RFC 8895
Application-Layer Traffic Optimization (ALTO) Incremental Updates Using Server-Sent Events (SSE)

Abstract

The Application-Layer Traffic Optimization (ALTO) protocol (RFC 7285) provides network-related information, called network information resources, to client applications so that clients can make informed decisions in utilizing network resources. This document presents a mechanism to allow an ALTO server to push updates to ALTO clients to achieve two benefits: (1) updates can be incremental, in that if only a small section of an information resource changes, the ALTO server can send just the changes and (2) updates can be immediate, in that the ALTO server can send updates as soon as they are available.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8895.

Copyright Notice

Table of Contents

1. Introduction

The Application-Layer Traffic Optimization (ALTO) protocol [RFC7285] provides network-related information, called network information resources, to client applications so that clients may make informed decisions in utilizing network resources. For example, an ALTO server provides network and cost maps, where a network map partitions the set of endpoints into a manageable number of sets each defined by a Provider-Defined Identifier (PID) and a cost map provides directed costs between PIDs. Given network and cost maps, an ALTO client can obtain costs between endpoints by first using the network map to get the PID for each endpoint and then using the cost map to get the costs between those PIDs. Such costs can be used by the client to choose communicating endpoints with low network costs.

The ALTO protocol defines only an ALTO client pull model without defining a mechanism to allow an ALTO client to obtain updates to network information resources, other than by periodically re-fetching them. In settings where an information resource may be large but only parts of it may change frequently (e.g., some entries of a cost map), complete re-fetching can be inefficient.

This document presents a mechanism to allow an ALTO server to push incremental updates to ALTO clients. Integrating server push and incremental updates provides two benefits: (1) updates can be small, in that if only a small section of an information resource changes, the ALTO server can send just the changes and (2) updates can be immediate, in that the ALTO server can send updates as soon as they are available.

While primarily intended to provide updates to GET-mode network and cost maps, the mechanism defined in this document can also provide updates to POST-mode ALTO services, such as the ALTO endpoint property and endpoint cost services. The mechanism can also support new ALTO services to be defined by future extensions, but a future service needs to satisfy requirements specified in Section 11.

The rest of this document is organized as follows. Section 3 gives background on the basic techniques used in this design: (1) JSON merge patch and JSON patch to allow incremental updates and (2) Server-Sent Events (SSE) [SSE] to allow server push. With the background, Section 4 gives a non-normative overview of the design. Section 5 defines individual messages in an update stream. Section 6 defines the update stream service. Section 7 defines the stream control service. Section 8 gives several examples to illustrate the two types of services. Section 9 describes operation and processing considerations by both ALTO servers and clients. Section 13 discusses a design feature that is not supported. Section 10 discusses security issues. Sections 11 and 12 review the requirements for future ALTO services to use SSE and IANA considerations, respectively.

2. Terms

Besides the terminologies as defined in [RFC7285], this document also uses additional terminologies defined as follows:

Update Stream:
    A reliable, in-order connection compatible with HTTP/1.x between an ALTO client and an ALTO server so that the server can push a sequence of update messages using [SSE] to the client.

Update Stream Server:
    This document refers to an ALTO server providing an update stream as an ALTO update stream server, or update stream server for short. Note that the ALTO server mentioned in this document refers to a general server that provides various kinds of services; it can be an update stream server or stream control server (see below). It can also be a server providing ALTO Information Resource Directory (IRD).

Update Message:
    A message that is either a data update message or a control update message.

Data Update Message:
    An update message that is for a single ALTO information resource and sent from the update stream server to the ALTO client when the resource changes. A data update message can be either a full-replacement message or an incremental-change message. Full replacement is a shorthand for a full-replacement message, and incremental change is a shorthand for an incremental-change message.

Full Replacement:
    A data update message for a resource that encodes the content of the resource in its original ALTO encoding.

Incremental Change:
>   A data update message that specifies only the difference between the new content and the previous version. An incremental change can be encoded using either JSON merge patch or JSON patch in this document.

Stream Control Service:
>   A service that provides an HTTP URI so that the ALTO client of an update stream can use it to send stream control requests to the ALTO server on the addition or removal of resources receiving update messages from the update stream. The ALTO server creates a new stream control resource for each update stream instance, assigns a unique URI to it, and sends the URI to the client as the first event in the stream. (Note that the stream control service in ALTO has no association with the similarly named Stream Control Transmission Protocol [RFC4960].)

Stream Control:
>   A shorthand for stream control service.

Stream Control Server:
>   An ALTO server providing the stream control service.

Substream-ID:
>   An ALTO client can assign a unique substream-id when requesting the addition of a resource receiving update messages from an update stream. The server puts the substream-id in each update event for that resource. The substream-id allows a client to use one update stream to receive updates to multiple requests for the same resource (i.e., with the same resource-id in an ALTO IRD), for example, for a POST-mode resource with different input parameters.

Data-ID:
>   A subfield of the "event" field of [SSE] to identify the ALTO data (object) to be updated. For an ALTO resource returning a multipart response, the data-id to identify the data (object) is the substream-id, in addition to the Content-ID of the object in the multipart response. The data-id of a single-part response is just the substream-id.

Control Update Message:
>   An update message for the update stream server to notify the ALTO client of related control information of the update stream. A control update message may be triggered by an internal event at the server, such as server overloading and hence the update stream server will no longer send updates for an information resource, or as a result of a client sending a request through the stream control service. The first message of an update stream is a control update message that provides a control URI to the ALTO client. The ALTO client can use the URI to send stream control requests to the stream control server.

2.1. 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.

3. Background

The design requires two basic techniques: encoding of incremental changes and server push. For incremental changes, existing techniques include JSON merge patch and JSON patch; this design uses both. For server push, existing techniques include HTTP/2 and [SSE]; this design adopts some design features of HTTP/2 but uses [SSE] as the basic server-push design. The rest of this section gives a non-normative summary of JSON merge patch, JSON patch, HTTP/2, and [SSE].

3.1. Incremental Encoding: JSON Merge Patch

To avoid always sending complete data, a server needs mechanisms to encode incremental changes, and JSON merge patch is one mechanism. [RFC7396] defines the encoding of incremental changes (called JSON merge patch objects) to be used by the HTTP PATCH method [RFC5789]. From [RFC7396], this document adopts only the JSON merge patch object encoding and does not use the HTTP PATCH method, as the updates are sent as events instead of HTTP methods; also, the updates are server to client, and PATCH semantics are more for client to server. Below is a non-normative summary of JSON merge patch objects; see [RFC7396] for the normative definition.

3.1.1. JSON Merge Patch Encoding

Informally, a JSON merge patch message consists of a JSON merge patch object (referred to as a patch in [RFC7396]), which defines how to transform one JSON value into another using a recursive merge patch algorithm. Specifically, the patch is computed by treating two JSON values (first one being the original and the second being the updated) as trees of nested JSON objects (dictionaries of name/value pairs), where the leaves are values (e.g., JSON arrays, strings, and numbers), other than JSON objects, and the path for each leaf is the sequence of keys leading to that leaf. When the second tree has a different value for a leaf at a path or adds a new leaf, the patch has a leaf, at that path, with the new value. When a leaf in the first tree does not exist in the second tree, the JSON merge patch tree has a leaf with a JSON "null" value. Hence, in the patch, null as the value of a name/value pair will delete the element with "name" in the original JSON value. The patch does not have an entry for any leaf that has the same value in both versions. See the MergePatch pseudocode at the beginning of Section 2 of [RFC7396] for the formal specification of how to apply a given patch. As a result, if all leaf values are simple scalars, JSON merge patch is a quite efficient representation of incremental changes. It is less efficient when leaf values are arrays, because JSON merge patch replaces arrays in their entirety, even if only one entry changes.

3.1.2. JSON Merge Patch ALTO Messages

To provide both examples of JSON merge patch and a demonstration of the feasibility of applying JSON merge patch to ALTO, the sections below show the application of JSON merge patch to two key ALTO messages.

3.1.2.1. JSON Merge Patch Network Map Messages

Section 11.2.1.6 of [RFC7285] defines the format of an ALTO network map message. Assume a simple example ALTO message sending an initial network map:

```
{
  "meta" : {
    "vtag": {
      "resource-id" : "my-network-map",
      "tag" : "da65eca2eb7a10ce8b059740b0b2e3f8eb1d4785"
    }
  },
  "network-map" : {
    "PID1" : {
      "ipv4" : [ "192.0.2.0/24", "198.51.100.0/25" ]
    },
    "PID2" : {
      "ipv4" : [ "198.51.100.128/25" ]
    },
    "PID3" : {
      "ipv4" : [ "0.0.0.0/0" ],
      "ipv6" : [ "::/0" ]
    }
  }
}
```

Consider the following JSON merge patch update message, which (1) adds an ipv4 prefix "203.0.113.0/25" and an ipv6 prefix "2001:db8:8000::/33" to "PID1", (2) deletes "PID2", and (3) assigns a new "tag" to the network map:

```
{
  "meta" : {
    "vtag" : {
      "tag" : "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
    }
  },
  "network-map": {
    "PID1" : {
      "ipv4" : [ "192.0.2.0/24", "198.51.100.0/25",
                 "203.0.113.0/25" ],
      "ipv6" : [ "2001:db8:8000::/33" ]
    },
    "PID2" : null
  }
}
```

Applying the JSON merge patch update to the initial network map is equivalent to the following ALTO network map:

```
{
  "meta" : {
    "vtag": {
      "resource-id" : "my-network-map",
      "tag" : "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
    }
  },
  "network-map" : {
    "PID1" : {
      "ipv4" : [ "192.0.2.0/24", "198.51.100.0/25",
                 "203.0.113.0/25" ],
      "ipv6" : [ "2001:db8:8000::/33" ]
    },
    "PID3" : {
      "ipv4" : [ "0.0.0.0/0" ],
      "ipv6" : [ "::/0" ]
    }
  }
}
```

3.1.2.2. JSON Merge Patch Cost Map Messages

Section 11.2.3.6 of [RFC7285] defines the format of an ALTO cost map message. Assume a simple example ALTO message for an initial cost map:

```
{
  "meta" : {
    "dependent-vtags" : [
      {"resource-id": "my-network-map",
       "tag": "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
      }
    ],
    "cost-type" : {
      "cost-mode"   : "numerical",
      "cost-metric": "routingcost"
    },
    "vtag": {
      "resource-id" : "my-cost-map",
      "tag" : "3ee2cb7e8d63d9fab71b9b34cbf764436315542e"
    }
  },
  "cost-map" : {
    "PID1": { "PID1": 1,   "PID2": 5,   "PID3": 10 },
    "PID2": { "PID1": 5,   "PID2": 1,   "PID3": 15 },
    "PID3": { "PID1": 20, "PID2": 15   }
  }
}
```

The following JSON merge patch message updates the example cost map so that (1) the "tag" field of the cost map is updated, (2) the cost of PID1->PID2 is 9 instead of 5, (3) the cost of PID3->PID1 is no longer available, and (4) the cost of PID3->PID3 is defined as 1.

```
{
  "meta" : {
    "vtag": {
      "tag": "c0ce023b8678a7b9ec00324673b98e54656d1f6d"
    }
  }
  "cost-map" : {
    "PID1" : { "PID2" : 9 },
    "PID3" : { "PID1" : null, "PID3" : 1 }
  }
}
```

Hence, applying the JSON merge patch to the initial cost map is equivalent to the following ALTO cost map:

```
{
  "meta" : {
    "dependent-vtags" : [
      {"resource-id": "my-network-map",
       "tag": "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
      }
    ],
    "cost-type" : {
      "cost-mode"  : "numerical",
      "cost-metric": "routingcost"
    },
    "vtag": {
      "resource-id": "my-cost-map",
      "tag": "c0ce023b8678a7b9ec00324673b98e54656d1f6d"
    }
  },
  "cost-map" : {
    "PID1": { "PID1": 1,   "PID2": 9,   "PID3": 10 },
    "PID2": { "PID1": 5,   "PID2": 1,   "PID3": 15 },
    "PID3": {              "PID2": 15, "PID3": 1   }
  }
}
```

3.2. Incremental Encoding: JSON Patch

3.2.1. JSON Patch Encoding

One issue of JSON merge patch is that it does not handle array changes well. In particular, JSON merge patch considers an array as a single object and hence can only replace an array in its entirety. When the change is to make a small change to an array, such as the deletion of an element from a large array, whole-array replacement is inefficient. Consider the example in Section 3.1.2.1. To add a new entry to the ipv4 array for PID1, the server needs to send a whole

new array. Another issue is that JSON merge patch cannot change a value to be null, as the JSON merge patch processing algorithm (MergePatch in Section 3.1.1) interprets a null as a removal instruction. On the other hand, some ALTO resources can have null values, and it is possible that the update will want to change the new value to be null.

JSON patch [RFC6902] can address the preceding issues. It defines a set of operators to modify a JSON object. See [RFC6902] for the normative definition.

3.2.2. JSON Patch ALTO Messages

To provide both examples of JSON patch and a demonstration of the difference between JSON patch and JSON merge patch, the sections below show the application of JSON patch to the same updates shown in Section 3.1.2.

3.2.2.1. JSON Patch Network Map Messages

First, consider the same update as in Section 3.1.2.1 for the network map. Below is the encoding using JSON patch:

```
[
  {
    "op": "replace",
    "path": "/meta/vtag/tag",
    "value": "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
  },
  {
    "op": "add",
    "path": "/network-map/PID1/ipv4/2",
    "value": "203.0.113.0/25"
  }
  {
    "op": "add",
    "path": "/network-map/PID1/ipv6",
    "value": ["2001:db8:8000::/33"]
  },
  {
    "op": "remove",
    "path": "/network-map/PID2"
  }
]
```

3.2.2.2. JSON Patch Cost Map Messages

Compared with JSON merge patch, JSON patch does not encode cost map updates efficiently. Consider the cost map update shown in Section 3.1.2.2, the encoding using JSON patch is:

```
[
  {
    "op": "replace",
    "path": "/meta/vtag/tag",
    "value": "c0ce023b8678a7b9ec00324673b98e54656d1f6d"
  },
  {
    "op": "replace",
    "path": "/cost-map/PID1/PID2",
    "value": 9
  },
  {
    "op": "remove",
    "path": "/cost-map/PID3/PID1"
  },
  {
    "op": "replace",
    "path": "/cost-map/PID3/PID3",
    "value": 1
  }
]
```

3.3. Multiplexing and Server Push: HTTP/2

HTTP/2 [RFC7540] provides two related features: multiplexing and server push. In particular, HTTP/2 allows a client and a server to multiplex multiple HTTP requests and responses over a single TCP connection. The requests and responses can be interleaved on a block (frame) by block (frame) basis, by indicating the requests and responses in HTTP/2 messages, avoiding the head-of-line blocking problem encountered with HTTP/1.1. To achieve the same goal, this design introduces substream-id to allow a client to receive updates to multiple resources. HTTP/2 also provides a server-push facility to allow a server to send asynchronous updates.

Despite the two features of HTTP/2, this design chooses a design compatible with HTTP/1.x for the simplicity of HTTP/1.x. A design based on HTTP/2 may more likely need to be implemented using a more complex HTTP/2 client library. In such a case, one approach for using server push for updates is for the update stream server to send each data update message as a separate server-push item and let the client apply those updates as they arrive. An HTTP/2 client library may not necessarily inform a client application when the server pushes a resource. Instead, the library might cache the pushed resource and only deliver it to the client when the client explicitly requests that URI. Further, it is more likely that a design based on HTTP/2 may encounter issues with a proxy between the client and the server, in that server push is optional and can be disabled by any proxy between the client and the server. This is not a problem for the intended use of server push; eventually, the client will request those resources, so disabling server push just adds a delay. But this means that Server Push is not suitable for resources that the client does not know to request.

Thus, this design leaves a design based on HTTP/2 as a future work and focuses on ALTO updates on HTTP/1.x and [SSE].

3.4. Server Push: Server-Sent Event

Server-Sent Events (SSE) are techniques that can work with HTTP/1.1. The following is a non-normative summary of SSE; see [SSE] for its normative definition.

SSE enable a server to send new data to a client by "server push". The client establishes an HTTP [RFC7230] [RFC7231] connection to the server and keeps the connection open. The server continually sends messages. Each message has one or more lines, where a line is terminated by a carriage return immediately followed by a new line, a carriage return not immediately followed by a new line, or a new line not immediately preceded by a carriage return. A message is terminated by a blank line (two line terminators in a row).

Each line in a message is of the form "field-name: string value". Lines with a blank field name (that is, lines that start with a colon) are ignored, as are lines that do not have a colon. The protocol defines three field names: event, id, and data. If a message has more than one "data" line, the value of the data field is the concatenation of the values on those lines. There can be only one "event" and "id" line per message. The "data" field is required; the others are optional.

Figure 1 is a sample SSE stream, starting with the client request. The server sends three events and then closes the stream.

```
(Client request)
GET /stream HTTP/1.1
Host: example.com
Accept: text/event-stream

(Server response)
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream

event: start
id: 1
data: hello there

event: middle
id: 2
data: let's chat some more ...
data: and more and more and ...

event: end
id: 3
data: goodbye
```

Figure 1: A Sample SSE Stream

4. Overview of Approach and High-Level Protocol Message Flow

With the preceding background, this section now gives a non-normative overview of the update mechanisms and message flow to be defined in later sections of this document. Figure 2 gives the main components and overall message flow.
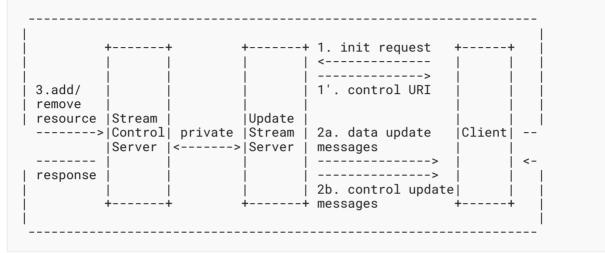
```
  --------------------------------------------------------------------
 |                                                                    |
 |           +-------+            +-------+ 1. init request   +------+ |
 |           |       |            |       | |  <-------------  |      | |
 |           |       |            |       | |  -------------> |      | |
 | 3.add/    |       |            |       | 1'. control URI   |      | |
 | remove    |       |            |       | |                 |      | |
 | resource  |Stream |            |Update |                   |      | |
 |  ------->|Control| private  |Update | 2a. data update     |Client| --
 |          |Server |<------->|Stream | messages              |      | |
 |  --------  |       |            |Server | ---------------> |      | | <-
 | response  |       |            |       | ---------------> |      | |
 |           |       |            |       | | 2b. control update|    | |
 |           +-------+            +-------+ messages          +------+ |
 |                                                                    |
  --------------------------------------------------------------------
```

Figure 2: ALTO SSE Architecture and Message Flow

4.1. Update Stream Service Message Flow

The building block of the update mechanism defined in this document is the update stream service (defined in Section 6), where each update stream service is a POST-mode service that provides update streams.

Note that the lines of the format "** ... **" are used to describe message flows in this section and the following sections.

** Initial request: client -> update server **:
   When an ALTO client requests an update stream service, the ALTO client establishes a persistent connection to the update stream server and submits an initial update-stream request (defined in Section 6.5), creating an update stream. This initial request creating the update stream is labeled "1. init request" in Figure 2.

   An update stream can provide updates to both GET-mode resources, such as ALTO network and cost maps, and POST-mode resources, such as ALTO endpoint property service. Also, to avoid creating too many update streams, this design allows an ALTO client to use one update stream to receive updates to multiple requests. In particular, the client may request to receive updates for the same resource but with different parameters for a POST-mode resource, in addition to being able to consolidate updates for multiple resources into a single stream. The

updates for each request is called a substream and hence the update server needs an identifier to indicate the substream when sending an update. To achieve this goal, the client assigns a unique substream-id when requesting updates to a resource in an update stream, and the server puts the substream-id in each update.

** Data updates: update server -> client **:
    The objective of an update stream is to continuously push (to an ALTO client) the data value changes for a set of resources, where the set of resources is specified by the ALTO client's requests. This document refers to messages sending such data-value changes as data update messages (defined in Section 5.2). Although an update stream may update one or more requests, each data update message updates only one request and is sent as a Server-Sent Event (SSE), as defined by [SSE]. A data update message is encoded either as a full replacement or as an incremental change. A full replacement uses the JSON message format defined by the ALTO protocol. There can be multiple encodings for incremental changes. The current design supports incremental changes using JSON merge patch [RFC7396] or JSON patch [RFC6902] to describe the changes of the resource. Future documents may define additional mechanisms for incremental changes. The update stream server decides when to send data update messages and whether to send full replacements or incremental changes. These decisions can vary from resource to resource and from update to update. Since the transport is a design compatible with HTTP/1.x, data update messages are delivered reliably and in order, and the lossless, sequential delivery of its messages allows the server to know the exact state of the client to compute the correct incremental updates. Figure 2 shows examples of data update messages (labeled "2a. data update messages") in the overall message flow.

** Control updates: update server -> client **:
    An update stream can run for a long time and hence there can be status changes at the update stream server side during the lifetime of an update stream; for example, the update stream server may encounter an error or need to shut down for maintenance. To support a robust, flexible protocol design, this document allows the update stream server to send control update messages (defined in Section 5.3) in addition to data update messages to the ALTO client. Figure 2 shows that both data updates and control updates can be sent by the server to the client (labeled "2b. control update messages").

4.2. Stream Control Service Message Flow

** Stream control: client -> stream control server **:
    In addition to control changes triggered from the update stream server side, in a flexible design, an ALTO client may initiate control changes as well, in particular, by adding or removing ALTO resources receiving updates. An ALTO client initiates such changes using the stream control service (defined in Section 7). Although one may use a design that the client uses as the same HTTP connection to send the control requests, it requires stronger server support, such as HTTP pipeline. For more flexibility, this document introduces stream control service. In particular, the update stream server of an update stream uses the first message to provide the URI of the stream control service (labeled "1': control URI" in Figure 2).

The ALTO client can then use the URI to ask the stream control server specified in the URI to request the update stream server to (1) send data update messages for additional resources, (2) stop sending data update messages for previously requested resources, or (3) gracefully stop and close the update stream altogether.

4.3. Service Announcement and Management Message Flow

** Service announcements: IRD server -> client **:
An update server may provide any number of update stream services, where each update stream may provide updates for a given subset of the ALTO server's resources. An ALTO server's Information Resource Directory (IRD) defines the update stream services and declares the set of resources for which each update stream service provides updates. The ALTO server selects the resource set for each update stream service. It is recommended that if a resource depends on one or more other resource(s) (indicated with the "uses" attribute defined in [RFC7285]), these other resource(s) should also be part of that update stream. Thus, the update stream for a cost map should also provide updates for the network map on which that cost map depends.

** Service management (server) **:
An ALTO client may request any number of update streams simultaneously. Because each update stream consumes resources on the update stream server, an update stream server may require client authorization and/or authentication, limit the number of open update streams, close inactive streams, or redirect an ALTO client to another update stream server.

5. Update Messages: Data Update and Control Update Messages

This section defines the format of update messages sent from the server to the client. It first defines the generic structure of update messages (Section 5.1). It then defines the details of the data update messages (Section 5.2) and the control update messages (Section 5.3). These messages will be used in the next two sections to define the update stream service (Section 6) and the stream control service (Section 7).

5.1. Generic ALTO Update Message Structure

Both data update and control update messages from the server to the client have the same basic structure. Each message includes a data field to provide data information, which is typically a JSON object, and an event field preceding the data field, to specify the media type indicating the encoding of the data field.

A data update message needs additional information to identify the ALTO data (object) to which the update message applies. To be generic, this document uses a data-id to identify the ALTO data (object) to be updated; see below.

Hence, the event field of ALTO update message can include two subfields (media-type and data-id), where the two subfields are separated by a comma (',', U+002C):

```
media-type [ ',' data-id ]
```

According to Section 4.2 of [RFC6838], the comma character is not allowed in a media-type name so there is no ambiguity when decoding of the two subfields.

Note that an update message does not use the SSE "id" field.

5.2. ALTO Data Update Message

A data update message is sent when a monitored resource changes. As discussed in the preceding section, the event field of a data update message includes two subfields: 'media-type' and 'data-id'.

The 'media-type' subfield depends on whether the data update is a complete specification of the identified data or an incremental patch (e.g., a JSON merge patch or JSON patch), if possible, describing the changes from the last version of the data. This document refers to these as full replacement and incremental change, respectively. The encoding of a full replacement is defined by its defining document (e.g., network and cost map messages by [RFC7285]) and uses the media type defined in that document. The encoding of JSON merge patch is defined by [RFC7396], with the media type "application/merge-patch+json"; the encoding of JSON patch is defined by [RFC6902], with media type "application/json-patch+json".

The 'data-id' subfield identifies the ALTO data to which the data update message applies.

First, consider the case that the resource contains only a single JSON object. For example, since an ALTO client can request data updates for both a cost map resource (object) and its dependent network map resource (object) in the same update stream, to distinguish the updates, the client assigns a substream-id for each resource receiving data updates. Substream-ids MUST be unique within an update stream but need not be globally unique. A substream-id is encoded as a JSON string with the same format as that of the type ResourceID (Section 10.2 of [RFC7285]). The type SubstreamID is used in this document to indicate a string of this format. The substream-id of a single JSON object is the 'data-id'.

As an example, assume that the ALTO client assigns substream-id "1" in its request to receive updates to the network map and substream-id "2" to the cost map. Then, the substream-ids are the data-ids indicating which objects will be updated. Figure 3 shows some examples of ALTO data update messages:

```
event: application/alto-networkmap+json,1
data: { ... full network map message ... }

event: application/alto-costmap+json,2
data: { ... full cost map message ... }

event: application/merge-patch+json,2
data: { ... JSON merge patch update for the cost map ... }
```

Figure 3: Examples of ALTO Data Update Messages

Next, consider the case that a resource may include multiple JSON objects. This document considers the case that a resource may contain multiple components (parts), and they are encoded using the media type "multipart/related" [RFC2387]. Each part of this multipart response **MUST** be an HTTP message including a Content-ID header and a JSON object body. Each component requiring the update stream service (defined in Section 6) **MUST** be identified by a unique Content-ID to be defined in its defining document.

For a resource using the media type "multipart/related", the 'data-id' subfield **MUST** be the concatenation of the substream-id, the '.' separator (U+002E), and the unique Content-ID, in order.

5.3. ALTO Control Update Message

Control update messages have the media type "application/alto-updatestreamcontrol+json", and the data is of type UpdateStreamControlEvent:

```
object {
    [String          control-uri;]
    [SubstreamID     started<1..*>;]
    [SubstreamID     stopped<1..*>;]
    [String          description;]
} UpdateStreamControlEvent;
```

control-uri:
    the URI providing stream control for this update stream (see Section 7). The server sends a control update message notifying the client of the control-uri. This control update message notifying the control-uri will be sent once and **MUST** be the first event in an update stream. If the URI value is NULL, the update stream server does not support stream control for this update stream; otherwise, the update stream server provides stream control through the given URI.

started:
    a list of substream-ids of resources. It notifies the ALTO client that the update stream server will start sending data update messages for each resource listed.

stopped:
> a list of substream-ids of resources. It notifies the ALTO client that the update stream server will no longer send data update messages for the listed resources. There can be multiple reasons for an update stream server to stop sending data update messages for a resource, including a request from the ALTO client using stream control (Section 6.7.1) or an internal server event.

description:
> a non-normative, human-readable text providing an explanation for the control event. When an update stream server stops sending data update messages for a resource, it is **RECOMMENDED** that the update stream server use the description field to provide details. There can be multiple reasons that trigger a "stopped" event; see above. The intention of this field is to provide a human-readable text for the developer and/or the administrator to diagnose potential problems.

6. Update Stream Service

An update stream service returns a stream of update messages, as defined in Section 5. An ALTO server's IRD (Information Resource Directory) **MAY** define one or more update stream services, which ALTO clients use to request new update stream instances. An IRD entry defining an update stream service **MUST** define the media type, HTTP method, and capabilities and uses as follows.

6.1. Media Type

The media type of an ALTO update stream service is "text/event-stream", as defined by [SSE].

6.2. HTTP Method

An ALTO update stream service is requested using the HTTP POST method.

6.3. Capabilities

The capabilities are defined as an object of type UpdateStreamCapabilities:

```
object {
  IncrementalUpdateMediaTypes incremental-change-media-types;
  Boolean                     support-stream-control;
} UpdateStreamCapabilities;

object-map {
   ResourceID -> String;
} IncrementalUpdateMediaTypes;
```

If this update stream can provide data update messages with incremental changes for a resource, the "incremental-change-media-types" field has an entry for that resource-id, and the value is the supported media types of the incremental change separated by commas. Normally, this will be "application/merge-patch+json", "application/json-patch+json", or "application/merge-patch

+json,application/json-patch+json", because, as described in Section 5, they are the only incremental change types defined by this document. However, future extensions may define other types of incremental changes.

When choosing the media types to encode incremental changes for a resource, the update stream server **MUST** consider the limitations of the encoding. For example, when a JSON merge patch specifies that the value of a field is null, its semantics are that the field is removed from the target and hence the field is no longer defined (i.e., undefined); see the MergePatch algorithm in Section 3.1.1 on how null value is processed. This, however, may not be the intended result for the resource, when null and undefined have different semantics for the resource. In such a case, the update stream server **MUST** choose JSON patch over JSON merge patch if JSON patch is indicated as a capability of the update stream server. If the server does not support JSON patch to handle such a case, the server then need to send a full replacement.

The "support-stream-control" field specifies whether the given update stream supports stream control. If the "support-stream-control" field is "true", the update stream server will use the stream control specified in this document; otherwise, the update stream server may use other mechanisms to provide the same functionality as stream control.

6.4. Uses

The "uses" attribute **MUST** be an array with the resource-ids of every resource for which this update stream can provide updates. Each resource specified in the "uses" **MUST** support full replacement; the update stream server can always send full replacement, and the ALTO client **MUST** accept full replacement.

This set may be any subset of the ALTO server's resources and may include resources defined in linked IRDs. However, it is **RECOMMENDED** that the ALTO server selects a set that is closed under the resource dependency relationship. That is, if an update stream's "uses" set includes resource R1 and resource R1 depends on ("uses") resource R0, then the update stream's "uses" set **SHOULD** include R0 as well as R1. For example, an update stream for a cost map **SHOULD** also provide updates for the network map upon which that cost map depends.

6.5. Request: Accept Input Parameters

An ALTO client specifies the parameters for the new update stream by sending an HTTP POST body with the media type "application/alto-updatestreamparams+json". That body contains a JSON object of type UpdateStreamReq, where:

```
object {
    [AddUpdatesReq    add;]
    [SubstreamID      remove<0..*>;]
} UpdateStreamReq;

object-map {
    SubstreamID -> AddUpdateReq;
} AddUpdatesReq;

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

add:
> Specifies the resources (and the parameters for the resources) for which the ALTO client wants updates. In the scope of the same update stream, the ALTO client **MUST** assign a substream-id that is unique in the scope of the update stream (Section 5.2) for each entry and use those substream-ids as the keys in the "add" field.

resource-id:
> The resource-id of an ALTO resource and **MUST** be in the update stream's "uses" list (Section 6.4). If the resource-id is a GET-mode resource with a version tag (or "vtag"), as defined in Sections 6.3 and 10.3 of [RFC7285], and the ALTO client has previously retrieved a version of that resource from the update stream server, the ALTO client **MAY** set the "tag" field to the tag part of the client's version of that resource. If that version is not current, the update stream server **MUST** send a full replacement before sending any incremental changes, as described in Section 6.7.1. If that version is still current, the update stream server **MAY** omit the initial full replacement.

incremental-changes:
> The ALTO client specifies whether it is willing to receive incremental changes from the update stream server for this substream. If the "incremental-changes" field is "true", the update stream server **MAY** send incremental changes for this substream. In this case, the client **MUST** support all incremental methods from the set announced in the server's capabilities for this resource; see Section 6.3 for the server's announcement of potential incremental methods. If a client does not support all incremental methods from the set announced in the server's capabilities, the client can set "incremental-changes" to "false", and the update stream server then **MUST NOT** send incremental changes for that substream. The default value for

"incremental-changes" is "true", so to suppress incremental changes, the ALTO client **MUST** explicitly set "incremental-changes" to "false". An alternative design of incremental-changes control is a more fine-grained control, by allowing a client to select a subset of incremental methods from the set announced in the server's capabilities. But this alternative design is not adopted in this document, because it adds complexity to the server, which is more likely to be the bottleneck. Note that the ALTO client cannot suppress full replacement. When the ALTO client sets "incremental-changes" to "false", the update stream server **MUST** send a full replacement instead of an incremental change to the ALTO client. The update stream server **MAY** wait until more changes are available and send a single full replacement with those changes. Thus, an ALTO client that declines to accept incremental changes may not get updates as quickly as an ALTO client that does.

input:
> If the resource is a POST-mode service that requires input, the ALTO client **MUST** set the "input" field to a JSON object with the parameters that the resource expects.

remove:
> It is used in update stream control requests (Section 7) and is not allowed in the update stream request. The update stream server **SHOULD** ignore this field if it is included in the request.

If a request has any errors, the update stream server **MUST NOT** create an update stream. Also, the update stream server will send an error response to the ALTO client, as specified in Section 6.6.

6.6. Response

If the update stream request has any errors, the update stream server **MUST** return an HTTP "400 Bad Request" to the ALTO client; the body of the response follows the generic ALTO error response format specified in Section 8.5.2 of [RFC7285]. Hence, an example ALTO error response has the format:

```
HTTP/1.1 400 Bad Request
Content-Length: 131
Content-Type: application/alto-error+json
Connection: Closed

{
    "meta":{
        "code":  "E_INVALID_FIELD_VALUE",
        "field": "add/my-network-map/resource-id",
        "value": "my-networkmap/#"
    }
}
```

Note that "field" and "value" are optional fields. If the "value" field exists, the "field" field **MUST** exist.

- If an update stream request does not have an "add" field specifying one or more resources, the error code of the error message **MUST** be E_MISSING_FIELD and the "field" field **SHOULD** be "add". The update stream server **MUST** close the stream without sending any events.
- If the "resource-id" field is invalid or is not associated with the update stream, the error code of the error message **MUST** be E_INVALID_FIELD_VALUE. The "field" field **SHOULD** be the full path of the "resource-id" field, and the "value" field **SHOULD** be the invalid resource-id. If there are more than one invalid resource-ids, the update stream server **SHOULD** pick one and return it. The update stream server **MUST** close the stream (i.e., TCP connection) without sending any events.
- If the resource is a POST-mode service that requires input, the client **MUST** set the "input" field to a JSON object with the parameters that that resource expects. If the "input" field is missing or invalid, the update stream server **MUST** return the same error response that that resource would return for missing or invalid input (see [RFC7285]). In this case, the update stream server **MUST** close the update stream without sending any events. If the input for several POST-mode resources is missing or invalid, the update stream server **MUST** pick one and return it.

The response to a valid request is a stream of update messages. Section 5 defines the update messages, and [SSE] defines how they are encoded into a stream.

An update stream server **SHOULD** send updates only when the underlying values change. However, it may be difficult for an update stream server to guarantee that in all circumstances. Therefore, a client **MUST NOT** assume that an update message represents an actual change.

6.7. Additional Requirements on Update Stream Service

6.7.1. Event Sequence Requirements

- The first event **MUST** be a control update message with the URI of the update stream control service (see Section 7) for this update stream. Note that the value of the control-uri can be "null", indicating that there is no control stream service.
- As soon as possible, after the ALTO client initiates the connection, the update stream server checks the "tag" field for each added update request. If the "tag" field is not specified in an added update request, the update stream server **MUST** first send a full replacement for the request. If the "tag" field is specified, the client can accept incremental changes, and the server can compute an incremental update based on the "tag" (the server needs to ensure that for a POST resource with input, the "tag" should indicate the correct result for different inputs); the update stream server **MAY** omit the initial full replacement.
- If this update stream provides updates for resource-ids R0 and R1 and if R1 depends on R0, then the update stream server **MUST** send the update for R0 before sending the related updates for R1. For example, suppose an update stream provides updates to a network map and its dependent cost maps. When the network map changes, the update stream server **MUST** send the network map update before sending the cost map updates.

- When the ALTO client uses the stream control service to stop updates for one or more resources (Section 7), the ALTO client **MUST** send a stream control request. The update stream server **MUST** send a control update message whose "stopped" field has the substream-ids of all stopped resources.

6.7.2. Cross-Stream Consistency Requirements

If multiple ALTO clients create multiple update streams from the same update stream resource and with the same update request parameters (i.e., same resource and same input), the update stream server **MUST** send the same updates to all of them. However, the update stream server **MAY** pack data items into different patch events, as long as the net result of applying those updates is the same.

For example, suppose two different ALTO clients create two different update streams for the same cost map, and suppose the update stream server processes three separate cost point updates with a brief pause between each update. The server **MUST** send all three new cost points to both clients. But the update stream server **MAY** send a single patch event (with all three cost points) to one ALTO client while sending three separate patch events (with one cost point per event) to the other ALTO client.

An update stream server **MAY** offer several different update stream resources that provide updates to the same underlying resource (that is, a resource-id may appear in the "uses" field of more than one update stream resource). In this case, those update stream resources **MUST** return the same update.

6.7.3. Multipart Update Requirements

This design allows any valid media type for full replacement. Hence, it supports ALTO resources using multipart to contain multiple JSON objects. This realizes the push benefit but not the incremental encoding benefit of SSE.

JSON patch and merge patch provide the incremental encoding benefit but can be applied to only a single JSON object. If an update stream service supports a resource providing a multipart media type, which we refer to as a multipart resource, then the update stream service needs to handle the issue that the message of a full multipart resource can include multiple JSON objects. To address the issue, when an update stream service specifies that it supports JSON patch or merge patch incremental updates for a multipart resource, the service **MUST** ensure that (1) each part of a multipart message is a single JSON object, (2) each part is specified by a static Content-ID in the initial full message, (3) each data update event applies to only one part, and (4) each data update specifies substream-id.content-id as the "event" field of the event, to identify the part to be updated.

6.8. Keep-Alive Messages

In an SSE stream, any line that starts with a colon (U+003A) character is a comment, and an ALTO client **MUST** ignore that line [SSE]. As recommended in [SSE], an update stream server **SHOULD** send a comment line (or an event) every 15 seconds to prevent ALTO clients and proxy servers from dropping the HTTP connection. Note that although TCP also provides a Keep-Alive function,

the interval between TCP Keep-Alive messages can depend on the OS configuration and varies. The preceding recommended SSE Keep-Alive allows the SSE client to detect the status of the update stream server with more certainty.

7. Stream Control Service

A stream control service allows an ALTO client to remove resources from the set of resources that are monitored by an update stream or add additional resources to that set. The service also allows an ALTO client to gracefully shut down an update stream.

When an update stream server creates a new update stream and if the update stream server supports stream control for the update stream, the update stream server creates a stream control service for that update stream. An ALTO client uses the stream control service to remove resources from the update stream instance or to request updates for additional resources. An ALTO client cannot obtain the stream control service through the IRD. Instead, the first event that the update stream server sends to the ALTO client has the URI for the associated stream control service (see Section 5.3).

Each stream control request is an individual HTTP request. The ALTO client **MAY** send multiple stream control requests to the stream control server using the same HTTP connection.

7.1. URI

The URI for a stream control service, by itself, **MUST** uniquely specify the update stream instance that it controls. The stream control server **MUST NOT** use other properties of an HTTP request, such as cookies or the client's IP address, to determine the update stream. Furthermore, an update stream server **MUST NOT** reuse a control service URI once the associated update stream has been closed.

The ALTO client **MUST** evaluate a relative control URI reference [RFC3986] (for example, a URI reference without a host or with a relative path) in the context of the URI used to create the update stream. The stream control service's host **MAY** be different from the update stream's host.

It is expected that there is an internal mechanism to map a stream control URI to the unique update stream instance to be controlled. For example, the update stream service may assign a unique, internal stream id to each update stream instance. However, the exact mechanism is left to the update stream service provider.

To prevent an attacker from forging a stream control URI and sending bogus requests to disrupt other update streams, the service should consider two security issues. First, if http, not https, is used, the stream control URI can be exposed to an on-path attacker. To address this issue, in a setting where the path from the server to the client can traverse such an attacker, the server **SHOULD** use https. Second, even without direct exposure, an off-path attacker may guess valid stream control URIs. To address this issue, the server **SHOULD** choose stream control URIs with enough randomness to make guessing difficult; the server **SHOULD** introduce mechanisms that

detect repeated guesses indicating an attack (e.g., keeping track of the number of failed stream control attempts). Please see the W3C's "Good Practices for Capability URLs" <https://www.w3.org/TR/capability-urls/>.

7.2. Media Type

An ALTO stream control response does not have a specific media type.

7.3. HTTP Method

An ALTO update stream control resource is requested using the HTTP POST method.

7.4. IRD Capabilities & Uses

None (Stream control services do not appear in the IRD).

7.5. Request: Accept Input Parameters

A stream control service accepts the same input media type and input parameters as the update stream service (Section 6.5). The only difference is that a stream control service also accepts the "remove" field.

If specified, the "remove" field is an array of substream-ids the ALTO client previously added to this update stream. An empty "remove" array is equivalent to a list of all currently active resources; the update stream server responds by removing all resources and closing the stream.

An ALTO client **MAY** use the "add" field to add additional resources. The ALTO client **MUST** assign a unique substream-id to each additional resource. Substream-ids **MUST** be unique over the lifetime of this update stream; an ALTO client **MUST NOT** reuse a previously removed substream-id. The processing of an "add" resource is the same as discussed in Sections 6.5 and 6.6.

If a request has any errors, the update stream server **MUST NOT** add or remove any resources from the associated update stream. Also, the stream control server will return an error response to the client, as specified in Section 7.6.

7.6. Response

The stream control server **MUST** process the "add" field before the "remove" field. If the request removes all active resources without adding any additional resources, the update stream server **MUST** close the update stream. Thus, an update stream cannot have zero resources.

If the request has any errors, the stream control server **MUST** return an HTTP "400 Bad Request" to the ALTO client. The body part of the response follows the generic ALTO error response format specified in Section 8.5.2 of [RFC7285]. An error response has the same format as specified in Section 6.6. Detailed error code and error information are specified as below.

- If the "add" request does not satisfy the requirements in Section 6.5, the stream control server **MUST** return the ALTO error message defined in Section 6.6.

- If any substream-id in the "remove" field was not added in a prior request, the error code of the error message **MUST** be E_INVALID_FIELD_VALUE, the "field" field **SHOULD** be "remove", and the "value" field **SHOULD** be an array of the invalid substream-ids. Thus, it is illegal to "add" and "remove" the same substream-id in the same request. However, it is legal to remove a substream-id twice. To support the preceding checking, the update stream server **MUST** keep track of previously used but now closed substream-ids.

- If any substream-id in the "add" field has been used before in this stream, the error code of the error message **MUST** be E_INVALID_FIELD_VALUE, the "field" field **SHOULD** be "add", and the "value" field **SHOULD** be an array of invalid substream-ids.

- If the request has a non-empty "add" field and a "remove" field with an empty list of substream-ids (to replace all active resources with a new set, the client **MUST** explicitly enumerate the substream-ids to be removed), the error code of the error message **MUST** be E_INVALID_FIELD_VALUE, the "field" field **SHOULD** be "remove", and the "value" field **SHOULD** be an empty array.

If the request is valid but the associated update stream has been closed, then the stream control server **MUST** return an HTTP "404 Not Found".

If the request is valid and the stream control server successfully processes the request without error, the stream control server should return either an HTTP "202 Accepted" response or an HTTP "204 No Content" response. The difference is that for the latter case, the stream control server is sure that the update stream server has also processed the request. Regardless of a 202 or 204 HTTP response, the final updates of related resources will be notified by the update stream server using its control update message(s), due to the modular design.

8. Examples

8.1. Example: IRD Announcing Update Stream Services

Below is an example IRD announcing three update stream services. The first, which is named "update-my-costs", provides updates for the network map, the "routingcost" and "hopcount" cost maps, and a Filtered Cost Map resource. The second, which is named "update-my-prop", provides updates to the endpoint properties service. The third, which is named "update-my-pv", provides updates to a nonstandard ALTO service returning a multipart response.

Note that in the "update-my-costs" update stream shown in the example IRD, the update stream server uses JSON patch for network map, and it uses JSON merge patch to update the other resources. Also, the update stream will only provide full replacements for "my-simple-filtered-cost-map".

Also, note that this IRD defines two Filtered Cost Map resources. They use the same cost types, but "my-filtered-cost-map" accepts cost constraint tests, while "my-simple-filtered-cost-map" does not. To avoid the issues discussed in Section 9.3, the update stream provides updates for the second but not the first.

This IRD also announces a nonstandard ALTO service, which is named "my-pv". This service accepts an extended endpoint cost request as an input and returns a multipart response, including an endpoint cost resource and a property map resource. This document does not rely

on any other design details of this new service. In this document, the "my-pv" service is only used to illustrate how the update stream service provides updates to an ALTO resource returning a multipart response.

```
   "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-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
        }
      }
    }
```

8.2. Example: Simple Network and Cost Map Updates

Given the update streams announced in the preceding example IRD, the section below shows an example of an ALTO client's request and the update stream server's immediate response, using the update stream resource "update-my-costs". In the example, the ALTO client requests updates for the network map and "routingcost" cost map but not for the "hopcount" cost map. The ALTO client uses the ALTO server's resource-ids as the substream-ids. Because the client does not provide a "tag" for the network map, the update stream server must send a full replacement for

the network map as well as for the cost map. The ALTO client does not set "incremental-changes" to "false", so it defaults to "true". Thus, the update stream server will send patch updates for the cost map and the network map.

```
POST /updates/costs HTTP/1.1
Host: alto.example.com
Accept: text/event-stream,application/alto-error+json
Content-Type: application/alto-updatestreamparams+json
Content-Length: 155

{ "add": {
    "my-network-map": {
      "resource-id": "my-network-map"
      },
    "my-routingcost-map": {
      "resource-id": "my-routingcost-map"
    }
  }
}
```

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream

event: application/alto-updatestreamcontrol+json
data: {"control-uri":
data: "https://alto.example.com/updates/streams/3141592653589"}

event: application/alto-networkmap+json,my-network-map
data: {
data:   "meta" : {
data:     "vtag": {
data:       "resource-id" : "my-network-map",
data:         "tag" : "da65eca2eb7a10ce8b059740b0b2e3f8eb1d4785"
data:       }
data:     },
data:     "network-map" : {
data:       "PID1" : {
data:         "ipv4" : [ "192.0.2.0/24", "198.51.100.0/25" ]
data:       },
data:       "PID2" : {
data:         "ipv4" : [ "198.51.100.128/25" ]
data:       },
data:       "PID3" : {
data:         "ipv4" : [ "0.0.0.0/0" ],
data:         "ipv6" : [ "::/0" ]
data:       }
data:     }
data:   }
data: }

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

After sending those events immediately, the update stream server will send additional events as the maps change. For example, the following represents a small change to the cost map. PID1->PID2 is changed to 9 from 5, PID3->PID1 is no longer available, and PID3->PID3 is now defined as 1:

```
event: application/merge-patch+json,my-routingcost-map
data: {
data:    "meta" : {
data:       "vtag": {
data:          "tag": "c0ce023b8678a7b9ec00324673b98e54656d1f6d"
data:       }
data:    },
data:    "cost-map": {
data:       "PID1" : { "PID2" : 9 },
data:       "PID3" : { "PID1" : null, "PID3" : 1 }
data:    }
data: }
```

As another example, the following represents a change to the network map: an ipv4 prefix "203.0.113.0/25" is added to PID1. It triggers changes to the cost map. The update stream server chooses to send an incremental change for the network map and send a full replacement instead of an incremental change for the cost map:

```
event: application/json-patch+json,my-network-map
data: {
data:    {
data:       "op": "replace",
data:       "path": "/meta/vtag/tag",
data:       "value" :"a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
data:    },
data:    {
data:       "op": "add",
data:       "path": "/network-map/PID1/ipv4/2",
data:       "value": "203.0.113.0/25"
data:    }
data: }

event: application/alto-costmap+json,my-routingcost-map
data: {
data:    "meta" : {
data:       "vtag": {
data:          "tag": "c0ce023b8678a7b9ec00324673b98e54656d1f6d"
data:       }
data:    },
data:    "cost-map" : {
data:       "PID1": { "PID1": 1,   "PID2": 3,   "PID3": 7 },
data:       "PID2": { "PID1": 12, "PID2": 1,   "PID3": 9 },
data:       "PID3": { "PID1": 14, "PID2": 8   }
data:    }
data: }
```

8.3. Example: Advanced Network and Cost Map Updates

This example is similar to the previous one, except that the ALTO client requests updates for the "hopcount" cost map as well as the "routingcost" cost map and provides the current version tag of the network map, so the update stream server is not required to send the full network map data update message at the beginning of the stream. In this example, the client uses the substream-ids "net", "routing", and "hops" for those resources. The update stream server sends the stream control URI and the full cost maps, followed by updates for the network map and cost maps as they become available:

```
POST /updates/costs HTTP/1.1
Host: alto.example.com
Accept: text/event-stream,application/alto-error+json
Content-Type: application/alto-updatestreamparams+json
Content-Length: 244

{ "add": {
    "net": {
      "resource-id": "my-network-map",
      "tag": "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
    },
    "routing": {
      "resource-id": "my-routingcost-map"
    },
    "hops": {
      "resource-id": "my-hopcount-map"
    }
  }
}
```

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream

event: application/alto-updatestreamcontrol+json
data: {"control-uri":
data: "https://alto.example.com/updates/streams/2718281828459"}

event: application/alto-costmap+json,routing
data: { ... full routingcost cost map message ... }

event: application/alto-costmap+json,hops
data: { ... full hopcount cost map message ... }

    (pause)

event: application/merge-patch+json,routing
data: {"cost-map": {"PID2" : {"PID3" : 31}}}

event: application/merge-patch+json,hops
data: {"cost-map": {"PID2" : {"PID3" : 4}}}
```

If the ALTO client wishes to stop receiving updates for the "hopcount" cost map, the ALTO client can send a "remove" request on the stream control URI:

```
POST /updates/streams/2718281828459 HTTP/1.1
Host: alto.example.com
Accept: text/plain,application/alto-error+json
Content-Type: application/alto-updatestreamparams+json
Content-Length: 24

{
  "remove": [ "hops" ]
}


HTTP/1.1 204 No Content
Content-Length: 0

    (stream closed without sending data content)
```

The update stream server sends a "stopped" control update message on the original request stream to inform the ALTO client that updates are stopped for that resource:

```
event: application/alto-updatestreamcontrol+json
data: {
data:    "stopped": ["hops"]
data: }
```

Below is an example of an invalid stream control request. The "remove" field of the request includes an undefined substream-id, and the stream control server will return an error response to the ALTO client.

```
POST /updates/streams/2718281828459 HTTP/1.1
Host: alto.example.com
Accept: text/plain,application/alto-error+json
Content-Type: application/alto-updatestreamparams+json
Content-Length: 31
{
  "remove": [ "properties" ]
}

HTTP/1.1 400 Bad Request
Content-Length: 89
Content-Type: application/alto-error+json

{
  "meta":{
  "code": "E_INVALID_FIELD_VALUE",
  "field": "remove",
  "value": "properties"
}
```

If the ALTO client no longer needs any updates and wishes to shut the update stream down gracefully, the client can send a "remove" request with an empty array:

```
POST /updates/streams/2718281828459 HTTP/1.1
Host: alto.example.com
Accept: text/plain,application/alto-error+json
Content-Type: application/alto-updatestreamparams+json
Content-Length: 17

{
  "remove": [ ]
}


HTTP/1.1 204 No Content
Content-Length: 0

    (stream closed without sending data content)
```

The update stream server sends a final control update message on the original request stream to inform the ALTO client that all updates are stopped and then closes the stream:

```
event: application/alto-updatestreamcontrol+json
data: {
data:    "stopped": ["net", "routing"]
data: }

    (server closes stream)
```

8.4. Example: Endpoint Property Updates

As another example, here is how an ALTO client can request updates for the property "priv:ietf-bandwidth" for one set of endpoints and "priv:ietf-load" for another. The update stream server immediately sends full replacements with the property values for all endpoints. After that, the update stream server sends data update messages for the individual endpoints as their property values change.

```
POST /updates/properties HTTP/1.1
Host: alto.example.com
Accept: text/event-stream
Content-Type: application/alto-updatestreamparams+json
Content-Length: 511

{ "add": {
    "props-1": {
      "resource-id": "my-props",
      "input": {
        "properties" : [ "priv:ietf-bandwidth" ],
        "endpoints" : [
          "ipv4:198.51.100.1",
          "ipv4:198.51.100.2",
          "ipv4:198.51.100.3"
        ]
      }
    },
    "props-2": {
      "resource-id": "my-props",
      "input": {
        "properties" : [ "priv:ietf-load" ],
        "endpoints" : [
          "ipv6:2001:db8:100::1",
          "ipv6:2001:db8:100::2",
          "ipv6:2001:db8:100::3"
        ]
      }
    }
  }
}
```

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream

event: application/alto-updatestreamcontrol+json
data: {"control-uri":
data: "https://alto.example.com/updates/streams/1414213562373"}

event: application/alto-endpointprops+json,props-1
data: { "endpoint-properties": {
data:     "ipv4:198.51.100.1" : { "priv:ietf-bandwidth": "13" },
data:     "ipv4:198.51.100.2" : { "priv:ietf-bandwidth": "42" },
data:     "ipv4:198.51.100.3" : { "priv:ietf-bandwidth": "27" }
data:  } }

event: application/alto-endpointprops+json,props-2
data: { "endpoint-properties": {
data:     "ipv6:2001:db8:100::1" : { "priv:ietf-load": "8" },
data:     "ipv6:2001:db8:100::2" : { "priv:ietf-load": "2" },
data:     "ipv6:2001:db8:100::3" : { "priv:ietf-load": "9" }
data:  } }

   (pause)

event: application/merge-patch+json,props-1
data: { "endpoint-properties":
data:   {"ipv4:198.51.100.1" : {"priv:ietf-bandwidth": "3"}}
data: }

   (pause)

event: application/merge-patch+json,props-2
data: { "endpoint-properties":
data:   {"ipv6:2001:db8:100::3" : {"priv:ietf-load": "7"}}
data: }
```

If the ALTO client needs the "priv:ietf-bandwidth" property and the "priv:ietf-load" property for additional endpoints, the ALTO client can send an "add" request on the stream control URI:

```
POST /updates/streams/1414213562373" HTTP/1.1
Host: alto.example.com
Accept: text/plain,application/alto-error+json
Content-Type: application/alto-updatestreamparams+json
Content-Length: 448

{ "add": {
    "props-3": {
      "resource-id": "my-props",
      "input": {
        "properties" : [ "priv:ietf-bandwidth" ],
        "endpoints" : [
          "ipv4:198.51.100.4",
          "ipv4:198.51.100.5"
        ]
      }
    },
    "props-4": {
      "resource-id": "my-props",
      "input": {
        "properties" : [ "priv:ietf-load" ],
        "endpoints" : [
          "ipv6:2001:db8:100::4",
          "ipv6:2001:db8:100::5"
        ]
      }
    }
  }
}


HTTP/1.1 204 No Content
Content-Length: 0

    (stream closed without sending data content)
```

The update stream server sends full replacements for the two new resources, followed by incremental changes for all four requests as they arrive:

```
event: application/alto-endpointprops+json,props-3
data: { "endpoint-properties": {
data:     "ipv4:198.51.100.4" : { "priv:ietf-bandwidth": "25" },
data:     "ipv4:198.51.100.5" : { "priv:ietf-bandwidth": "31" },
data:   } }

event: application/alto-endpointprops+json,props-4
data: { "endpoint-properties": {
data:     "ipv6:2001:db8:100::4" : { "priv:ietf-load": "6" },
data:     "ipv6:2001:db8:100::5" : { "priv:ietf-load": "4" },
data:   } }

    (pause)

event: application/merge-patch+json,props-3
data: { "endpoint-properties":
data:   {"ipv4:198.51.100.5" : {"priv:ietf-bandwidth": "15"}}
data: }

    (pause)

event: application/merge-patch+json,props-2
data: { "endpoint-properties":
data:   {"ipv6:2001:db8:100::2" : {"priv:ietf-load": "9"}}
data: }

    (pause)

event: application/merge-patch+json,props-4
data: { "endpoint-properties":
data:   {"ipv6:2001:db8:100::4" : {"priv:ietf-load": "3"}}
data: }
```

8.5. Example: Multipart Message Updates

This example shows how an ALTO client can request a nonstandard ALTO service returning a multipart response. The update stream server immediately sends full replacements of the multipart response. After that, the update stream server sends data update messages for the individual parts of the response as the ALTO data (object) in each part changes.

```
POST /updates/pv HTTP/1.1
Host: alto.example.com
Accept: text/event-stream
Content-Type: application/alto-updatestreamparams+json
Content-Length: 382

{
  "add": {
    "ecspvsub1": {
      "resource-id": "my-pv",
      "input": {
        "cost-type": {
          "cost-mode": "array",
          "cost-metric": "ane-path"
        },
        "endpoints": {
          "srcs": [ "ipv4:192.0.2.2" ],
          "dsts": [ "ipv4:192.0.2.89", "ipv4:203.0.113.45" ]
        },
        "ane-properties": [ "maxresbw", "persistent-entities" ]
      }
    }
  }
}
```

```
    HTTP/1.1 200 OK
    Connection: keep-alive
    Content-Type: text/event-stream

    event: application/alto-updatestreamcontrol+json
    data: {"control-uri":
    data:    "https://alto.example.com/updates/streams/1414"}

    event: multipart/related;boundary=example-pv;
          type=application/alto-endpointcost+json,ecspvsub1
    data: --example-pv
    data: Content-ID: ecsmap
    data: Content-Type: application/alto-endpointcost+json
    data:
    data: { ... data (object) of an endpoint cost map ... }
    data: --example-pv
    data: Content-ID: propmap
    data: Content-Type: application/alto-propmap+json
    data:
    data: { ... data (object) of a property map ... }
    data: --example-pv--

       (pause)

    event: application/merge-patch+json,ecspvsub1.ecsmap
    data: { ... merge patch for updates of ecspvsub1.ecsmap ... }

    event: application/merge-patch+json,ecspvsub1.propmap
    data: { ... merge patch for updates of ecspvsub1.propmap ... }
```

9. Operation and Processing Considerations

9.1. Considerations for Choosing Data Update Messages

The update stream server should be cognizant of the effects of its update schedule, which includes both the choice of timing (i.e., when/what to trigger an update) and the choice of message format (i.e., given an update, send a full replacement or an incremental change). In particular, the update schedule can have effects on both the overhead and the freshness of information. To minimize overhead, the server may choose to batch a sequence of updates for resources that frequently change by sending cumulative updates or a full replacement after a while. The update stream server should be cognizant that batching reduces the freshness of information. The server should also consider the effect of such delays on client behaviors (see below on client timeout on waiting for updates of dependent resources).

For incremental updates, this design allows both JSON patch and JSON merge patch for incremental changes. JSON merge patch is clearly superior to JSON patch for describing incremental changes to cost maps, endpoint costs, and endpoint properties. For these data structures, JSON merge patch is more space efficient, as well as simpler to apply. There is no advantage allowing a server to use JSON patch for those resources.

The case is not as clear for incremental changes to network maps.

First, consider small changes, such as moving a prefix from one PID to another. JSON patch could encode that as a simple insertion and deletion, while JSON merge patch would have to replace the entire array of prefixes for both PIDs. On the other hand, to process a JSON patch update, the ALTO client would have to retain the indexes of the prefixes for each PID. Logically, the prefixes in a PID are an unordered set, not an array; aside from handling updates, a client has no need to retain the array indexes of the prefixes. Hence, to take advantage of JSON patch for network maps, ALTO clients would have to retain additional, otherwise unnecessary, data.

Second, consider more involved changes, such as removing half of the prefixes from a PID. JSON merge patch would send a new array for that PID, while JSON patch would have to send a list of remove operations and delete the prefix one by one.

Therefore, each update stream server may decide on its own whether to use JSON merge patch or JSON patch according to the changes in network maps.

9.2. Considerations for Client Processing Data Update Messages

In general, when an ALTO client receives a full replacement for a resource, the ALTO client should replace the current version with the new version. When an ALTO client receives an incremental change for a resource, the ALTO client should apply those patches to the current version of the resource.

However, because resources can depend on other resources (e.g., cost maps depend on network maps), an ALTO client **MUST NOT** use a dependent resource if the resource on which it depends has changed. There are at least two ways an ALTO client can do that. The following paragraphs illustrate these techniques by referring to network and cost map messages, although these techniques apply to any dependent resources.

Note that when a network map changes, the update stream server **MUST** send the network map update message before sending the updates for the dependent cost maps (see Section 6.7.1).

One approach is for the ALTO client to save the network map update message in a buffer and continue to use the previous network map and the associated cost maps until the ALTO client receives the update messages for all dependent cost maps. The ALTO client then applies all network and cost map updates atomically.

Alternatively, the ALTO client **MAY** update the network map immediately. In this case, the cost maps using the network map become invalid because they are inconsistent with the current network map; hence, the ALTO client **MUST** mark each such dependent cost map as temporarily invalid and **MUST NOT** use each such cost map until the ALTO client receives a cost map update message indicating that it is based on the new network map version tag.

The update stream server **SHOULD** send updates for dependent resources (i.e., the cost maps in the preceding example) in a timely fashion. However, if the ALTO client does not receive the expected updates, a simple recovery method is that the ALTO client closes the update stream connection, discards the dependent resources, and reestablishes the update stream. The ALTO

client **MAY** retain the version tag of the last version of any tagged resources and give those version tags when requesting the new update stream. In this case, if a version is still current, the update stream server will not resend that resource.

Although not as efficient as possible, this recovery method is simple and reliable.

9.3. Considerations for Updates to Filtered Cost Maps

If an update stream provides updates to a Filtered Cost Map that allows constraint tests, then an ALTO client **MAY** request updates to a Filtered Cost Map request with a constraint test. In this case, when a cost changes, the update stream server **MUST** send an update if the new value satisfies the test. If the new value does not, whether the update stream server sends an update depends on whether the previous value satisfied the test. If it did not, the update stream server **SHOULD NOT** send an update to the ALTO client. But if the previous value did, then the update stream server **MUST** send an update with a "null" value to inform the ALTO client that this cost no longer satisfies the criteria.

An update stream server can avoid having to handle such a complicated behavior by offering update streams only for Filtered Cost Maps that do not allow constraint tests.

9.4. Considerations for Updates to Ordinal Mode Costs

For an ordinal mode cost map, a change to a single cost point may require updating many other costs. As an extreme example, suppose the lowest cost changes to the highest cost. For a numerical mode cost map, only that one cost changes. But for an ordinal mode cost map, every cost might change. While this document allows an update stream server to offer incremental updates for ordinal mode cost maps, update stream server implementors should be aware that incremental updates for ordinal costs are more complicated than for numerical costs, and ALTO clients should be aware that small changes may result in large updates.

An update stream server can avoid this complication by only offering full replacements for ordinal cost maps.

9.5. Considerations for SSE Text Formatting and Processing

SSE was designed for events that consist of relatively small amounts of line-oriented text data, and SSE clients frequently read input one line at a time. However, an update stream sends a full cost map as a single events, and a cost map may involve megabytes, if not tens of megabytes, of text. This has implications that the ALTO client and the update stream server may consider.

First, some SSE client libraries read all data for an event into memory and then present it to the client as a character array. However, a client may not have enough memory to hold the entire JSON text for a large cost map. Hence, an ALTO client **SHOULD** consider using an SSE library that presents the event data in manageable chunks, so the ALTO client can parse the cost map incrementally and store the underlying data in a more compact format.

Second, an SSE client library may use a low-level, generic socket read library that stores each line of an event data, just in case the higher-level parser may need the line delimiters as part of the protocol formatting. A server sending a complete cost map as a single line may then generate a multi-megabyte data "line", and such a long line may then require complex memory management at the client. It is **RECOMMENDED** that an update stream server limit the lengths of data lines.

Third, an SSE server may use a library, which may put line breaks in places that would have semantic consequences for the ALTO updates; see Section 11. The update stream server implementation **MUST** ensure that no line breaks are introduced to change the semantics.

10. Security Considerations

The security considerations (Section 15 of [RFC7285]) of the base protocol fully apply to this extension. For example, the same authenticity and integrity considerations (Section 15.1 of [RFC7285]) still fully apply; the same considerations for the privacy of ALTO users (Section 15.4 of [RFC7285]) also still fully apply.

The additional services (addition of update streams and stream control URIs) provided by this extension extend the attack surface described in Section 15.1.1 of [RFC7285]. Below, we discuss the additional risks and their remedies.

10.1. Update Stream Server: Denial-of-Service Attacks

Allowing persistent update stream connections enables a new class of Denial-of-Service attacks.

For the update stream server, an ALTO client might create an unreasonable number of update stream connections or add an unreasonable number of substream-ids to one update stream.

To avoid these attacks on the update stream server, the server **SHOULD** choose to limit the number of active streams and reject new requests when that threshold is reached. An update stream server **SHOULD** also choose to limit the number of active substream-ids on any given stream or limit the total number of substream-ids used over the lifetime of a stream and reject any stream control request that would exceed those limits. In these cases, the update stream server **SHOULD** return the HTTP status "503 Service Unavailable".

It is important to note that the preceding approaches are not the only possibilities. For example, it may be possible for the update stream server to use somewhat more clever logic involving IP reputation, rate-limiting, and compartmentalization of the overall threshold into smaller thresholds that apply to subsets of potential clients.

While the preceding techniques prevent update stream DoS attacks from disrupting an update stream server's other services, it does make it easier for a DoS attack to disrupt the update stream service. Therefore, an update stream server **MAY** prefer to restrict update stream services to authorized clients, as discussed in Section 15 of [RFC7285].

Alternatively, an update stream server **MAY** return the HTTP status "307 Temporary Redirect" to redirect the client to another ALTO server that can better handle a large number of update streams.

10.2. ALTO Client: Update Overloading or Instability

The availability of continuous updates can also cause overload for an ALTO client, in particular, an ALTO client with limited processing capabilities. The current design does not include any flow control mechanisms for the client to reduce the update rates from the server. Under overloading, the client **MAY** choose to remove the information resources with high update rates.

Also, under overloading, the client may no longer be able to detect whether information is still fresh or has become stale. In such a case, the client should be careful in how it uses the information to avoid stability or efficiency issues.

10.3. Stream Control: Spoofed Control Requests and Information Breakdown

An outside party that can read the update stream response or that can observe stream control requests can obtain the control URI and use that to send a fraudulent "remove" requests, thus disabling updates for the valid ALTO client. This can be avoided by encrypting the update stream and stream control requests (see Section 15 of [RFC7285]). Also, the update stream server echoes the "remove" requests on the update stream, so the valid ALTO client can detect unauthorized requests.

In general, as the architecture allows the possibility for the update stream server and the stream control server to be different entities, the additional risks should be evaluated and remedied. For example, the private communication path between the servers may be attacked, resulting in a risk of communications breakdown between them, as well as invalid or spoofed messages claiming to be on that private communications path. Proper security mechanisms, including confidentiality, authenticity, and integrity mechanisms, should be considered.

11. Requirements on Future ALTO Services to Use This Design

Although this design is quite flexible, it has underlying requirements.

The key requirements are that (1) each data update message is for a single resource and (2) an incremental change can be applied only to a resource that is a single JSON object, as both JSON merge patch and JSON patch can apply only to a single JSON object. Hence, if a future ALTO resource can contain multiple objects, then either each individual object also has a resource-id or an extension to this design is made.

At the low-level encoding level, new line in SSE has its own semantics. Hence, this design requires that resource encoding does not include new lines that can be confused with SSE encoding. In particular, the data update message **MUST NOT** include "event: " or "data: " at a new line as part of data message.

If an update stream provides updates to a Filtered Cost Map that allows constraint tests, the requirements for such services are stated in Section 9.3.

12. IANA Considerations

This document defines two new media types: "application/alto-updatestreamparams+json", as described in Section 6.5, and "application/alto-updatestreamcontrol+json", as described in Section 5.3. All other media types used in this document have already been registered, either for ALTO, JSON merge patch, or JSON patch.

12.1. application/alto-updatestreamparams+json Media Type

Type name: application

Subtype name: alto-updatestreamparams+json

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: Encoding considerations are identical to those specified for the "application/json" media type. See [RFC8259].

Security considerations: Security considerations relating to the generation and consumption of ALTO Protocol messages are discussed in Section 10 of RFC 8895 and Section 15 of [RFC7285].

Interoperability considerations: RFC 8895 specifies format of conforming messages and the interpretation thereof.

Published specification: Section 6.5 of RFC 8895.

Applications that use this media type: ALTO servers and ALTO clients either stand alone or are embedded within other applications.

Fragment identifier considerations: N/A

Additional information:

   Deprecated alias names for this type: N/A

   Magic number(s):   N/A

   File extension(s):   RFC 8895 uses the media type to refer to protocol messages and thus does not require a file extension.

   Macintosh file type code(s):   N/A

Person & email address to contact for further information: See Authors' Addresses section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section.

Change controller: Internet Engineering Task Force (mailto:iesg@ietf.org).

12.2. application/alto-updatestreamcontrol+json Media Type

Type name: application

Subtype name: alto-updatestreamcontrol+json

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: Encoding considerations are identical to those specified for the "application/json" media type. See [RFC8259].

Security considerations: Security considerations relating to the generation and consumption of ALTO Protocol messages are discussed in Section 10 of RFC 8895 and Section 15 of [RFC7285].

Interoperability considerations: RFC 8895 specifies format of conforming messages and the interpretation thereof.

Published specification: Section 5.3 of RFC 8895.

Applications that use this media type: ALTO servers and ALTO clients either stand alone or are embedded within other applications.

Fragment identifier considerations: N/A

Additional information:

    Deprecated alias names for this type: N/A

    Magic number(s):   N/A

    File extension(s):   RFC 8895 uses the media type to refer to protocol messages and thus does not require a file extension.

    Macintosh file type code(s):   N/A

Person & email address to contact for further information: See Authors' Addresses section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section.

Change controller: Internet Engineering Task Force (mailto:iesg@ietf.org).

13. Appendix: Design Decision: Not Allowing Stream Restart

If an update stream is closed accidentally, when the ALTO client reconnects, the update stream server must resend the full maps. This is clearly inefficient. To avoid that inefficiency, the SSE specification allows an update stream server to assign an id to each event. When an ALTO client reconnects, the ALTO client can present the id of the last successfully received event, and the update stream server restarts with the next event.

However, that mechanism adds additional complexity. The update stream server must save SSE messages in a buffer in case ALTO clients reconnect. But that mechanism will never be perfect: If the ALTO client waits too long to reconnect or if the ALTO client sends an invalid ID, then the update stream server will have to resend the complete maps anyway.

Furthermore, this is unlikely to be a problem in practice. ALTO clients who want continuous updates for large resources, such as full network and cost maps, are likely to be things like P2P trackers. These ALTO clients will be well connected to the network; they will rarely drop connections.

Mobile devices certainly can and do drop connections and will have to reconnect. But mobile devices will not need continuous updates for multi-megabyte cost maps. If mobile devices need continuous updates at all, they will need them for small queries, such as the costs from a small set of media servers from which the device can stream the currently playing movie. If the mobile device drops the connection and reestablishes the update stream, the update stream server will have to retransmit only a small amount of redundant data.

In short, using event ids to avoid resending the full map adds a considerable amount of complexity to avoid a situation that is very rare. The complexity is not worth the benefit.

The update stream service does allow the ALTO client to specify the tag of the last received version of any tagged resource, and if that is still current, the update stream server need not retransmit the full resource. Hence, ALTO clients can use this to avoid retransmitting full network maps. Cost maps are not tagged, so this will not work for them. Of course, the ALTO protocol could be extended by adding version tags to cost maps, which would solve the retransmission-on-reconnect problem. However, adding tags to cost maps might add a new set of complications.

14. References

14.1. 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>.

[RFC2387] Levinson, E., "The MIME Multipart/Related Content-type", RFC 2387, DOI
           10.17487/RFC2387, August 1998, <https://www.rfc-editor.org/info/rfc2387>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI):
           Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005,
           <https://www.rfc-editor.org/info/rfc3986>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration
           Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://
           www.rfc-editor.org/info/rfc6838>.

[RFC6902] Bryan, P., Ed. and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Patch",
           RFC 6902, DOI 10.17487/RFC6902, April 2013, <https://www.rfc-editor.org/info/
           rfc6902>.

[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>.

[RFC7396] Hoffman, P. and J. Snell, "JSON Merge Patch", RFC 7396, DOI 10.17487/RFC7396,
           October 2014, <https://www.rfc-editor.org/info/rfc7396>.

[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>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format",
           STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-
           editor.org/info/rfc8259>.

[SSE]   Hickson, I., "Server-Sent Events", W3C Recommendation, February 2015,
           <https://www.w3.org/TR/eventsource/>.

14.2. Informative References

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI
           10.17487/RFC4960, September 2007, <https://www.rfc-editor.org/info/rfc4960>.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/
           RFC5789, March 2010, <https://www.rfc-editor.org/info/rfc5789>.

[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>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1):
           Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://
           www.rfc-editor.org/info/rfc7231>.

[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>.

Acknowledgments

Contributors

Sections 2, 5.1, 5.2, and 8.5 of this document are based on contributions from Jingxuan Jensen Zhang, and he is considered an author.

Authors' Addresses

Wendy Roome
Nokia Bell Labs (Retired)
124 Burlington Rd
Murray Hill, NJ 07974
United States of America
Phone: +1-908-464-6975
Email: wendy@wdroome.com

Y. Richard Yang
Yale University
51 Prospect St
New Haven, CT
United States of America
Email: yry@cs.yale.edu