0-9]{3}|65[1-4][0-9]{2}|655[1-2][
0-9]|6553[1-5])$';
      pattern
        '(([^:]+:){6}([[:^:]]+:[[:^:]]+)|(. * \. * . * ))|'
        + '((( [^:]+: ) * [^:]+ ) ? : ( ( [^:]+: ) * [^:]+ ) ? )'
        + ':(^)([1-9]|[1-5]?[0-9]{2,4}|6[1-4][0-9]{3}|65[1-4][0-9]{2}|655[1-2][
0-9]|6553[1-5])$';
    }
    description
      "The ipv6-address type represents an IPv6 address in full,
mixed, shortened, and shortened-mixed notation followed by a port numbe
r.";
  }
}

```

```

typedef ip-address-port {
    type union {
        type ipv4-address-port;
        type ipv6-address-port;
    }
    description
    "The ip-address-port type represents an IP address:port number
    and is IP version neutral.";
}

typedef domain-name-port {
    type string {
        pattern
        '((([a-zA-Z0-9_]([a-zA-Z0-9\_-]){0,61})?[a-zA-Z0-9]\.)*'
        + '([a-zA-Z0-9_]([a-zA-Z0-9\_-]){0,61})?[a-zA-Z0-9]\.?)'
        + '|\.'
        + ':\^()([1-9]|[1-5]?[0-9]{2,4}|6[1-4][0-9]{3}|65[1-4][0-9]{2}|655[1-2][
0-9]|6553[1-5])$';
        length "1..258";
    }
    description
    "The domain-name-port type represents a DNS domain name followed by a port
    number.
    The name SHOULD be fully qualified whenever possible.";
}

typedef host-port {
    type union {
        type ip-address-port;
        type domain-name-port;
    }
    description
    "The host type represents either an IP address or a DNS
    domain name followed by a port number.";
}

container peering-info {
    leaf variant {
        type string;
        mandatory true;
        description "Variant of peering-response document";
    }
}

container transport-info {
    leaf transport {
        type enumeration {
            enum "TCP";
            enum "TLS";
            enum "UDP";
            enum "TCP;TLS";
        }
    }
}

```

```
        enum "TCP;TLS;UDP";
        enum "TCP;UDP";
    }
    description "Transport Protocol(s) used in SIP communication";
}

leaf-list registrar {
    type host-port;
    max-elements 3;
    description "List of service provider registrar servers";
}

leaf registrarRealm {
    type string;
    description "Realm for REGISTER requests carrying credentials";
}

leaf-list callControl {
    type host-port;
    max-elements 3;
    description "List of service provider call control servers";
}

leaf-list dns {
    type inet:ip-address;
    max-elements 2;
    description "IP address of the DNS Server(s) hosted by the service pr
provider";
}

leaf outboundProxy {
    type host-port;
    description "SIP Outbound Proxy";
}
}

container call-specs {
    leaf earlyMedia {
        type boolean;
        description "Flag indicating whether the service provider is expected
to deliver early media.";
    }

    leaf signalingForking {
        type boolean;
        description "Flag indicating whether the service provider is capable
of forking incoming calls ";
    }
}
```

```

    leaf supportedMethods {
        type string;
        description "Leaf/Leaf List indicating the different SIP methods
        support by the service provider.";
    }

    container numRange {
        leaf numRangeType {
            type string;
            description "String indicating whether the DID number range is pass
ed
            by value or by reference"
        }

        leaf count {
            when "../numRangeType = 'range' or ../numRangeType = 'block'";
            type int32;
            description "Number of DID numbers present in the number range."
        }

        leaf-list value {
            type string;
            description "Value of the DID number range or URL being passed as
            reference."
        }
    }
}

container media {
    container mediaTypeAudio {
        leaf-list mediaFormat {
            type string;
            description "Leaf List indicating the audio media formats supported
.";
        }
    }

    container fax {
        leaf-list protocol {
            type enumeration {
                enum "pass-through";
                enum "t38";
            }
            max-elements 2;
            description "Leaf List indicating the different fax protocols
            supported by the service provider.";
        }
    }
}

```

```
o container rtp {
    leaf RTPTrigger {
        type boolean;
        description "Flag indicating whether the service provider expects t
            receive the first media packet.";
    }

    leaf symmetricRTP {
        type boolean;
        description "Flag indicating whether the service provider expects
            symmetric RTP defined in [RFC4961]";
    }
}

container rtcp {
    leaf symmetricRTCP {
        type boolean;
        description " Flag indicating whether the service provider expects
            symmetric RTP defined in [RFC4961].";
    }

    leaf RTCPfeedback {
        type boolean;
        description "Flag Indicating support for RTP profile extension for
            RTCP-based feedback, as defined in [RFC4585]";
    }
}

container dtmf {
    leaf payloadNumber {
        type int8 {
            range "96..127";
        }
        description "Leaf that indicates the payload number(s) supported by
            the service provider for DTMF relay via Named-Telephony-Events";
    }

    leaf iteration {
        type boolean;
        description "Flag identifying whether the service provider supports
            NTE DTMF relay using the procedures of [RFC2833] or [RFC4733] .";
    }
}

container security {
    container signaling {
        leaf type {
```

```

        type string {
            pattern "TLS";
        }
        description "Type of signaling security supported.";
    }

    leaf version {
        type string {
            pattern "([1-9]\.[0-9])(;[1-9]\.[0-9])?(NULL)";
        }
        description "Indicates TLS version for SIP signaling";
    }
}

container mediaSecurity {
    leaf keyManagement {
        type string {
            pattern "(SDES(;DTLS-SRTP,version=[1-9]\.[0-9](,[1-9]\.[0-9])?)?)"
| (DTLS-SRTP,version=[1-9]\.[0-9](,[1-9]\.[0-9])?)|(NULL)";
        }
        description "Leaf that identifies the key management methods
supported by the service provider for SRTP.";
    }
}

    leaf certLocation {
        type string;
        description "Location of the service provider certificate chain for S
IP over TLS.";
    }
}

    leaf extensions {
        type string;
        description "Lists the various SIP extensions supported by SP";
    }
}
}

```

9.3. Node Definitions

This sub-sections provides the definition and encoding rules of the various nodes of the YANG module defined in section 9.2

***capability-set*:** This node serves as a container for all the other nodes in the YANG module; the capability-set node is akin to the root element of an XML schema.

***variant*:** This node identifies the version number of the capability set document. This draft defines the parameters for variant 1.0;

future specifications might define a richer parameter set, in which case the variant must be changed to 2.0, 3.0 and so on. Future extensions to the capability set document MUST also ensure that the corresponding YANG module is defined.

transport-info: The transport-info node is a container that encapsulates transport characteristics of SIP sessions between enterprise and service provider networks.

transport: A leaf node that enumerates the different Transport Layer protocols supported by the SIP service provider. Valid transport layer protocols include: UDP, TCP, TLS or a combination of them (with the exception of TLS and UDP).

registrar: A leaf-list that specifies the transport address of one or more registrar servers in the service provider network. The transport address of the registrar can be provided using a combination of a valid IP address and port number, or a subdomain of the SIP service provider network, or the fully qualified domain name (FQDN) of the SIP service provider network. If the transport address of a registrar is specified using either a subdomain or a fully qualified domain name, the DNS element must be populated with one or more valid DNS server IP addresses.

callControl: A leaf-list that specifies the transport address of the call server(s) in the service provider network. The enterprise network must use an applicable transport protocol in conjunction with the call control server(s) transport address when transmitting call setup requests. The transport address of a call server(s) within the service provider network can be specified using a combination of a valid IP address and port number, or a subdomain of the SIP service provider network, or a fully qualified domain name of the SIP service provider network. If the transport address of a call control server(s) is specified using either a subdomain or a fully qualified domain name, the DNS element must be populated with one or more valid DNS server IP addresses. The transport address specified in this element can also serve as the target for non-call requests such as SIP OPTIONS.

dns: A leaf list that encodes the IP address of one or more DNS servers hosted by the SIP service provider. If the enterprise network is unaware of the IP address, port number, and transport protocol of servers within the service provider network (for example, the registrar and call control server), it must use DNS NAPTR and SRV. Alternatively, if the enterprise network has the fully qualified domain name of the SIP service provider network, it must use DNS to resolve the said FQDN to an IP address. The dns element encodes the IP address of one or more DNS servers hosted in the

service provider network. If however, either the registrar or callControl elements or both are populated with a valid IP address and port pair, the dns element must be set to the quadruple octet of 0.0.0.0

outboundProxy: A leaf list that specifies the transport address of one or more outbound proxies. The transport address can be specified by using a combination of an IP address and a port number, a subdomain of the SIP service provider network, or a fully qualified domain name and port number of the SIP service provider network. If the outbound-proxy sub-element is populated with a valid transport address, it represents the default destination for all outbound SIP requests and therefore, the registrar and callControl elements must be populated with the quadruple octet of 0.0.0.0

call-specs: A container that encapsulates information about call specifications, restrictions and additional handling criteria for SIP calls between the enterprise and service provider network.

earlyMedia: A leaf that specifies whether the service provider network is expected to deliver in-band announcements/tones before call connect. The P-Early-Media header field can be used to indicate pre-connect delivery of tones and announcements on a per-call basis. However, given that signalling and media could traverse a large number of intermediaries with varying capabilities (in terms of handling of the P-Early-Media header field) within the enterprise, such devices can be appropriately configured for media cut through if it is known before-hand that early media is expected for some or all of the outbound calls. This element is a Boolean type, where a value of 1/true signifies that the service provider is capable of early media. A value of 0/false signifies that the service provider is not expected to generate early media.

signalingForking: A leaf that specifies whether outbound call requests from the enterprise might be forked on the service provider network that MAY lead to multiple early dialogs. This information would be useful to the enterprise network in appropriately handling multiple early dialogs reliably and in enforcing local policy. This element is a Boolean type, where a value of 1/true signifies that the service provider network can potentially fork outbound call requests from the enterprise. A value of 0/false indicates that the service provider will not fork outbound call requests.

supportedMethods: A leaf node that specifies the various SIP methods supported by the SIP service provider. The list of supported methods help to appropriately configuration various devices within the enterprise network. For example, if the service provider

enumerates support for the OPTIONS method, the enterprise network could periodically send OPTIONS requests as a keep-alive mechanism.

***numRange*:** Is a container that specifies the Direct Inward Dial (DID) number range allocated to the enterprise network by the SIP service provider. The DID number range allocated by the service provider to the enterprise network might be a contiguous or a non-contiguous block. The number range allocated to an enterprise can be communicated as a value or as a reference. For large enterprise networks, the size of the DID range might run into several hundred numbers. For situations in which the enterprise is allocated a large DID number range or a non-contiguous number range it is RECOMMENDED that the SIP service provider communicate this information by reference, that is, through a URL. The enterprise network is required to de-reference this URL in order to obtain the DID number range allocated by the SIP service provider. The numRange container can be used more than once. Refer to the example provided in Section 10.1.

***numRangeType*:** A leaf node that indicates whether the DID range is communicated by value or by reference. It can have a value of 'range', 'block' or 'reference'.

***count*:** A leaf node that indicates the size of the DID number range. The number range may be contiguous or non-contiguous. This leaf node MUST NOT be included when using the 'reference' numRangeType value.

***value*:** A leaf-list that encapsulates the DID number range allocated to the enterprise. If the numRangeType value is set to 'range' or 'block', this is the list of numbers allocated to the enterprise. If the numRangeType value is set to 'reference', this is the URL of the resource containing the DID number range. To ensure ease of parsing, it is RECOMMENDED that the resource contain a number range formatted as if it were being passed as a block or range.

***media*:** A container that is used to collectively encapsulate the characteristics of UDP-based audio streams. A future extension to this draft may extend the media container to describe other media types. The media container is also used to encapsulate basic information about Real-Time Transport Protocol (RTP) and Real-Time Transport Control Protocol (RTCP) from the perspective of the service provider network.

***mediaTypeAudio*:** A container for the mediaFormat leaf-list. This container collectively encapsulates the various audio media formats supported by the SIP service provider.

***mediaFormat*:** A leaf-list encoding the various audio media formats supported by the SIP service provider. The relative ordering of different media format leaf nodes from left to right indicates preference from the perspective of the service provider. Each mediaFormat node begins with the encoding name of the media format, which is the same encoding name as used in the "RTP/AVP" and "RTP/SAVP" profiles. The encoding name is followed by required and optional parameters for the given media format as specified when the media format is registered [RFC 4855 [15]]. Given that the parameters of media formats can vary from one communication session to another, for example, across two separate communication sessions, the packetization time (ptime) used for the PCMU media format might vary from 10 to 30 ms, the parameters included in the format element must be the ones that are expected to be invariant from the perspective of the service provider. Providing information about supported media formats and their respective parameters, allows enterprise networks to configure the media plane characteristics of various devices such as endpoints and middleboxes. The encoding name, one or more required parameters, one or more optional parameters are all separated by a semicolon. The formatting of a given media format parameter, must follow the formatting rules as specified for that media format.

***fax*:** A container that encapsulates the fax protocol(s) supported by the SIP service provider. The fax container encloses a leaf-list (named protocol) that enumerates whether the service provider supports t38 relay, protocol-based fax passthrough or both. The relative ordering of leaf nodes within the leaf lists indicates preference.

***rtp*:** A container that encapsulates generic characteristics of RTP sessions between the enterprise and service provider network. This node is a container for the "RTPTrigger" and "SymmetricRTP" leaf nodes.

***RTPTrigger*:** A leaf node indicating whether the SIP service provider network always expects the enterprise network to send the first RTP packet for an established communication session. This information is useful in scenarios such as "hairpinned" calls, in which the caller and callee are on the service provider network and because of sub-optimal media routing, an enterprise device such as an SBC is retained in the media path. Based on the encoding of this node, it is possible to configure enterprise devices such as SBCs to start streaming media (possibly filled with silence payloads) toward the address:port tuples provided by caller and callee. This node is a Boolean type. A value of 1/true indicates that the service provider expects the enterprise network to send the first RTP packet, whereas a value of 0/false indicates that the service provider network does

not require the enterprise network to send the first media packet. While the practise of preserving the enterprise network in a hairpinned call flow is fairly common, it is recommended that SIP service providers avoid this practise. In the context of a hairpinned call, the enterprise device retained in the call flow can easily eavesdrop on the conversation between the offnet parties.

symmetricRTP: A leaf node indicating whether the SIP service provider expects the enterprise network to use symmetric RTP as defined in RFC 4961 [16]. Uncovering this expectation is useful in scenarios where "latching" [RFC 7362 [17]] is implemented in the service provider network. This node is a Boolean type, a value of 1/true indicates that the service provider expects the enterprise network to use symmetric RTP, whereas a value of 0/false indicates that the enterprise network can use asymmetric RTP.

rtcp: A container that encapsulates generic characteristics of RTCP sessions between the enterprise and service provider network. This node is a container for the "RTCPFeedback" and "SymmetricRTCP" leaf nodes.

RTCPFeedback: A leaf node that indicates whether the SIP service provider supports the RTP profile extension for RTCP-based feedback RFC 4585 [18]. Media sessions spanning enterprise and service provider networks, are rarely made to flow directly between the caller and callee, rather, it is often the case that media traffic flows through network intermediaries such as SBCs. As a result, RTCP traffic from the service provider network is intercepted by these intermediaries, which in turn can either pass across RTCP traffic unmodified or modify RTCP traffic before it is forwarded to the endpoint in the enterprise network. Modification of RTCP traffic would be required, for example, if the intermediary has performed media payload transformation operations such as transcoding or transrating. In a similar vein, for the RTCP-based feedback mechanism as defined in RFC 4585 [19] to be truly effective, intermediaries must ensure that feedback messages are passed reliably and with the correct formatting to enterprise endpoints. This might require additional configuration and considerations that need to be dealt with at the time of provisioning the intermediary device. This node is a Boolean type, a value of 1/true indicates that the service provider supports the RTP profile extension for RTP-based feedback and a value of 0/false indicates that the service provider does not support the RTP profile extension for RTP-based feedback.

symmetricRTCP: A leaf node indicating whether the SIP service provider expects the enterprise network to use symmetric RTCP as defined in RFC 4961 [20]. This node is a Boolean type, a value of 1 indicates that the service provider expects symmetric RTCP reports,

whereas a value of 0 indicates that the enterprise can use asymmetric RTP.

dtmf: A container that describes the various aspects of DTMF relay via RTP Named Telephony Events. The dtmf container allows SIP service providers to specify two facets of DTMF relay via Named Telephony Events:

1. The payload type number using the payloadNumber leaf node.
2. Support for RFC 2833 [21] or RFC 4733 [22] using the iteration leaf node.

In the context of named telephony events, senders and receivers may negotiate asymmetric payload type numbers. For example, the sender might advertise payload type number 97 and the receiver might advertise payload type number 101. In such instances, it is either required for middleboxes to interwork payload type numbers or allow the endpoints to send and receive asymmetric payload numbers. The behaviour of middleboxes in this context is largely dependent on endpoint capabilities or on service provider constraints. Therefore, the payloadNumber leaf node can be used to determine middlebox configuration before-hand.

RFC 4733 [23] iterates over RFC 2833 [24] by introducing certain changes in the way NTE events are transmitted. SIP service providers can indicate support for RFC 4733 [25] by setting the iteration flag to 1 or indicating support for RFC 2833 [26] by setting the iteration flag to 0.

security: A container that encapsulates characteristics about encrypting signalling streams between the enterprise and SIP service provider networks.

signaling: A container that encapsulates the type of security protocol for the SIP communication between the enterprise SBC and the service provider.

type: A leaf node that specifies the protocol used for protecting SIP signalling messages between the enterprise and service provider network. The value of the type leaf node is only defined for Transport Layer Security (TLS). Accordingly, if TLS is allowed for SIP sessions between the enterprise and service provider network, the type leaf node is set to the string "tls".

version: A leaf node that specifies the version(s) of TLS supported in decimal format. If multiple versions of TLS are supported, they should be separated by semi-colons. If the service provide does not

support TLS for protecting SIP sessions, the signalling element is set to the string "NULL".

mediaSecurity: A container that describes the various characteristics of securing media streams between enterprise and service provider networks.

keyManagement: A leaf node that specifies the key management method used by the service provider. Possible values of this node include: "SDS" and "DTLS-SRTP". A value of "SDS" signifies that the SIP service provider uses the methods defined in RFC 4568 [27] for the purpose of key management. A value of "DTLS-SRTP" signifies that the SIP service provider uses the methods defined in RFC 5764 [28] for the purpose of key management. If the value of this leaf node is set to "DTLS-SRTP", the various versions of DTLS supported by the SIP service provider MUST be encoded as per the formatting rules of Section 9.2. If the service provider does not support media security, the keyManagement node MUST be set to "NULL".

certLocation: If the enterprise network is required to exchange SIP traffic over TLS with the SIP service provider, and if the SIP service provider is capable of accepting TLS connections from the enterprise network, it may be required for the SIP service provider certificates to be pre-installed on the enterprise edge element. In such situations, the certLocation leaf node is populated with a URL, which when dereferenced, provides a single PEM encoded file that contains all certificates in the chain of trust. This is an optional leaf node.

extensions: A leaf node that is a semicolon separated list of all possible SIP option tags supported by the service provider network. These extensions must be referenced using name registered under IANA. If the service provider network does not support any extensions to baseline SIP, the extensions node must be set to "NULL".

9.4. Extending the Capability Set

There are situations in which equipment manufactures or service providers would benefit from extending the YANG module defined in this draft. For example, service providers could extend the YANG module to include information that further simplifies direct IP peering. Such information could include: trunk group identifiers, direct-inward-dial (DID) number ranges allocated to the enterprise, customer/enterprise account numbers, service provider support numbers, among others. Extension of the module can be achieved by importing the module defined in this draft. An example is provided below: Consider a new YANG module "vendorA" specified for VendorA's

enterprise SBC. The "vendorA-config" YANG module is configured as follows:

```
module vendorA-config {
  namespace "urn:ietf:params:xml:ns:yang:vendorA-config";
  prefix "vendorA";

  description
    "Data model for configuring VendorA Enterprise SBC";

  revision 2020-05-06 {
    description "Initial revision of VendorA Enterprise SBC configuration data
model";
  }

  import ietf-peering {
    prefix "peering";
  }

  augment "/peering:peering-info" {
    container vendorAConfig {
      leaf vendorAConfigParam1 {
        type int32;
        description "vendorA configuration parameter 1 (SBC Device ID)";
      }

      leaf vendorAConfigParam2 {
        type string;
        description "vendorA configuration parameter 2 (SBC Device name)";
      }
      description "Container for vendorA SBC configuration";
    }
  }
}
```

In the example above, a custom module named "vendorA-config" uses the "augment" statement as defined in Section 4.2.8 of [RFC 7950 [29]] to extend the module defined in this draft.

10. Example Capability Set Document Encoding

This section provides examples of how capability set documents that leverage the YANG module defined in this document can be encoded over JSON or XML.

10.1. XML Capability Set Document

```

<peering-info xmlns="urn:ietf:params:xml:ns:yang:ietf-peering"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:ietf:params:xml:ns:yang:ietf-peering ietf-peering.x
sd">
  <variant>1.0</variant>
  <transport-info>
    <transport>TCP;TLS;UDP</transport>
    <registrar>registrar1.voip.example.com:5060</registrar>
    <registrar>registrar2.voip.example.com:5060</registrar>
    <registrarRealm>voip.example.com</registrarRealm>
    <callControl>callServer1.voip.example.com:5060</callControl>
    <callControl>192.168.12.25:5065</callControl>
    <dns>8.8.8.8</dns>
    <dns>208.67.222.222</dns>
    <outboundProxy>0.0.0.0</outboundProxy>
  </transport-info>
  <call-specs>
    <earlyMedia>true</earlyMedia>
    <signalingForking>false</signalingForking>
    <supportedMethods>INVITE;OPTIONS;BYE;CANCEL;ACK;PRACK;SUBSCRIBE;NOTIFY;RE
GISTER</supportedMethods>
    <numRange>
      <type>range</type>
      <count>20</count>
      <value>19725455000</value>
    </numRange>
    <numRange>
      <type>block</type>
      <count>2</count>
      <value>1972546000</value>
      <value>1972546001</value>
    </numRange>
  </call-specs>
  <media>
    <mediaTypeAudio>
      <mediaFormat>PCMU;rate=8000;ptime=20</mediaFormat>
      <mediaFormat> G729;rate=8000;annexb=yes</mediaFormat>
      <mediaFormat>G722;rate=8000;bitrate=56k,64k</mediaFormat>
    </mediaTypeAudio>
    <fax>
      <protocol>pass-through</protocol>
      <protocol>t38</protocol>
    </fax>
    <rtp>
      <RTPTrigger>true</RTPTrigger>
      <symmetricRTP>true</symmetricRTP>
    </rtp>
    <rtcp>

```

```

        <symmetricRTCP>true</symmetricRTCP>
        <RTCPFeedback>true</RTCPFeedback>
    </rtcp>
</media>
<dtmf>
    <payloadNumber>101</payloadNumber>
    <iteration>0</iteration>
</dtmf>
<security>
    <signaling>
        <type>TLS</type>
        <version>1.0;1.2</version>
    </signaling>
    <mediaSecurity>
        <keyManagement>SDES;DTLS-SRTP,version=1.2</keyManagement>
    </mediaSecurity>
    <certLocation>https://sipserviceprovider.com/certificateList.pem</certLoc
ation>
    </security>
    <extensions>timer;rel100;gin;path</extensions>
</peering-response>

```

11. Example Exchange

This section depicts an example of the configuration flow that ultimately results in the enterprise edge element obtaining the capability set document from the SIP service provider. Assuming the enterprise edge element has been pre-configured with the request target for the capability set document or has dynamically found the request target, the edge element generates a HTTPS GET request. This request can be challenged by the service provider to authenticate the enterprise.

```

GET //capdoc?trunkid=trunkent1456 HTTP/1.1
Host: capserver.sspl.com
Accept:application/peering-info+xml

```

The capability set document is obtained in the body of the response and is encoded in XML.

```

HTTP/1.1 200 OK
Content-Type: application/peering-info+xml
Content-Length: nnn

<peering-info>
...
</peering-info>

```

12. Security Considerations

[TBD]

13. References

13.1. Normative References

[1] Bradner, S., "Key Words for Use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119 [30], March 1997.

[4] Bjorklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020 [31], October 2010.

[8] Schoenwaelder, J., "Common YANG Data Types", RFC 6991 [32], July 2013.

13.2. Informative References

[2] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261 [33], June 2002.

[3] Roach, A. B., "SIP-Specific Event Notification", RFC 6665 [34], July 2012.

[5] Fielding, R., Reschke, J., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231 [35], June 2014.

[6] Enns, R., Bjorklund, M., Schoenwaelder, J., Bierman, A., "Network Configuration Protocol (NETCONF)", RFC 6241 [36], June 2011.

[7] Bjorklund, M., Berger, L., "YANG Tree Diagrams", RFC 8340 [37], March 2018.

[9] Wing, D., "Symmetric RTP / RTP Control Protocol (RTCP)", RFC 4961 [38], July 2007.

[10] Ott, J., Wenger, S., Sato, N., Burmeister, C., Matsushita, "Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF)", RFC 4585 [39], July 2006.

[11] Schulzrinne, H., Petrack, S., "RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals", RFC 2833 [40], May 2000.

[12] Schulzrinne, H., Taylor, T., "RTP Payload for DTMF Digits, Telephony Tones, and Telephony Signals", RFC 4733 [41], December, 2006.

[13] Casner, S., "Media Type Registration of RTP Payload Formats", RFC 4855 [42], February 2007.

[14] Dierks, T., Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246 [43], August 2008.

[15] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446 [44], August 2018.

[17] Ivov, E., Kaplan, H., Wing, D., "Latching: Hosted NAT Traversal (HNT) for Media in Real-Time Communication", RFC 7362 [45], September 2014.

[18] Jones, P., Salgueiro, G., Jones, M., Smarr, J., "WebFinger", [RFC 7033 [46]], September 2013.

[19] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", [RFC 6749 [47]], October 2012.

[20] "SIP-PBX / Service Provider Interoperability SIPconnect 2.0 - Technical Recommendation", SIP Forum, SIP CONNECT 2.0 [48], December 7, 2016

14. Acknowledgments

[TBD]

15. References

15.1. URIs

[1] <https://tools.ietf.org/html/rfc3261>

[2] <https://www.sipforum.org/download/sipconnect-technical-recommendation-version-2-0/?wpdmdl=2818>

[3] <https://tools.ietf.org/html/rfc2119>

[4] <https://tools.ietf.org/html/rfc7092>

[5] <https://tools.ietf.org/html/rfc7231>

[6] <https://tools.ietf.org/html/rfc7092>

[7] <https://tools.ietf.org/html/rfc6665>

[8] <https://tools.ietf.org/html/rfc6020>

- [9] <https://tools.ietf.org/html/rfc6241>
- [10] <https://tools.ietf.org/html/rfc5246>
- [11] <https://tools.ietf.org/html/rfc7235>
- [12] <https://tools.ietf.org/html/rfc7230>
- [13] <https://tools.ietf.org/html/rfc8340>
- [14] <https://tools.ietf.org/html/rfc6991>
- [15] <https://tools.ietf.org/html/rfc4855>
- [16] <https://tools.ietf.org/html/rfc4961>
- [17] <https://tools.ietf.org/html/rfc7362>
- [18] <https://tools.ietf.org/html/rfc4585>
- [19] <https://tools.ietf.org/html/rfc4585>
- [20] <https://tools.ietf.org/html/rfc4961>
- [21] <https://tools.ietf.org/html/rfc2833>
- [22] <https://tools.ietf.org/html/rfc4733>
- [23] <https://tools.ietf.org/html/rfc4733>
- [24] <https://tools.ietf.org/html/rfc2833>
- [25] <https://tools.ietf.org/html/rfc4733>
- [26] <https://tools.ietf.org/html/rfc2833>
- [27] <https://tools.ietf.org/html/rfc4568>
- [28] <https://tools.ietf.org/html/rfc5764>
- [29] <https://tools.ietf.org/html/rfc7950>
- [30] <https://tools.ietf.org/html/rfc2119>
- [31] <https://tools.ietf.org/html/rfc6020>
- [32] <https://tools.ietf.org/html/rfc6991>

- [33] <https://tools.ietf.org/html/rfc3261>
- [34] <https://tools.ietf.org/html/rfc6665>
- [35] <https://tools.ietf.org/html/rfc7231>
- [36] <https://tools.ietf.org/html/rfc6241>
- [37] <https://tools.ietf.org/html/rfc8040>
- [38] <https://tools.ietf.org/html/rfc4961>
- [39] <https://tools.ietf.org/html/rfc4585>
- [40] <https://tools.ietf.org/html/rfc2833>
- [41] <https://tools.ietf.org/html/rfc4733>
- [42] <https://tools.ietf.org/html/rfc4855>
- [43] <https://tools.ietf.org/html/rfc5246>
- [44] <https://tools.ietf.org/html/rfc8446>
- [45] <https://tools.ietf.org/html/rfc7362>
- [46] <https://tools.ietf.org/html/rfc7033>
- [47] <https://tools.ietf.org/html/rfc6749>
- [48] <https://www.sipforum.org/download/sipconnect-technical-recommendation-version-2-0-2/>

Authors' Addresses

Kaustubh Inamdar
Cisco Systems

Email: kinamdar@cisco.com

Sreekanth Narayanan
Cisco Systems

Email: sreenara@cisco.com

Cullen Jennings
Cisco Systems

Email: fluffy@iii.ca

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

J. Rosenberg
Five9
C. Jennings
Cisco Systems
A. Minessale
Signalwire/Freeswitch
J. Livingood
Comcast
J. Uberti
Google
July 8, 2019

Real Time Internet Peering Protocol
draft-rosenbergjennings-dispatch-ripp-03

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.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

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

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Background	3
1.2. Problem Statement	4
1.3. Solution	5
1.4. Why Now?	5
2. Solution Requirements	6
3. Design Approaches	7
3.1. HBH, not E2E	7
3.2. Client-Server, not Agent-to-Agent	8
3.3. Signaling and Media Together	8
3.4. URIs not IPs	9
3.5. OAuth not MTLs or private IP	9
3.6. TLS not SRTP or SIPS	10
3.7. Authenticated CallerID	10
3.8. Calls Separate from Connections	11
3.9. Path Validation, not ICE	11
4. Reference Architecture	11
5. Terminology	14
6. Overview of Operation	15
7. Example	18
7.1. Inbound Call	20
7.2. Outbound Call	21
7.3. End of call	21
8. Detailed Behaviours	22

8.1.	Configuration	22
8.2.	RIPP Trunk Provisioning	22
8.3.	Capabilities	24
8.4.	Initiating Calls	26
8.5.	Establishing the Signaling Byways	28
8.6.	The Media Sequence	28
8.7.	Opening Media Byways	29
8.8.	Sending and Receiving Media	30
8.9.	Terminating and Re-establishing Connections and Byways .	31
8.10.	Signaling - Events	31
8.11.	Call Termination	33
8.12.	GET Transactions	34
8.13.	Graceful Call Migration: Server	34
8.14.	Graceful Call Migration: Client	34
8.15.	Ungraceful Call Migration	34
9.	SIP Gateway	35
9.1.	RIPP to SIP	36
9.2.	SIP to RIPP	36
10.	RAML API	36
11.	IANA Considerations	39
12.	Security Considerations	39
13.	Acknowledgements	39
14.	Informative References	39
	Authors' Addresses	40

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 based 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.

- o 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.

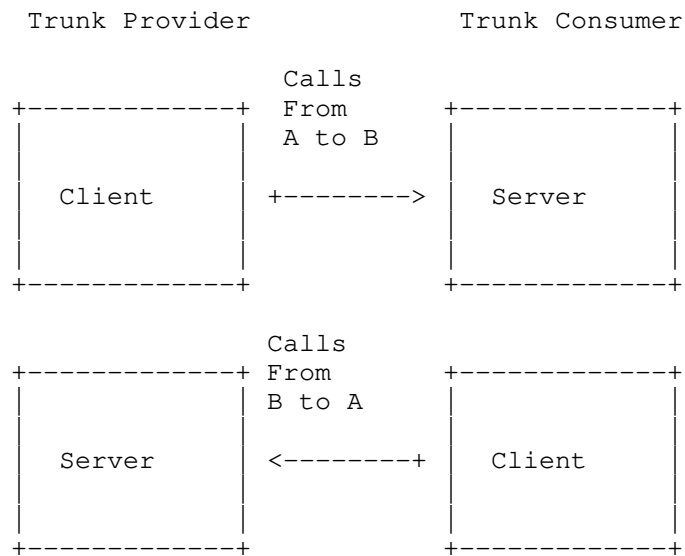
3.9. Path Validation, not ICE

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.

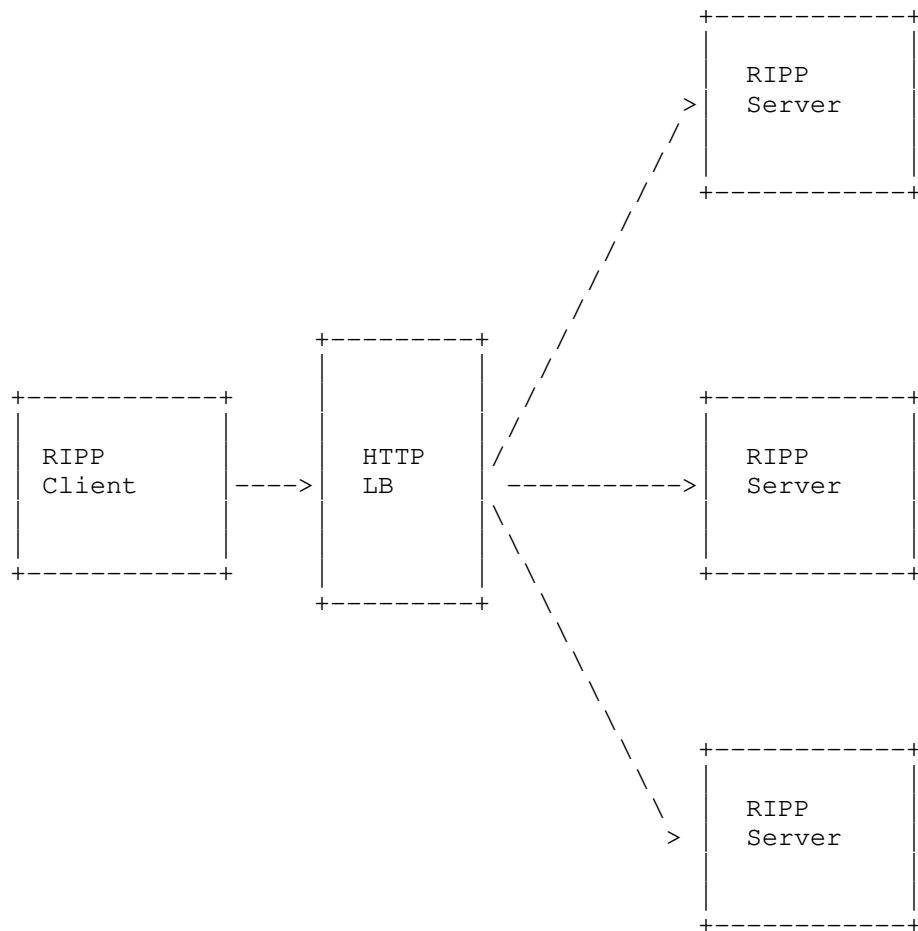
4. Reference Architecture

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 of 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.

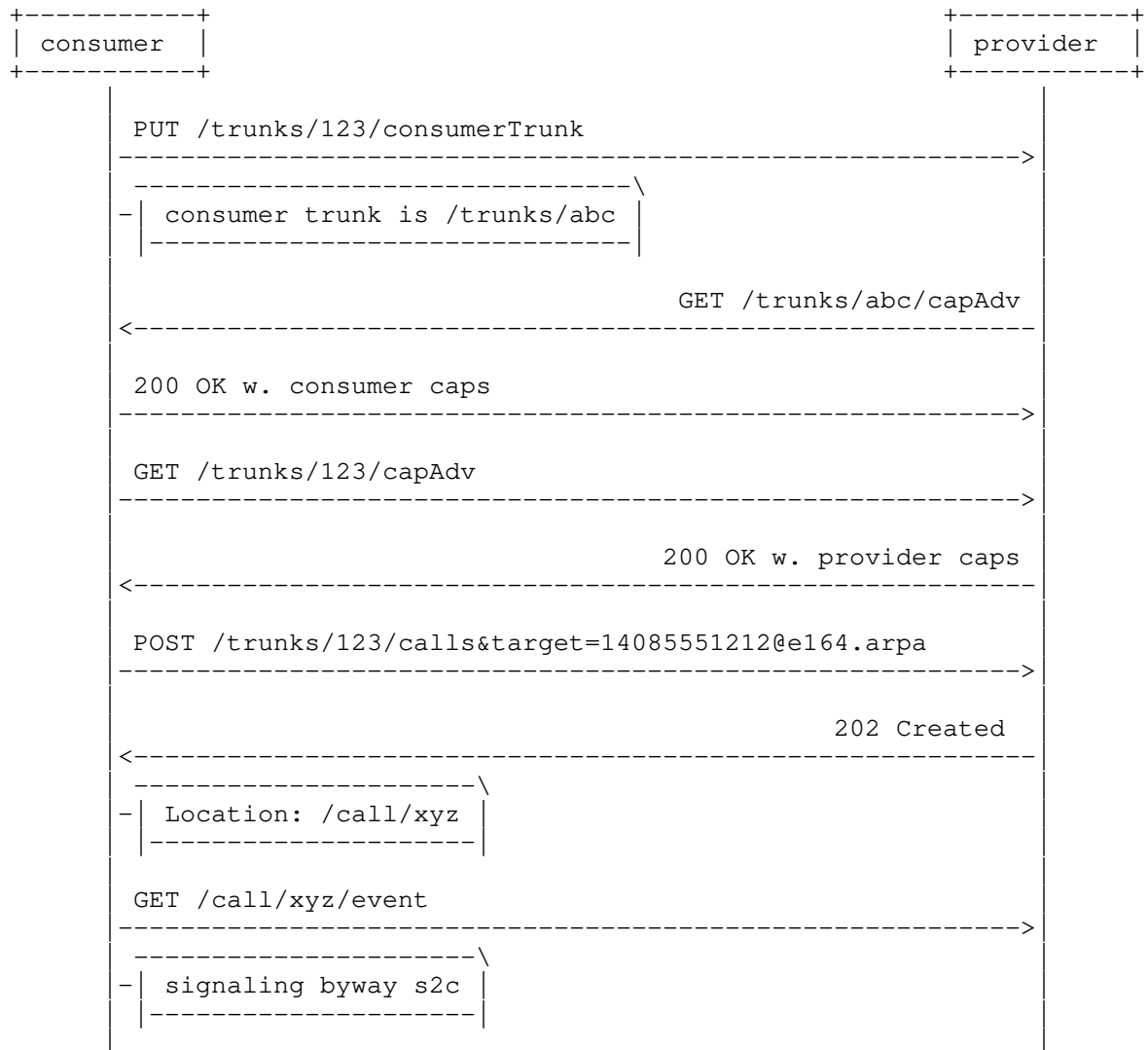
RIPP provides a simple technique for allowing a call to seamlessly migrate from one client instance to another on a different host, or from one server instance to another on a different host. For a client, it need only end the byways in use for the call and re-initiate from a different instance. Similarly, 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:



```

200 OK
<-----
POST /call/xyz/media-forward
----->
- |-----\
  | media byway c2s |
  |-----|

200 OK
<-----
POST /call/xyz/media-reverse
----->
- |-----\
  | media byway s2c |
  |-----|

200 OK
<-----
ringing eg. SIP 180
- |-----\

ringing event
<-----
accepted eg. SIP 200
- |-----\

accepted
<-----
- |-----\
  | caller hangs up |
  |-----|

PUT /call/xyz/event
----->
- |-----\
  | end event |
  |-----|

200 OK
<-----

```

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 make an HTTP post to the consumer trunk URI to set up the incoming call. This is done 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 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 cause 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 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.

- o 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.

- o 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.
- o 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.
- o 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.
- o 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".
- o 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.

- o 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 than 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:

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

- o 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.

- o CJ - Not keen on SIP reason codes - they did not contain enough info for all the Q950 stuff and were not particularly 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.

- o CJ - would it work to have this all far simpler and just have the trunk cap advertisement 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 provider 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 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.

- o CJ - lets call this previousEvent 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.

- o 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: Retrieve the previous event from server
        responses:
          200:
            body:
              application/json:
                type: Event
    /event:
      get:
        description: Wait for next event then retrieve the most recent e
vent 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 clien
t 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 serve
r to client
        body:
          application/octet-stream:
            type: BywayFlow
        responses:
          200:
            application/octet-stream:
              type: BywayFlow
```

11. IANA Considerations
12. Security Considerations
13. Acknowledgements
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

Jason Livingood
Comcast

Email: jason_livingood@comcast.com

Internet-Draft

RIPP

July 2019

Justin Uberti
Google

Email: justin@uberti.name