Workgroup: ALTO Working Group
Internet-Draft:
draft-ietf-alto-new-transport-00
Published: 22 June 2022
Intended Status: Standards Track
Expires: 24 December 2022
Authors: R. Schott Y. Yang
Deutsche Telekom Yale University
K. Gao J. Zhang
Sichuan University Tongji University

ALTO/H2: The ALTO Protocol using HTTP/2

Abstract

The ALTO base protocol [RFC7285] uses HTTP/1.x as the transport protocol and hence ALTO transport includes the limitations of HTTP/1.x. ALTO/SSE [RFC8895] addresses some of the limitations, but is still based on HTTP/1.x. This document introduces ALTO new transport, which provides the transport functions of ALTO/SSE on top of HTTP/2, for more efficient ALTO transport.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119][RFC8174] when, and only when, they appear in all capitals, as shown here.

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 24 December 2022.

Copyright Notice

Copyright (c) 2022 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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction
2. ALTO/H2 Design Requirements
3. ALTO/H2 Design Overview
4. Transport Queue
  4.1. Transport Queue Operations
  4.2. Examples
5. Incremental Updates Queue
  5.1. Incremental Updates Queue Operations
  5.2. Examples
6. Individual Updates
  6.1. Individual Updates Operations
  6.2. Examples
7. Receiver Set
  7.1. Receiver Set Operations
  7.2. Examples
8. ALTO/H2 Stream Management
  8.1. Objectives
  8.2. Client -> Server [Create Transport Queue]
  8.3. Client -> Server [Close Transport Queue]
  8.4. Client -> Server [Request on Data of a Transport Queue on Stream SID_tq]
  8.5. Server -> Client [PUSH_PROMISE for Transport Queue on Stream SID_tq]
  8.6. Concurrency Management
9. ALTO/H2 Information Resource Directory (IRD)
10. Security Considerations
11. IANA Considerations
12. Acknowledgments
13. References
  13.1. Normative References
  13.2. Informative References
Appendix A. Outlook to ALTO with HTTP/3

Authors' Addresses

1. Introduction

Application-Layer Traffic Optimization (ALTO) provides a means for network applications to obtain network status information. The ALTO base protocol [RFC7285] is based on the sequential request and response model of HTTP/1.1 [RFC7230]; hence, in the base protocol, an ALTO client can issue only a sequence of requests on network information resources, and the ALTO server sends the information resources one-by-one, in the order of the request sequence.

To address the use cases where an ALTO client may need to efficiently monitor changes to a set of network information resources and the protocol is still based on the HTTP/1.1 model, the ALTO Working Group introduces ALTO/SSE (ALTO Incremental Update based on Server-Sent-Event) [RFC8895], so that an ALTO client can manage (i.e., add and remove) a set of requests maintained at an ALTO server, and the server can continuously, concurrently, and incrementally push updates whenever a monitored network information resource changes. Figure 1 shows the architecture and message flow of ALTO/SSE, which can be considered as a more general transport protocol than the ALTO base transport protocol. Although ALTO/SSE allows the concurrent transport of multiple ALTO information resources, it has complexities and limitations. For example, it requires that the server provide a separate control URI, leading to complexity in management.

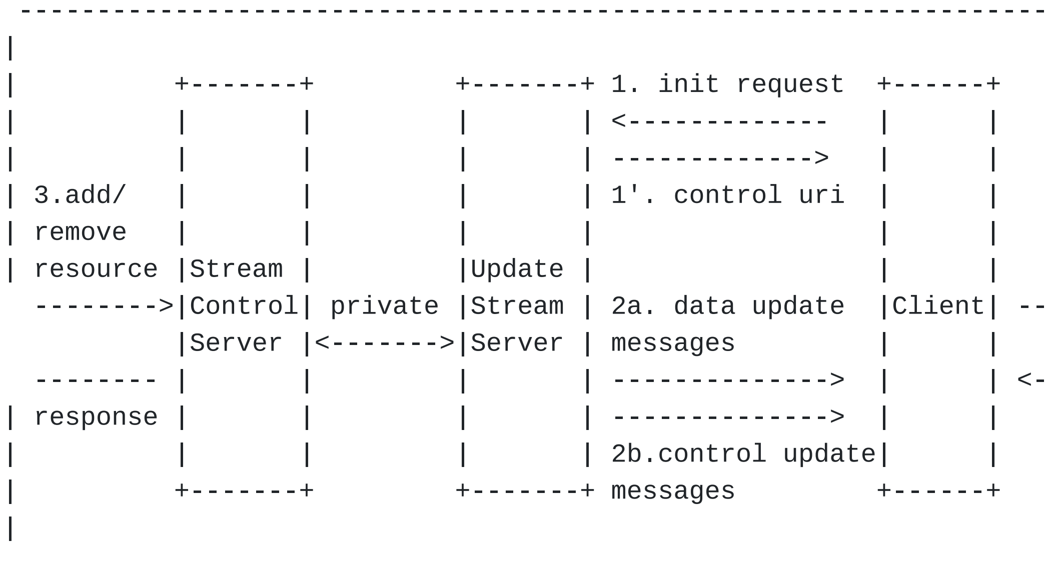

Figure 1: ALTO SSE Architecture and Message Flow.

This document specifies ALTO/H2, which realizes ALTO/SSE but takes advantage of new HTTP capabilities provided by HTTP/2 [RFC7540].

2. ALTO/H2 Design Requirements

ALTO/H2 is designed to satisfy a set of requirements. First, it should satisfy the following requirements to realize the functions of ALTO/SSE:

- R0: Client can request any resource using the connection, just as using ALTO base protocol using HTTP/1.x.
- R1: The client can request the addition (start) of incremental updates to a resource.
- R2: The client can request the deletion (stop) of incremental updates to a resource.
- R3: The server can signal to the client the start or stop of incremental updates to a resource.
- R4: The server can choose the type of each incremental update endcoding, as long as the type is indicated to be acceptable by the client.

Following the ALTO framework [RFC7285] [RFC7971], ALTO/H2 should still be HTTP based:

- R5: The design follows basic principle of HTTP---Representational State Transfer and hence can use only HTTP verbs (GET, POST, PUT, DELETE, HEAD).
- R6: The design takes advantage of HTTP/2 design features such as parallel transfer and respects HTTP/2 semantics such as the semantics of PUSH_PROMISE.

To allow flexible deployment, the new transport protocol should be flexible:

- R7: The design should support capability negotiation.

3. ALTO/H2 Design Overview

A key design of ALTO new transport is to distinguish between information about ALTO resources and information about ALTO transport. It introduces the following transport information structures to distribute ALTO information resources:

- The transport state from the ALTO server to an ALTO client (or a set of clients) for an ALTO information resource is conceptually

through a transport queue. A static ALTO information resource (e.g., Cost Map, Network Map) has a single transport queue, and a dynamic ALTO information resource (e.g., Filtered Cost Map) may create a queue for each unique filter request.

* Each transport queue maintains two states: (1) the incremental update message queue, which includes a sequence of incremental update messages and (2) the receiver set, which includes the set of receivers receiving incremental push updates from the ALTO server.

* The transport queue state is exposed to clients through views; that is, a client can see only a virtual view of the server state.

Figure 2 shows an example illustrating the aforementioned information. Each ALTO client (Client 1, Client 2, Client 3) maintains a single HTTP/2 connection with the ALTO server.

Information Resource:

a) Static resource such as NetworkMap
b) Filterable resource such as FilteredCostMap

```
                              +-------------+
                              |             |
         +--------------------| ALTO Server |-----------+
         |                  +-|             |-+         |
         |                  | +-------------+ |         |
         |                  |                 |         |
---------|------------------|-----------------|---------|------------
         |                  |                 |         | Information
         |                  |                 |         | Resource
+-------------+   +-------------+   +-------------+   +-------------+
| Information |   | Information |   | Information |   | Information |
| Resource #1 |   | Resource #2 |   | Resource #3 |   | Resource #4 |
+-------------+   +-------------+   +-------------+   +-------------+
       |                              /    \
-------|-----------------------------/------\------------------------
       |                            /        \            Transport
       |                      +----/          \------+    Queue
       |                      |                      |
  +--------+             +--------+             +--------+
  |   tq1  |-----+       |   tq2  |-----+       |   tq3  |-----+
  +----|---+     |       +----|---+     |       +----|---+     |
       |         |            |         |            |         |
  +----|---+ +---|----+  +----|---+ +---|----+  +----|---+ +---|----+
  | tq1/uq | | tq1/rs |  | tq2/uq | | tq2/rs |  | tq3/uq | | tq3/rs |
  +--------+ +--------+  +--------+ +--------+  +--------+ +--------+
       |\       /\              |         /           |          |
-------|-\-----/--\-------------|--------/------------|----------|---
       |  \   /    +-------+    |       /             |          |
       |   +-/-----------+  \   |      /              |          |
       |    /             \  \  |     /               +          +
       |   /            +--\--\-|----/--+ single       \        /
       |  /             +---\--\|---/---+ http2/3       \      /
   +----------+             +----------+    connection +----------+
   | Client 1 |             | Client 2 |               | Client 3 |
   +----------+             +------- --+               +----------+
```

tq   = transport queue
tq/uq = incremental updates queue
sq/rs = receiver set

Figure 2: ALTO New Transport Information Structure.

The basic work flow of a client connecting to an ALTO server is the following:

```
Client opens a connection to the server
Client opens/identifies a transport queue tq
  Client requests transport queue status of tq
  Client requests an element in the incremental update queue

  Client becomes a receiver
  Client receives incremental push updates
Client closes the transport queue tq
Client closes the connection
```

Figure 3: ALTO New Transport Information Structure.

4. Transport Queue

4.1. Transport Queue Operations

A transport queue supports three basic operations (CRD): create, read (get status), and delete.

Create a transport queue: An ALTO client creates a transport queue using the HTTP POST method with ALTO SSE AddUpdateReq ([RFC 8895] Sec. 6.5) as the parameter:

```
object {
    ResourceID   resource-id;
    [JSONString  tag;]
    [Boolean     incremental-changes;]
    [Object      input;]
 } AddUpdateReq;
```

A successful POST request MUST return the URI for the transport queue. Unless the request has incremental-changes to be false, the client is added to receiver set as well, indicating that the client will receive automatic, incremental push updates.

Read a transport queue: A client reads the status of a transport queue by issuing a GET request to the transport queue URI returned from the POST method.

Delete a transport queue: a transport queue exposed to a client can be closed (deleted) either explicitly or implicitly.

* Explicit delete: A client uses the HTTP DELETE method to explicitly delete a transport queue. If successful, the transport queue is deleted from the local view of the client, although the server may still maintain the transport queue for other client connections.

* Implicit delete: Transport queue for a client is ephemeral: the close of the HTTP connection between the client and the server deletes the transport queue from the client's view --- when the client reconnects, the client MUST NOT assume that the transport queue is still valid.

Error codes: ALTO/H2 uses HTTP error codes.

4.2. Examples

The first example is a client creating a transport queue.

```
Client -> server request

HEADERS
  - END_STREAM
  + END_HEADERS
    :method = POST
    :scheme = https
    :path = /tqs
    host = alto.example.com
    accept = application/alto-error+json,
                 application/alto-transport+json
    content-type = application/alto-transport+json
    content-length = TBD

DATA
 - END_STREAM
 {
    "resource-id": "my-routingcost-map"
 }
```

```
Server -> client response:

HEADERS
  - END_STREAM
  + END_HEADERS
    :status = 200
    content-type = application/alto-transport+json
    content-length = TBD

DATA
  - END_STREAM
   {"tq": “/tqs/2718281828459”}
```

The client can then read the status of the transport queue using the read operation (GET) in the same HTTP connection. Below is an example (structure of incremental updates queue will be specified in the next section):

```
Client -> server request

HEADERS
  - END_STREAM
  + END_HEADERS
    :method = GET
    :scheme = https
    :path = /tqs/2718281828459
    host = alto.example.com
    accept = application/alto-error+json,
                 application/alto-transport+json

Server -> client response:

HEADERS
  - END_STREAM
  + END_HEADERS
    :status = 200
    content-type = application/alto-transport+json
    content-length = TBD

DATA
  - END_STREAM
 { "uq":
    [
      {“seq”:        101,
       "media-type": "application/alto-costmap+json",
       “tag”:        "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe" },
      {“seq”:        102,
       "media-type": "application/merge-patch+json",
       “tag”:        "cdf0222x59740b0b2e3f8eb1d4785acd42231bfe" },
      {“seq”:        103,
       "media-type": "application/merge-patch+json",
       “tag”:        "8eb1d4785acd42231bfecdf0222x59740b0b2e3f",
       "link":       "/tqs/2718281828459/snapshot/2e3f"}

    ],
  "rs": ["self"]
 }
```

5. Incremental Updates Queue

5.1. Incremental Updates Queue Operations

Among the CRUD operations, an incremental updates queue supports only the read operation: a client cannot create, update, or delete incremental updates queue directly---it is read only, and associated with transport queue automatically.

Reads an incremental updates queue: A client reads the status of an incremental updates queue using the HTTP GET method: GET transport-queue-uri/uq, where the transport-queue-uri is the URI returned in the transport queue create method.

The response informs the client the backlog status, and potential direct links. Specifically, the response is a JSON array, with each element being one incremental update, with three required fields and one optional field:

* "seq": a required JSON integer indicating the sequence number of the incremental update; As JSON allows a large integer space, when the server reaches the largest integer, the server SHOULD close the incremental update queue;

* "media-type", a required JSON string giving the type of the incremental update (see ALTO/SSE);

* "tag": a required JSON string giving a unique tag (see [RFC7285];

* "link": an optional JSON string giving an optional link for a client to directly request a resource as a complete snapshot (not through incremental updates).

Note that the server determines the state (window of history and type of each update) in the incremental updates queue, as specified by [R4].

5.2. Examples

Assume the same example in the preceding section. The client can check the status of the incremental updates queue of a transport queue from the same connection:

```
Client -> server request:

HEADERS
  - END_STREAM
  + END_HEADERS
    :method = GET
    :scheme = https
    :path = /tqs/2718281828459/uq
    host = alto.example.com
    accept = application/alto-error+json,
                 application/alto-transport+json

Server -> client response:

HEADERS
  - END_STREAM
  + END_HEADERS
    :status = 200
    content-type = application/alto-transport+json
    content-length = TBD

DATA
  - END_STREAM
 {
    [
      {“seq”:        101,
       "media-type": "application/alto-costmap+json",
       “tag”:        "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe" },
      {“seq”:        102,
       "media-type": "application/merge-patch+json",
       “tag”:        "cdf0222x59740b0b2e3f8eb1d4785acd42231bfe" },
      {“seq”:        103,
       "media-type": "application/merge-patch+json",
       “tag”:        "8eb1d4785acd42231bfecdf0222x59740b0b2e3f",
       "link":       "/tqs/2718281828459/snapshot/2e3f"}

    ],

 }
```

6. Individual Updates

6.1. Individual Updates Operations

A client can only read an individual update. The read can be either pull read issued by the client or a push from the server to the client.

Client pull read: A client uses HTTP GET method on the incremental updates queue concatenated by the "seq" to pull an individual update.

Server push read: a client starts to receive server push when it is added to the receiver set. A client can add itself to the receiver set when creating the transport queue, or add itself explicitly to the receiver set (see the next section).

The work flow of server push of individual updates is the following:

* Initialization: the first update pushed from the server to the client MUST be the later of the following two: (1) the last independent update in the incremental updates queue; and (2) the following entry of the entry that matches the tag when the client creates the transport queue. The client MUST set SETTINGS_ENABLE_PUSH to be consistent.

* Push state: the server MUST maintain the last entry pushed to the client (and hence per client, per connection state) and schedule next update push accordingly.

* Push management: The client MUST NOT cancel (RST_STREAM) a PUSH_PROMISE to avoid complex server state management.

6.2. Examples

The first example is a client pull example, in which the client directly requests an individual update.

```
Client -> server request:

HEADERS
  + END_STREAM
  + END_HEADERS
    :method = GET
    :scheme = https
    :path = /tqs/2718281828459/uq/101
    host = alto.example.com
    accept = application/alto-error+json,
                 application/alto-costmap+json

Server -> client response:

HEADERS
  - END_STREAM
  + END_HEADERS
    :status = 200
    content-type = application/alto-costmap+json
    content-length = TBD

DATA
  + END_STREAM
 {
   "meta" : {
      "dependent-vtags" : [{
         "resource-id": "my-network-map",
         "tag": "da65eca2eb7a10ce8b059740b0b2e3f8eb1d4785"
       }],
      "cost-type" : {
        "cost-mode"  : "numerical",
        "cost-metric": "routingcost"
      },
      "vtag": {
        "resource-id" : "my-routingcost-map",
        "tag" : "3ee2cb7e8d63d9fab71b9b34cbf764436315542e"
      }
   },
   "cost-map" : {
     "PID1": { "PID1": 1,  "PID2": 5,  "PID3": 10 },
     "PID2": { "PID1": 5,  "PID2": 1,  "PID3": 15 },
     "PID3": { "PID1": 20, "PID2": 15  }
   }
}
```

Note from the transport queue state that the 103 message has an OPTIONAL link to a complete snapshot, which a client can request.

Instead of directly requesting, the client can wait for the server
for incremental push, where the server first sends PUSH_PROMISE with
the GET URI as above.

```
Server -> client PUSH_PROMISE in current stream:

PUSH_PROMISE
  - END_STREAM
    Promised Stream 4
    HEADER BLOCK
    :method = GET
    :scheme = https
    :path = /tqs/2718281828459/uq/101
    host = alto.example.com
    accept = application/alto-error+json,
                 application/alto-costmap+json

Server -> client content Stream 4:

HEADERS
  + END_STREAM
  + END_HEADERS
    :status = 200
    content-type = application/alto-costmap+json
    content-length = TBD

DATA
  + END_STREAM
 {
   "meta" : {
      "dependent-vtags" : [{
         "resource-id": "my-network-map",
         "tag": "da65eca2eb7a10ce8b059740b0b2e3f8eb1d4785"
       }],
      "cost-type" : {
        "cost-mode"  : "numerical",
        "cost-metric": "routingcost"
      },
      "vtag": {
        "resource-id" : "my-routingcost-map",
        "tag" : "3ee2cb7e8d63d9fab71b9b34cbf764436315542e"
      }
   },
   "cost-map" : {
     "PID1": { "PID1": 1,  "PID2": 5,  "PID3": 10 },
     "PID2": { "PID1": 5,  "PID2": 1,  "PID3": 15 },
     "PID3": { "PID1": 20, "PID2": 15  }
   }
}

Server -> client PUSH_PROMISE in current stream:

PUSH_PROMISE
```

```
  - END_STREAM
    Promised Stream 6
    HEADER BLOCK
    :method = GET
    :scheme = https
    :path = /tqs/2718281828459/uq/102
    host = alto.example.com
    accept = application/alto-error+json,
                 application/merge-patch+json

Server -> client content Stream 6

HEADERS
  + END_STREAM
  + END_HEADERS
    :status = 200
    content-type = application/merge-patch+json
    content-length = TBD

DATA
  + END_STREAM
 { ...}
```

7. Receiver Set

7.1. Receiver Set Operations

Among the CRUD operations, a client can add to or delete itself from the receiver set of a transport queue. It can also read the status of the receiver set.

Creat: A client can add itself in the receiver set by using the HTTP PUT method: DELETE transport-queue/rs/self

Read: A client can see only itself in the receiver set. The appearance of self in the receiver set (read does not return not exists) is an indication that push starts.

Delete: A client can delete itself (stops receiving push) either explicitly or implicitly.

* Explicit delete: A client deletes itself using the HTTP DELETE method: DELETE transport-queue/rs/self.

* Implicit delete: Transport queue is connection ephemeral: close of connection or stream for the transport queue deletes the transport queue (from the view) for the client.

7.2. Examples

A client can stop incremental push updates by sending the request:

```
DELETE /tqs/2718281828459/rs/self HTTP/2
Accept: application/alto-transport+json


HTTP/2 200 OK
```

8. ALTO/H2 Stream Management

8.1. Objectives

A main benefit of using HTTP/2 for ALTO is to take advantage of HTTP/2 streams. In particular, the objectives of ALTO/H2 include:

* Allow stream concurrency to reduce latency

* Minimize the number of streams created

* Enforce dependency among streams (so that if A depends on B, then A should be sent before B)

* Encode dependency to enforce semantics (correctness)

To realize the objectives specified in the preceding section, ALTO/H2 MUST satisfy the following stream management requirements in all 4 phases specified in the next 4 subsections.

8.2. Client -> Server [Create Transport Queue]

Each request to create a transport queue (POST) MUST choose a new client selected stream ID (SID_tq), with the following requirements:

* Stream Identifier of the frame is a new client-selected stream ID; Stream Dependency in HEADERS is 0 (connection) for an independent resource, the other transport queue if the dependency is known.

* Invariant: Stream keeps open until close or error.

8.3. Client -> Server [Close Transport Queue]

DELETE to close a transport queue (SID_tq) MUST be sent in SID_tq, with the following requirements:

* Stream Identifier of the frame is SID_tq, and Stream Dependency in HEADER is 0 (connection), so that a client cannot close a different stream.

* HEADERS indicates END_STREAM; server response SHOULD close the stream.

8.4. Client -> Server [Request on Data of a Transport Queue on Stream SID_tq]

The request and response MUST satisfy the following requirements:

* The Stream Identifier of the frame is a new client-selected stream ID, and Stream Dependency in HEADERs MUST be SID_tq, so that a client cannot issue request on a closed transport queue;

* Both the request and the response MUST indicate END_STREAM.

8.5. Server -> Client [PUSH_PROMISE for Transport Queue on Stream SID_tq]

The server push MUST satisfy the following requirements:

* PUSH_PROMISE MUST be sent in stream SID_tq to serialize to allow the client to know the push order;

* Each PUSH_PROMISE chooses a new server-selected stream ID, and the stream is closed after push.

8.6. Concurrency Management

* ALTO/H2 must allow concurrency control using the SETTINGS_MAX_CONCURRENT_STREAMS option in HTTP/2.

* From the client to the server direction, there MUST be one stream for each open transport queue, and hence a client can always close a transport queue (which it uses to open the stream) and hence can also close, without the risk of deadlock.

* From the server to the client direction, each push needs to open a new stream and this should be controlled bu SETTINGS_MAX_CONCURRENT_STREAMS.

9. ALTO/H2 Information Resource Directory (IRD)

Extending the IRD example in Section 8.1 of [RFC8895], below is the IRD of an ALTO server supporting ALTO base protocol, ALTO/SSE, and ALTO/H2.

In particular,

```
"my-network-map": {
  "uri": "https://alto.example.com/networkmap",
  "media-type": "application/alto-networkmap+json",
},
"my-routingcost-map": {
  "uri": "https://alto.example.com/costmap/routingcost",
  "media-type": "application/alto-costmap+json",
  "uses": ["my-networkmap"],
  "capabilities": {
    "cost-type-names": ["num-routingcost"]
  }
},
"my-hopcount-map": {
  "uri": "https://alto.example.com/costmap/hopcount",
  "media-type": "application/alto-costmap+json",
  "uses": ["my-networkmap"],
  "capabilities": {
    "cost-type-names": ["num-hopcount"]
  }
},
"my-filtered-cost-map": {
  "uri": "https://alto.example.com/costmap/filtered/constraints",
  "media-type": "application/alto-costmap+json",
  "accepts": "application/alto-costmapfilter+json",
  "uses": ["my-networkmap"],
  "capabilities": {
    "cost-type-names": ["num-routingcost", "num-hopcount"],
    "cost-constraints": true
  }
},
"my-simple-filtered-cost-map": {
  "uri": "https://alto.example.com/costmap/filtered/simple",
  "media-type": "application/alto-costmap+json",
  "accepts": "application/alto-costmapfilter+json",
  "uses": ["my-networkmap"],
  "capabilities": {
    "cost-type-names": ["num-routingcost", "num-hopcount"],
    "cost-constraints": false
  }
},
"my-props": {
  "uri": "https://alto.example.com/properties",
  "media-type": "application/alto-endpointprops+json",
  "accepts": "application/alto-endpointpropparams+json",
  "capabilities": {
    "prop-types": ["priv:ietf-bandwidth"]
  }
},
"my-pv": {
```

```
      "uri": "https://alto.example.com/endpointcost/pv",
      "media-type": "multipart/related;
                     type=application/alto-endpointcost+json",
      "accepts": "application/alto-endpointcostparams+json",
      "capabilities": {
        "cost-type-names": [ "path-vector" ],
        "ane-properties": [ "maxresbw", "persistent-entities" ]
      }
    },
    "update-my-costs": {
      "uri": "https://alto.example.com/updates/costs",
      "media-type": "text/event-stream",
      "accepts": "application/alto-updatestreamparams+json",
      "uses": [
         "my-network-map",
         "my-routingcost-map",
         "my-hopcount-map",
         "my-simple-filtered-cost-map"
      ],
      "capabilities": {
        "incremental-change-media-types": {
          "my-network-map": "application/json-patch+json",
          "my-routingcost-map": "application/merge-patch+json",
          "my-hopcount-map": "application/merge-patch+json"
        },
        "support-stream-control": true
      }
    },
    "update-my-costs-h2": {
      "uri": "https://alto.example.com/updates-h2/costs",
      "media-type": "application/alto-h2",
      "accepts": "application/alto-updatestreamparams+json",
      "uses": [
         "my-network-map",
         "my-routingcost-map",
         "my-hopcount-map",
         "my-simple-filtered-cost-map"
      ],
      "capabilities": {
        "incremental-change-media-types": {
          "my-network-map": "application/json-patch+json",
          "my-routingcost-map": "application/merge-patch+json",
          "my-hopcount-map": "application/merge-patch+json"
        },
        "support-stream-control": true
      }
    },

    "update-my-props": {
```

```
    "uri": "https://alto.example.com/updates/properties",
    "media-type": "text/event-stream",
    "uses": [ "my-props" ],
    "accepts": "application/alto-updatestreamparams+json",
    "capabilities": {
      "incremental-change-media-types": {
        "my-props": "application/merge-patch+json"
      },
      "support-stream-control": true
    }
  },
  "update-my-pv": {
    "uri": "https://alto.example.com/updates/pv",
    "media-type": "text/event-stream",
    "uses": [ "my-pv" ],
    "accepts": "application/alto-updatestreamparams+json",
    "capabilities": {
      "incremental-change-media-types": {
        "my-pv": "application/merge-patch+json"
      },
      "support-stream-control": true
    }
  }
```

Note that it is straightforward for an ALTO sever to run HTTP/2 and support concurrent retrieval of multiple resources such as "my-network-map" and "my-routingcost-map" using multiple HTTP/2 streams with the need to introducing ALTO/H2.

The resource "update-my-costs-h2" provides an ALTO/H2 based connection, and this is indicated by the media-type "application/alto-h2". For an ALTO/H2 connection, the client can send in a sequence of control requests using media type application/alto-updatestreamparams+json. The server creates HTTP/2 streams and pushes updates to the client.

10. Security Considerations

The properties defined in this document present no security considerations beyond those in Section 15 of the base ALTO specification [RFC7285].

11. IANA Considerations

IANA will need to register the alto-h2 media type under ALTO registry as defined in [RFC7285].

12. Acknowledgments

The authors of this document would also like to thank many for the reviews and comments.

13. References

13.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[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>.

[RFC7285] Alimi, R., Ed., Penno, R., Ed., Yang, Y., Ed., Kiesel, S., Previdi, S., Roome, W., Shalunov, S., and R. Woundy, "Application-Layer Traffic Optimization (ALTO) Protocol", RFC 7285, DOI 10.17487/RFC7285, September 2014, <https://www.rfc-editor.org/info/rfc7285>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/

RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8895] Roome, W. and Y. Yang, "Application-Layer Traffic Optimization (ALTO) Incremental Updates Using Server-Sent Events (SSE)", RFC 8895, DOI 10.17487/RFC8895, November 2020, <https://www.rfc-editor.org/info/rfc8895>.

13.2. Informative References

[RFC7971] Stiemerling, M., Kiesel, S., Scharf, M., Seidel, H., and S. Previdi, "Application-Layer Traffic Optimization (ALTO) Deployment Considerations", RFC 7971, DOI 10.17487/RFC7971, October 2016, <https://www.rfc-editor.org/info/rfc7971>.

Appendix A. Outlook to ALTO with HTTP/3

This draft is focusing on HTTP/2 enhancement of the ALTO protocol and the design takes advantage of HTTP/2 design features such as parallel transfer and respects HTTP/2 semantics (e.g., PUSH_PROMISE). Since QUIC and HTTP/3 respectively are coming up for various protocols on the Internet it is understandable that the question arises, if ATLO could also take advantage of the advantages of HTTP/3. QUIC can be seen as a replacement for TCP+TLS+HTTP2. HTTP/3 bases on the QUIC transport protocol and uses UDP instead of a TCP connection.

QUIC has been developed by the IETF QUIC Working Group with the following goals:

* Minimizing connection establishment and overall transport latency for applications, starting with HTTP/2
* Providing multiplexing without head-of-line blocking
* Requiring only changes to path endpoints to enable deployment
* Enabling multipath and forward error correction extensions
* Providing always-secure transport, using TLS 1.3 by default

If HTTP/3 is not supported, it automatically runs on HTTP/2. The prerequisite for HTTP/3 is that both client and server support it.

The basic assumption is that an implementation that runs on HTTP/2 should also run-on HTTP/3. This should be transparent. HTTP/3 uses "well known port" UDP 443 analogous to TCP 443. The network between client and server must not filter HTTP/3.

Since many applications still using HTTP/2 it is mandatory for ALTO to support this protocol first. This ensures compatibility. Therefore, this document describes the update of ALTO from HTTP/1.x to HTTP/2. The usage of HTTP/3 will be described in a separate document so that compatibility of ALTO with HTTP/3 will be ensured in a later stage.

Authors' Addresses

Roland Schott
Deutsche Telekom
Heinrich-Hertz-Strasse 3-7
64295 Darmstadt
Germany

Email: Roland.Schott@telekom.de

Y. Richard Yang
Yale University
51 Prospect St
New Haven, CT 06520
United States of America

Email: yry@cs.yale.edu

Kai Gao
Sichuan University
Chengdu
201804
China

Email: kgao@scu.edu.cn

Jingxuan Jensen Zhang
Tongji University
4800 Cao'An Hwy
Shanghai
201804
China

Email: jingxuan.n.zhang@gmail.com