ALTO WG
Internet-Draft
Intended status: Standards Track
Expires: March 17, 2017

W. Roome
Nokia Bell Labs
Y. Yang
Tongji/Yale University

September 13, 2016

ALTO Incremental Updates Using Server-Sent Events (SSE)
draft-ietf-alto-incr-update-sse-03

Abstract

The Application-Layer Traffic Optimization (ALTO) [RFC7285] protocol provides network related information to client applications so that clients may make informed decisions. To that end, an ALTO Server provides Network and Cost Maps. Using those maps, an ALTO Client can determine the costs between endpoints.

However, the ALTO protocol does not define a mechanism to allow an ALTO client to obtain updates to those maps, other than by periodically re-fetching them. Because the maps may be large (potentially tens of megabytes), and because only parts of the maps may change frequently (especially Cost Maps), that can be extremely inefficient.

Therefore this document presents a mechanism to allow an ALTO Server to provide updates to ALTO Clients. Updates can be both immediate, in that the server can send updates as soon as they are available, and incremental, in that if only a small section of a map changes, the server can send just the changes.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Status of this Memo

and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on March 17, 2017.

Table of Contents

1. Introduction

The Application-Layer Traffic Optimization (ALTO) [RFC7285] protocol provides network related information to client applications so that clients may make informed decisions. To that end, an ALTO Server provides Network and Cost Maps, where a Network Map partitions the set of endpoints into a manageable number of Provider-Defined Identifiers (PIDs), and a Cost Map provides directed costs between PIDs. Given Network and Cost Maps, an ALTO Client can obtain costs between endpoints by using the Network Map to get the PID for each endpoint, and then using the Cost Map to get the costs between those PIDs.

However, the ALTO protocol does not define a mechanism to allow a client to obtain updates to those maps, other than by periodically re-fetching them. Because the maps may be large (potentially tens of megabytes), and because parts of the maps may change frequently (especially Cost Maps), that can be extremely inefficient.

Therefore this document presents a mechanism to allow an ALTO Server to provide incremental updates to ALTO Clients. Updates can be both immediate, in that the server can send updates as soon as they are available, and incremental, in that if only a small section of a map changes, the server can send just the changes.

While primarily intended to provide updates to Network and Cost Maps, the mechanism defined in this document can provide updates to any ALTO resource, including POST-mode services such as Endpoint Property and Endpoint Cost Services, as well as new ALTO services to be defined by future extensions.

The rest of this document is organized as follows. Section 2 gives an overview of the incremental update approach, which is based on Server-Sent Events (SSEs). Section 4 and Section 5 give SSEs and JSON Merge Patch, the technologies on which ALTO updates are based. Section 6 defines the update events, Section 7 and Section 8 define the update services themselves, and Section 9 gives several examples. Section 10 describes how a client should handle incoming updates. Section 11 and Section 12 discuss the design decisions behind this update mechanism and other considerations. The remaining sections review the security and IANA considerations.

2. Overview of Approach

This section presents a non-normative overview of the update mechanism to be defined in this document.

An ALTO Server can offer one or more Update Stream resources, where each Update Stream resource (or Update Stream for short) is a POST-mode service that returns a continuous sequence of update messages for one or more ALTO resources. An Update Stream can provide updates to both GET-mode resources, such as Network and Cost Maps, and POST-mode resources, such as Endpoint Property Services.

Each update message updates one resource, and is sent as a Server-Sent Event (SSE), as defined by [SSE]. An update message is either a full replacement or else an incremental change. Full replacement updates use the JSON message formats defined by the ALTO protocol. Incremental updates use JSON Merge Patch ([RFC7396]) to describe the changes to the resource. The ALTO Server decides when to send update messages, and whether to send full replacements or incremental updates. These decisions can vary from resource to resource and from update to update.

An ALTO Server may offer any number of Update Stream resources, for any subset of the server's resources. An ALTO Server's Information Resource Directory (IRD) defines the Update Stream resources, and declares the set of resources for which each Update Stream provides updates. The server selects the resource set for each stream. 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 stream. Thus the Update Stream for a Cost Map should also provide updates for the Network Map on which that Cost Map depends.

When an ALTO Client requests an Update Stream resource, the client establishes a new persistent connection to the server. The server responds by sending an event with the URI of a stream-control resource for this update stream. The control URI allows a client to modify the newly-created update stream. For example, the client can ask the server to send update events for additional resources, to stop sending update events for previously requested resources, or to gracefully stop and close the update stream altogether.

A client may request any number of Update Streams simultaneously. Because each stream consumes resources on the server, a server may limit the number of open Update Streams, may close inactive streams, may provide Update Streams via other processors, or may require client authorization/authentication.

3. Changes Since Version -01

- Defined a new "Stream Control" resource (Section 8) to allow clients to add or remove resources from a previously created 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.

- The client now assigns a unique client-id to each resource in an update stream. The server puts the client-id in each update event for that resource (before, the server used the server's resource-id). This allows a client to use one stream to get updates to two different Endpoint Cost requests (before, that required two separate streams).

4. Overview of Server-Sent Events (SSEs)

The following is a non-normative summary of Server-Sent Events (SSEs). See [SSE] for the normative definition.

Server-Sent Events 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 which start with a colon) are ignored, as are lines which 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" or "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: good bye
```

Figure 1: A Sample SSE stream.

5. Incremental Update Message Format

5.1. Overview of JSON Merge Patch

The following is a non-normative summary of JSON Merge Patch. See [RFC7396] for the normative definition.

JSON Merge Patch is intended to allow applications to update server resources via the HTTP PATCH method [RFC5789]. This document adopts the JSON Merge Patch message format to encode the changes, but uses a different transport mechanism.

Informally, a Merge Patch object is a JSON data structure that defines how to transform one JSON value into another. Merge Patch treats the two JSON values as trees of nested JSON Objects (dictionaries of name-value pairs), where the leaves are values other than JSON Objects (e.g., JSON Arrays, Strings, Numbers, etc.), 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 Merge Patch tree 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 Merge Patch tree has a leaf with a JSON "null"

value. The Merge Patch tree does not have an entry for any leaf that has the same value in both versions.

As a result, if all leaf values are simple scalars, JSON Merge Patch is a very efficient representation of the change. It is less efficient when leaf values are arrays, because JSON Merge Patch replaces arrays in their entirety, even if only one entry changes.

Formally, the process of applying a Merge Patch is defined by the following recursive algorithm, as specified in [RFC7396]:

```
define MergePatch(Target, Patch) {
  if Patch is an Object {
    if Target is not an Object {
      Target = {} # Ignore the contents and
                  # set it to an empty Object
    }
    for each Name/Value pair in Patch {
      if Value is null {
        if Name exists in Target {
          remove the Name/Value pair from Target
        }
      } else {
        Target[Name] = MergePatch(Target[Name], Value)
      }
    }
    return Target
  } else {
    return Patch
  }
}
```

Note that null as the value of a name/value pair will delete the element with "name" in the original JSON value.

5.2. JSON Merge Patch Applied to Network Map Messages

Section 11.2.1.6 of [RFC7285] defines the format of a Network Map message. Here is a simple example:

```
{
  "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" ]
    }
  }
}
```

When applied to that message, the following Merge Patch update message adds the ipv6 prefix "2001:db8:8000::/33" to "PID1", deletes "PID2", and assigns a new "tag" to the Network Map:

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

Here is the updated 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" ],
      "ipv6" : [ "2001:db8:8000::/33" ]
    },
    "PID3" : {
      "ipv4" : [ "0.0.0.0/0" ],
      "ipv6" : [ "::/0" ]
    }
  }
}
```

5.3. JSON Merge Patch Applied to Cost Map Messages

Section 11.2.3.6 of [RFC7285] defines the format of a Cost Map message. Here is a simple example:

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

The following Merge Patch message updates the example cost map so that PID1->PID2 is 9 instead of 5, PID3->PID1 is no longer available, and PID3->PID3 is now defined as 1:

```
{
  "cost-map" : {
    "PID1" : { "PID2" : 9 },
    "PID3" : { "PID1" : null, "PID3" : 1 }
  }
}
```

Here is the updated cost map:

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

6. ALTO Event Stream

The Update Stream service (Section 7) returns a stream of Update Events (Section 6.2) and Control Events (Section 6.3).

6.1. ALTO Event Format

Update and Control Events have the same basic structure. The data field is a JSON object, and the event field contains the media type of the data field, and an optional client id. Update Events use the client id to identify the ALTO resource to which the update message applies. Client ids MUST follow the rules for ALTO ResourceIds (see {10.2} of [RFC7285]. Client ids MUST be unique within an Update Stream, but need not be globally unique. For example, if a client requests updates for both a Cost Map and its Network Map, the client might assign id "1" to the Network Map and "2" to the Cost Map. Alternatively, the client could use the ALTO resource ids for those two maps.

JSON specifications use the type ClientId for a client-id.

The two sub-fields of the event field are encoded as comma-separated strings:

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

Note that media type names may not contain a comma (character code 0x2c).

The Update Stream Service does not use the SSE "id" field.

6.2. ALTO Update Events

The Update Stream Service sends an update event when a monitored resource changes. The data is either a complete specification of the resource, or else a JSON Merge Patch object describing the changes from the last version. We will refer to these as full-replacement and Merge Patch messages, respectively. The data objects in full-replacement messages are defined by [RFC7285]; examples are Network and Cost Map messages. They have the media types defined in that document. The data objects in Merge Patch messages are defined by [RFC7396], and they have the media type "application/merge-patch+json", as defined by [RFC7396].

Figure 2 shows some examples of ALTO update events:

```
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: { ... Merge Patch update for the Cost Map ... }
```

Figure 2: Examples of ALTO update events.

6.3. ALTO Control Events

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

```
object {
   [String     control-uri;]
   [String     remove<1..*>;]
} UpdateStreamControlEvent;
```

The "control-uri" field is the URI of the Stream Control resource for this Update Stream (Section 8). The ALTO server MUST send a control event with that URI as the first event in an Update Stream.

The "remove" field is a list of client-ids of resources for which the server will no longer send updates. The server sends this event after processing a Stream Controller request to remove those resources (Section 7.6.2).

7. Update Stream Service

An Update Stream returns a stream of SSE messages, as defined in Section 6. An ALTO Server's IRD (Information Resource Directory) MAY define one or more Update Stream resources, which clients use to request new Update Stream instances.

When a server creates a new Update Stream, it also create a new Stream Controller for that Update Stream. A client uses that Stream Controller to remove resources from the Update Stream instance, or to request updates for additional resources. A client cannot obtain the Stream Controller through the IRD. Instead, the first event that the server sends to the client has the URI for the associated controller (see Section 6.3.

Section 8 describes the Stream Controller.

7.1. Media Type

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

7.2. HTTP Method

An ALTO Update Stream is requested using the HTTP POST method.

7.3. 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;]
   [ClientId        remove<0..*>;]
} UpdateStreamReq;

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

object {
   String       resource-id;
   [String      tag;]
   [Boolean     incremental-updates;]
   [Object      input;]
} AddUpdateReq;
```

The "add" field specifies the resources for which the client wants updates, and has one entry for each resource. The client creates a unique client-id (Section 6.1) for each such resource, and uses those client-ids as the keys in the "add" field.

An Update Stream request MUST have an "add" field specifying one or more resources. If it does not, the server MUST return an E_INVALID_FIELD_VALUE error response (see Section 8.5.2 of [RFC7285]), and MUST close the stream without sending any events.

The "resource-id" field is the resource-id of an ALTO resource, and MUST be in the Update Streams's "uses" list (see Section 7.5). If any resource-id is invalid, or is not associated with this Update Stream, the server MUST return an E_INVALID_FIELD_VALUE error response (see Section 8.5.2 of [RFC7285]), and MUST close the stream without sending any events.

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 if the client has previously retrieved a version of that resource from the server, the 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 server MUST send a full-replacement update before sending any incremental updates, as described in Section 7.6.2. If that version is still current, the ALTO Server MAY omit the initial full-replacement update.

If the "incremental-updates" field for a resource-id is "true", the server MAY send incremental update events for this resource-id (assuming the server supports incremental updates for that resource; see Section 7.4). If the "incremental-updates" field is "false", the ALTO Server MUST NOT send incremental update events for that

resource. In this case, whenever a change occurs, the server MUST send a full-replacement update instead of an incremental update. The server MAY wait until more changes are available, and send a single full-replacement update with those changes. Thus an ALTO Client which declines to accept incremental updates may not get updates as quickly as a client which does.

The default for "incremental-updates" is "true", so to suppress incremental updates, the client MUST explicitly set "incremental-updates" to "false". Note that the client cannot suppress full-replacement update events.

If the resource is a POST-mode service which requires input, the client MUST set the "input" field to a JSON Object with the parameters that resource expects. If the "input" field is missing or invalid, the ALTO Server MUST return the same error response that that resource would return for missing or invalid input (see [RFC7285]). In this case, the server MUST close the Update Stream without sending any events. If the inputs for several POST-mode resources are missing or invalid, the server MUST pick one error response and return it.

The "remove" field is used in Stream Controller requests (see Section 8), and is not allowed in the Update Stream request. If the "remove" field exists, the server MUST return an E_INVALID_FIELD_VALUE error response (see Section 8.5.2 of [RFC7285]), and MUST close the stream without sending any events.

7.4. Capabilities

The capabilities are defined by an object of type UpdateStreamCapabilities:

```
object {
  IncrementalUpdateMediaTypes incremental-update-media-types;
} UpdateStreamCapabilities;

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

If this Update Stream can provide incremental update events for a resource, the "incremental-update-media-types" field has an entry for that resource-id, and the value is the media-type of the incremental update message. Normally this will be "application/merge-patch+json", because, as described in Section 6, JSON Merge Patch is the only incremental update event type defined by this document. However future extensions may define other types of

incremental updates.

7.5. Uses

The "uses" attribute MUST be an array with the resource-ids of every resource for which this stream can provide updates.

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 select 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.

7.6. Response

The response is a stream of SSE update events. Section 6 defines the events, and [SSE] defines how they are encoded into a stream.

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

There are additional requirements on the server's response, as described below.

7.6.1. Keep-Alive Messages

In an SSE stream, any line which starts with a colon (U+003A) character is a comment, and an ALTO Client MUST ignore that line ([SSE]). As recommended in [SSE], an ALTO Server SHOULD send a comment line (or an event) every 15 seconds to prevent clients and proxy servers from dropping the HTTP connection.

7.6.2. Event Sequence Requirements

- The first event MUST be a control event with the URI of the Stream Controller (Section 8) for this Update Stream (Section 6.3).

- As soon as possible after the client initiates the connection, the ALTO Server MUST send a full-replacement update event for each resource-id requested by the client. The only exception is for a GET-mode resource with a version tag. In this case the server MAY omit the initial full-replacement event for that resource if the "tag" field the client provided for that resource-id matches the

tag of the server's current version.

- If this stream provides updates for resource-ids R0 and R1, and if R1 depends on R0, then the ALTO Server MUST send the update for R0 before sending the related update for R1. For example, suppose a stream provides updates to a Network Map and its dependent Cost Maps. When the Network Map changes, the ALTO Server MUST send the Network Map update before sending the Cost Map updates.

- If this stream provides updates for resource-ids R0 and R1, and if R1 depends on R0, then the ALTO Server SHOULD send an update for R1 as soon as possible after sending the update for R0. For example, when a Network Map changes, the ALTO Server SHOULD send update events for the dependent Cost Maps as soon as possible after the update event for the Network Map.

- When the client uses the Stream Controller to stop updates for one or more resources (Section 8), the ALTO Server MUST send a control event (Section 6.3) whose "remove" field has the client-ids of those resources. If the client uses the Stream Controller to terminate all active resources and close the stream, the server MUST send a control event whose "remove" field has the client-ids of all active resources.

7.6.3. Cross-Stream Consistency Requirements

If several clients create Update Streams for updates to the same resource, the server MUST send the same updates to all of them. However, the server MAY pack data items into different Merge Patch events, as long as the net result of applying those updates is the same.

For example, suppose two different clients create Update Streams for the same Cost Map, and suppose the ALTO 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 server MAY send a single Merge Patch event (with all three cost points) to one client, while sending three separate Merge Patch events (with one cost point per event) to the other client.

A 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 data.

8. Update Stream Controller

An Update Stream Controller allows a client to remove resources from the set of resources that are monitored by an Update Stream, or add additional resources to that set. The controller also allows a client to gracefully shutdown an Update Stream.

The Stream Controller is not obtained from the ALTO Server's IRD. Instead, when a client requests a new Update Stream, the server creates a new controller for that stream, and sends its URI to the client as the first event in the Update Stream (Section 7.6.2).

As described below, each control request adds resources to the set of monitored resources, or removes previously added resources, or does both. Each control request is a separate HTTP request; the client MAY NOT stream multiple control requests in one HTTP request. However, if the client and server support HTTP Keep-Alive ([RFC7230]), the client MAY send multiple HTTP requests on the same TCP/IP connection.

8.1. URI

The URI for a Stream Controller, by itself, MUST uniquely specify the Update Stream instance which it controls. The 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, a server MUST NOT re-use a controller URI once the associated Update Stream has been closed.

The client MUST evaluate a non-absolute controller URI (for example, a URI without a host, or with a relative path) in the context of the URI used to create the Update Stream. The controller's host MAY be different from the Update Stream's host.

It is expected that the server will assign a unique stream id to each Update Stream instance, and will embed that id in the associated Stream Controller URI. However, the exact mechanism is left to the server. Clients MUST NOT attempt to deduce a stream id from the controller URI.

To prevent an attacker from forging a Stream Controller URI and sending bogus requests to disrupt other Update Streams, Stream Controller URIs SHOULD contain sufficient random redundency to make it difficult to guess valid URIs.

8.2. Media Type

An ALTO Stream Controller response does not have a specific media type. If a request is successful, the server returns an HTTP "204 No Content" response. If a request is unsuccessful, the server returns an ALTO error response (Section 8.5.2 of [RFC7285])

8.3. HTTP Method

An ALTO Update Stream Controller request uses the POST method.

8.4. Accept Input Parameters

A Stream Controller accepts the same input media type and input parameters as the Update Stream Service (Section 7.3). The only difference is that a Stream Controller also accepts the "remove" field.

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

A client MAY use the "add" field to add additional resources. However, the client MUST assign a unique client-id to each resource. Client-ids MUST be unique over the lifetime of this Update Stream: a client MUST NOT re-use a previously removed client-id.

If a request has any error, the server MUST NOT add or remove any resources from the associated Update Stream. In particular,

- Each "add" request must satisfy the requirements in Section 7.3. If not, the server MUST return the error response defined in Section 7.3.

- As described in Section 7.6.2, for each "add" request, the ALTO Server MUST send a full-replacement update event for that resource before sending any incremental updates. The only exception is for a GET-mode resource with a version tag. In this case the server MAY omit the full-replacement event for that resource if the "tag" field the client provided matches the server’s current version.

- The server MUST return an E_INVALID_FIELD_VALUE error if a client-id in the "remove" field was not added in a prior request. Thus it is illegal to "add" and "remove" the same client-id in the same request. However, it is legal to remove a client-id twice.

- The server MUST return an E_INVALID_FIELD_VALUE error if a client-id in the "add" field has been used before in this stream.

- The server MUST return an E_INVALID_FIELD_VALUE error if the request has a non-empty "add" field and a "remove" field with an empty list of client-ids (to replace all active resources with a new set, the client MUST explicitly enumerate the client-ids to be removed).

- If the associated Update Stream has been closed, the server MUST return either an ALTO E_INVALID_FIELD_VALUE error, or else an HTTP error, such as "404 Not Found".

8.5. Capabilities & Uses

None (Stream Controllers do not appear in the IRD).

8.6. Response

If a request is successful, the server returns an HTTP "204 No Content" response with no data. If there are any errors, the server MUST return the appropriate error code, and MUST NOT add or remove any resources from the Update Stream. Thus control requests are atomic: they cannot partially succeed.

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

Whenever a server removes resources as a result of a Stream Controller request, the server MUST send the corresponding "remove" Control Events (Section 6.3) on the Update Stream. If one control request removes several resources, the server MAY send one Control Event for all those resources, or a separate event for each removed resource, or any combination thereof.

9. Examples

9.1. Example: Simple Network and Cost Map Updates

Here is an example of a client’s request and the server’s immediate response, using the Update Stream resource "update-my-costs" defined in the IRD in Section 9.4. The client requests updates for the Network Map and "routingcost" Cost Map, but not for the "hopcount" Cost Map. The client uses the server’s resource-ids as the client-ids. Because the client does not provide a "tag" for the Network

Map, the server must send a full update for the Network Map as well as for the Cost Map. The client does not set "incremental-updates" to "false", so it defaults to "true". Thus server will send Merge Patch updates for the Cost Map, but not for the Network Map, because this Update Stream resource does not provide incremental updates for 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: ###

{ "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: "http://alto.example.com/updates/streams/3141592653589"}

event: application/alto-networkmap+json,my-network-map
data: { ... full Network Map message ... }

event: application/alto-costmap+json,my-routingcost-map
data: { ... full routingcost Cost Map message ... }
```

After sending those events immediately, the ALTO Server will send additional events as the maps change. For example, the following represents a small change to the Cost Map:

```
event: application/merge-patch+json,my-routingcost-map
data: {"cost-map": {"PID1" : {"PID2" : 9}}}
```

If a major change to the Network Map occurs, the ALTO Server MAY choose to send full Network and Cost Map messages rather than Merge Patch messages:

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

event: application/alto-costmap+json,my-routingcost-map
data: { ... full Cost Map message ... }
```

9.2. Example: Advanced Network and Cost Map Updates

This example is similar to the previous one, except that the 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 server is not required to send the full Network Map update event at the beginning of the stream. In this example, the client uses the client-ids "net", "routing" and "hops" for those resources. The ALTO Server sends the stream id 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: ###

{ "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: "http://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 client wishes to stop receiving updates for the "hopcount" Cost Map, the client can send a "remove" request on the Stream Controller 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: ###

{
  "remove": [ "hops" ]
}


HTTP/1.1 204 No Content
Content-Length: 0

    (stream closed without sending data content)
```

The ALTO Server sends a "remove" control event on the original request stream to inform the client that updates are stopped for that resource:

```
event: application/alto-updatestreamcontrol+json
```

```
data: { "remove": ["hops"] }
```

If the 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: ###

{
  "remove": [ ]
}


HTTP/1.1 204 No Content
Content-Length: 0

    (stream closed without sending data content)
```

The ALTO Server sends a final "remove" control event on the original request stream to inform the client that all updates are stopped, and then closes the stream:

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

    (server closes stream)
```

9.3. Example: Endpoint Property Updates

As another example, here is how a client can request updates for the property "priv:ietf-bandwidth" for one set of endpoints, and "priv:ietf-load" for another. The ALTO Server immediately sends full-replacement messages with the property values for all endpoints. After that, the server sends update events 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: ###

{ "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: "http://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 client needs the "bandwidth" property for additional endpoints, the client can send a "add" request on the Stream Controller 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: ###

{ "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 ALTO Server sends full replacement events for the two new resources, followed by incremental updates 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: }
```

9.4. IRD Example

Here is an example of an IRD that offers two Update Stream services. The first provides updates for the Network Map, the "routingcost" and "hopcount" Cost Maps, and a Filtered Cost Map resource. The second Update Stream provides updates to the Endpoint Properties service.

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 12.1, the Update Stream provides updates for the second, but not the first.

```
  "my-network-map": {
```

```
      "uri": "http://alto.example.com/networkmap",
      "media-type": "application/alto-networkmap+json",
    },
    "my-routingcost-map": {
      "uri": "http://alto.example.com/costmap/routingcost",
      "media-type": "application/alto-costmap+json",
      "uses": ["my-networkmap"],
      "capabilities": {
        "cost-type-names": ["num-routingcost"]
      }
    },
    "my-hopcount-map": {
      "uri": "http://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": "http://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": "http://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": "http://alto.example.com/properties",
      "media-type": "application/alto-endpointprops+json",
      "accepts": "application/alto-endpointpropparams+json",
      "capabilities": {
        "prop-types": ["priv:ietf-bandwidth"]
      }
    },
    "update-my-costs": {
```

```
      "uri": "http://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-update-media-types": {
          "my-routingcost-map": application/merge-patch+json",
          "my-hopcount-map": "application/merge-patch+json"
        }
      }
    },
    "update-my-props": {
      "uri": "http://alto.example.com/updates/properties",
      "media-type": "text/event-stream",
      "uses": [ "my-props" ],
      "accepts": "application/alto-updatestreamparams+json",
      "capabilities": {
        "incremental-update-media-types": {
          "my-props": "application/merge-patch+json"
        }
      }
    }
```

10. Client Actions When Receiving Update Messages

In general, when a client receives a full-replacement update message for a resource, the client should replace the current version with the new version. When a client receives a Merge Patch update message for a resource, the 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 a client can do that. We will 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 ALTO Server MUST send the Network Map update message before sending the updates for the dependent Cost Maps (see Section 7.6.2).

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 client receives the update messages for all dependent Cost Maps. The client then applies all Network and Cost Map updates atomically.

Alternatively, the client MAY update the Network Map immediately. In this case, the client MUST mark each dependent Cost Map as temporarily invalid, and MUST NOT use that map until the client receives a Cost Map update message with the new Network Map version tag. Note that the client MUST NOT delete the Cost Maps, because the server may send Merge Patch update messages.

The ALTO Server SHOULD send updates for dependent resources in a timely fashion. However, if the client does not receive the expected updates, the client MUST close the Update Stream connection, discard the dependent resources, and reestablish the Update Stream. The 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 ALTO Server will not re-send that resource.

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

11. Design Decisions and Discussions

11.1. HTTP/2 Server-Push

HTTP/2 ([RFC7540]) provides a Server Push facility. Although the name implies that it might be useful for sending asynchronous updates from the server to the client, in reality Server Push is not well suited for that task. To see why it is not, here is a quick summary of HTTP/2.

HTTP/2 allows a client and server to multiplex many HTTP requests and responses over a single TCP connection. The requests and responses can be interleaved on a block by block basis, avoiding the head-of-line blocking problem encountered with the Keep-Alive mechanism in HTTP/1.1. Server Push allows the server to send a resource (an image, a CSS file, a javascript file, etc.) to the client before the client explicitly requests it. A server can only push cacheable GET-mode resources. By pushing a resource, the server implicitly tells the client, "Add this resource to your cache, because a resource you have requested needs it."

One approach for using Server Push for ALTO updates is for the server

to send each update event as a separate Server Push item, and let the client apply those updates as they arrive. Unfortunately there are several problems with that approach.

First, HTTP/2 does not guarantee that pushed resources are delivered to the client in the order they were sent by the client, so each update event would need a sequence number, and the client would have to re-sequence them.

Second, an HTTP/2-aware client library will 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.

But the third problem is the most significant: 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 which the client does not know to request.

Thus we do not believe HTTP/2 Server Push is suitable for delivering asynchronous updates. Hence we have chosen to base ALTO updates on HTTP/1.1 and SSE.

11.2. Not Allowing Stream Restart

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

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

Furthermore, this is unlikely to be a problem in practice. Clients who want continuous updates for large resources, such as full Network and Cost Maps, are likely to be things like P2P trackers. These 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 ALTO 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 which we expect is very rare. We believe that complexity is not worth the benefit.

The Update Stream service does allow the client to specify the tag of the last received version of any tagged resource, and if that is still current, the server need not retransmit the full resource. Hence 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.

11.3. Is Incremental Update Useful for Network Maps?

It is not clear whether incremental updates (that is, Merge Patch updates) are useful for Network Maps. For minor changes, such as moving a prefix from one PID to another, they can be useful. But more involved changes to the Network Map are likely to be "flag days": they represent a completely new Network Map, rather than a simple, well-defined change.

At this point we do not have sufficient experience with ALTO deployments to know how frequently Network Maps will change, or how extensive those changes will be. For example, suppose a link goes down and the network uses an alternative route. This is a frequent occurrence. If an ALTO Server models that by moving prefixes from one PID to another, then Network Maps will change frequently. However, an ALTO Server might model that as a change in costs between PIDs, rather than a change in the PID definitions. If a server takes that approach, simple routing changes will affect Cost Maps, but not Network Maps.

So while we allow a server to use Merge Patch on Network Maps, we do not require the server to do so. Each server may decide on its own whether to use Merge Patch for Network Maps.

This is not to say that Network Map updates are not useful. Clearly Network Maps will change, and update events are necessary to inform

clients of the new map. Further, there maybe another incremental update encoding that is better suited for updating Networks Maps; see the discussions in the next section.

11.4. Other Incremental Update Message Types

Other JSON-based incremental update formats have been defined, in particular JSON Patch ([RFC6902]). The update events defined in this document have the media-type of the update data. JSON Patch has its own media type ("application/json-patch+json"), so this update mechanism could easily be extended to allow servers to use JSON Patch for incremental updates.

However, we think that JSON Merge Patch is clearly superior to JSON Patch for describing incremental updates 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; we see no advantage to allowing a server to use JSON Patch for those resources.

The case is not as clear for incremental updates to Network Maps. For example, suppose a prefix moves from one PID to another. JSON Patch could encode that as a simple insertion and deletion, while 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 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, clients would have to retain additional, otherwise unnecessary, data.

However, it is entirely possible that JSON Patch will be appropriate for describing incremental updates to new, as yet undefined ALTO resources. In this case, the extensions defining those new resources can use the update framework defined in this document, but recommend using JSON Patch, or some other method, to describe the incremental changes.

12. Miscellaneous Considerations

12.1. Considerations For Updates To Filtered Cost Maps

If an Update Stream provides updates to a Filtered Cost Map which allows constraint tests, then a client MAY request updates to a Filtered Cost Map request with a constraint test. In this case, when a cost changes, the server MUST send an update if the new value

satisfies the test. If the new value does not, whether the server sends an update depends on whether the previous value satisfied the test. If it did not, the server SHOULD NOT send an update to the client. But if the previous value did, then the server MUST send an update with a "null" value, to inform the client that this cost no longer satisfies the criteria.

An ALTO Server can avoid such issues by offering Update Streams only for Filtered Cost Maps which do not allow constraint tests.

12.2. Considerations For Incremental 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 a server to offer incremental updates for ordinal mode cost maps, server implementors should be aware that incremental updates for ordinal costs are more complicated than for numerical costs, and clients should be aware that small changes may result in large updates.

An ALTO Server can avoid this complication by only offering full replacement updates for ordinal cost maps.

12.3. Considerations Related to SSE Line Lengths

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 full cost maps as single events, and a cost map may involve megabytes, if not tens of megabytes, of text. This has implications for both the ALTO Server and Client.

First, SSE clients might not be able to handle a multi-megabyte data "line". Hence it is RECOMMENDED that an ALTO server limit data lines to at most 2,000 characters.

Second, some SSE client packages read all the data for an event into memory, and then present it to the client as a single character array. However, a client computer 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 which presents the event data in manageable chunks, so the client can parse the cost map incrementally and store the underlying data in a more compact format.

13. Security Considerations

13.1. Denial-of-Service Attacks

Allowing persistent update stream connections enables a new class of Denial-of-Service attacks. A client might create an unreasonable number of update stream connections, or add an unreasonable number of client-ids to one update stream. To avoid those attacks, an ALTO Server MAY choose to limit the number of active streams, and reject new requests when that threshold is reached. A server MAY also chose to limit the number of active client-ids on any given stream, or limit the total number of client-ids used over the lifetime of a stream, and reject any stream control request which would exceed those limits. In these cases, the server SHOULD return the HTTP status "503 Service Unavailable".

While this technique prevents Update Stream DoS attacks from disrupting an ALTO Server's other services, it does make it easier for a DoS attack to disrupt the Update Stream service. Therefore a server may prefer to restrict Update Stream services to authorized clients, as discussed in Section 15 of [RFC7285].

Alternatively an ALTO Server MAY return the HTTP status "307 Temporary Redirect" to redirect the client to another ALTO Server which can better handle a large number of update streams.

13.2. Spoofed Control Requests

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

13.3. Privacy

This extension does not introduce any privacy issues not already present in the ALTO protocol.

14. IANA Considerations

This document defines two new media-types, "application/ alto-updatestreamparams+json", as described in Section 7.3, and "application/alto-updatestreamcontrol+json", as described in

Section 6.3. All other media-types used in this document have already been registered, either for ALTO or JSON Merge Patch.

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 [RFC7159].

Security considerations: Security considerations relating to the generation and consumption of ALTO Protocol messages are discussed in Section 13 of this document and Section 15 of [RFC7285].

Interoperability considerations: This document specifies format of conforming messages and the interpretation thereof.

Published specification: Section 7.3 of this document.

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

Additional information:

Magic number(s): n/a

File extension(s): This document uses the mime 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).

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 [RFC7159].

Security considerations: Security considerations relating to the generation and consumption of ALTO Protocol messages are discussed in Section 13 of this document and Section 15 of [RFC7285].

Interoperability considerations: This document specifies format of conforming messages and the interpretation thereof.

Published specification: Section 6.3 of this document.

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

Additional information:

Magic number(s): n/a

File extension(s): This document uses the mime 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).

15. References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, BCP 14, March 1997.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, March 2010.

[RFC6902] Bryan, P. and M. Nottingham, "JavaScript Object Notation (JSON) Patch", RFC 6902, April 2013.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[RFC7285] Almi, R., Penno, R., Yang, Y., Kiesel, S., Previdi, S., Roome, W., Shalunov, S., and R. Woundy, "Application-Layer Traffic Optimization (ALTO) Protocol", RFC 7285, September 2014.

[RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

[RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[RFC7232] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014.

[RFC7233] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, June 2014.

[RFC7234] Fielding, R., Nottingham, M., and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014.

[RFC7235] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014.

[RFC7396] Hoffman, P. and J. Snell, "JSON Merge Patch", RFC 7396, October 2014.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, May 2015.

[SSE] Hickson, I., "Server-Sent Events (W3C)", W3C Recommendation 03 February 2015, February 2015.

Appendix A. Acknowledgments

Thank you to Xiao Shi (Yale University) for his contributions to an earlier version of this document.

Authors' Addresses

Wendy Roome
Nokia Bell Labs
600 Mountain Ave, Rm 3B-324
Murray Hill, NJ 07974
USA

Phone: +1-908-582-7974
Email: wendy.roome@nokia.com

Y. Richard Yang
Tongji/Yale University
51 Prospect St
New Haven CT
USA

Email: yang.r.yang@gmail.com