ALTO WG
Internet-Draft
Intended status: Standards Track
Expires: April 1, 2016

W. Roome
Alcatel-Lucent
Y. Yang
Tongji/Yale University

September 29, 2015

ALTO Incremental Updates Using Server-Sent Events (SSE)
draft-ietf-alto-incr-update-sse-01

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

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

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

Internet-Drafts are draft documents valid for a maximum of six months

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

This Internet-Draft will expire on April 1, 2016.

Copyright Notice

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 3 defines the update events, and Section 4 defines the format of the incremental update messages. Section 5 defines the new Update Stream Service, Section 6 describes how a client should handle incoming updates, and Section 7 gives an example of the Information Resource Directory (IRD) for an ALTO Server that offers a comprehensive set of Update Services. Section 8 discusses the design decisions behind this update mechanism. 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 ([RFC7386]) 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, although the set should be closed under the ALTO resource dependency relationship (i.e., the "uses" relationship). Thus the Update Stream for a Cost Map should also provide updates for the Network Map upon 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 connection remains open, and the server continues to send updates, until either the client or the server closes it. 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 -00

- Defined a "stream id". The server defines a unique id for each update stream, and sends it as the first event (Section 6.6.2).

- Revised the input parameter syntax to allow "stop-updates" requests as well as "start-update" requests (Section 6.3 and Section 6.3.2).

- Said a server MAY send an update even if the value does not actually change (Section 6.6).
- Added discussions related to ordinal-mode cost maps (Section 6.8) and the line length of SSE events (Section 6.9).
- Expanded the Security Considerations (Section 10).

4. Update Events

4.1. Overview of 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 ([RFC2616]) connection to the server, and keeps the connection open. The server continually sends messages. Messages are delimited by two new-lines (this is a slight simplification; see [SSE] for details). Each line is of the form "field-name: string value". 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. Note that the server may "chunk" the returned data (see [RFC2616]); for simplicity, we have omitted those details.

```
GET /stream HTTP/1.1
Host: example.com
Accept: text/event-stream


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.

4.2. ALTO Update Events

In the events defined in this document, the data field is a JSON object. That object is either a complete specification of an ALTO resource, or else a JSON Merge Patch object describing changes to apply to an ALTO resource. 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. The data objects in Merge Patch messages are defined by [RFC7386].

To indicate whether the data is a full-replacement or a Merge Patch object, in our update messages, the SSE "event" field has two sub-fields: the resource-id of an ALTO resource, and the media-type of the JSON message in the data field. The media-types for full-replacement messages are defined by [RFC7285], and include "application/alto-networkmap+json" for Network Map messages and "application/alto-costmap+json" for Cost Map messages. The media-type for a JSON Merge Patch message is "application/merge-patch+json", and is defined by [RFC7386]. An extension document may introduce other media-types to indicate new types of update messages.

Specifically, the two sub-fields of the event field are encoded as:

```
resource-id , media-type
```

Note that a comma (character code 0x2c) is allowed in ALTO resource-ids, but not in media-type names. Hence when parsing the SSE event field to obtain the two sub-fields, a client MUST split the string on the last comma.

This document does not use the SSE "id" field.

Figure 2 shows some examples of ALTO update events:

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

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

event: my-routingcost-map,application/merge-patch+json
data: { ... Merge Patch update for the Cost Map ... }
```

Figure 2: Examples of ALTO update events.

5. Incremental Update Message Format

5.1. Overview of JSON Merge Patch

The following is a non-normative summary of JSON Merge Patch. See [RFC7386] 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 [RFC7386]:

```
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 "2000::/3" to "PID1", deletes "PID2", and assigns a new "tag" to the Network Map:

```
{
  "meta" : {
    "vtag" : {
      "tag" : "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
    }
  },
  "network-map": {
    "PID1" : {
      "ipv6" : [ "2000::/3" ]
    },
    "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" : [ "2000::/3" ]
    },
    "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. Update Stream Service

An Update Stream Service returns a stream of SSE messages, as defined in Section 4.2. An Update Stream resource can be used to request a new update stream, or to stop updates for a previously requested stream.

6.1. Media Type

The media type of an ALTO Update Stream resource is "text/event-stream".

6.2. HTTP Method

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

6.3. Accept Input Parameters

An ALTO Client supplies the Update Stream resource parameters by specifying media type "application/alto-updatestreamparams+json" with an HTTP POST body containing a JSON Object of type UpdateStreamReq, where:

```
object {
   [StartUpdatesReq   start-updates;]
   [String            stop-updates<0..*>;]
   [String            stream-id;]
} UpdateStreamReq;

object-map {
   ResourceId -> ResourceUpdateReq;
} StartUpdatesReq;

object {
   [String      tag;]
   [Boolean     incremental-updates;]
   [Object      input;]
} ResourceUpdateReq;
```

A client uses the "start-updates" field to request a new update stream, and the "stop-updates" field to stop updates for some or all of the resources in a previously established update stream. One or the other field is required.

6.3.1. Parameters for "start-updates" Requests

The value of the "start-updates" field is a JSON Object of type StartUpdatesReq. The keys of this object are the resource-ids of the resources for which the client wants updates. Each resource-id MUST be one of those in the Update Streams's "uses" list (see Section 6.5). The ResourceUpdateReq values give additional parameters for the updates for each resource.

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 update events.

If the client wants to receive updates for a resource, but does not need to set any of the sub-fields described below, the client MUST provide an entry for that resource-id whose value is an empty JSON Object (e.g., "{}").

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 6.4). 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 ALTO Server SHOULD send the full-replacement message soon after the change, although the server MAY wait until more changes are available. 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-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 "tag" part of the resource's version tag. If that version is still current, the ALTO Server SHOULD omit sending a full replacement update at the start of the stream (see Section 6.6.2).

If the resource-id 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 update events. If the inputs for several POST-mode resources are missing or invalid, the server MUST pick one error response and return it.

6.3.2. Parameters for "stop-updates" Requests

A client uses the "stop-updates" field to stop updates from a previously established stream. The value is a JSON Array of resource ids requested in that stream. The server MUST stop sending updates for those resources. If "stop-updates" is a zero-length array, the server MUST stop sending updates for all remaining resources. When no resources are left, the server MUST close the update stream. The server MUST ignore any resource-ids which have already been stopped, or which were not in the corresponding update-stream request.

"stream-id" is a unique identifier assigned when the stream was created (see Section 6.6), and is required for a "stop-updates" request. If the string does not match the id of an active update

stream, the server MUST return an E_INVALID_FIELD_VALUE error response (see Section 8.5.2 of [RFC7285]).

SSE is a one-way protocol; it does not define a mechanism for clients to send data other than the initial request. Furthermore, the persistent-connection feature of HTTP 1.1 ([RFC2616]) is not compatible with SSE, because when an SSE request is complete, the server closes the underlying HTTP connection.

Therefore when sending a "stop-updates" request, the client MUST create a new HTTP connection to the ALTO server, and MUST send the "stop-updates" request on that new connection. The client MUST NOT send a "stop-updates" request on the connection on which it receives SSE updates. The "stream-id" field identifies the stream to be stopped.

A client MAY terminate an update stream by simply closing it. However, it may take some time for the server to recognize that the stream is closed, and the server may interpret that as an error. A "stop-updates" request allows a client to stop an update stream quickly and cleanly. It also allows a client to stop updates for some resources, but continue getting updates for others.

6.4. 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 4.2, JSON Merge Patch is the only incremental update event type defined by this document. However future extensions may define other types of incremental updates.

6.5. Uses

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

This set can include any subset of the resources proved by the ALTO Server, 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.

6.6. Response

The response depends on the input parameters sent by the client. If the client sends a "stop-updates" request, unless there is an error, the server closes the stream immediately without sending any response.

For a "start-updates" request, the response is a stream of SSE update events. Section 4.2 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.

6.6.1. 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.

6.6.2. Event Sequence Requirements

- The first event MUST provide the stream id which the server assigns to this update stream. The event's resource-id is that of the Update Stream resource, and the media-type is the same as the input parameters for an Update Stream request ("application/

alto-updatestreamparams+json", as defined in Section 6.3). The data value is a JSON Object of that type, and the "stream-id" field has the id assigned to this stream. A stream id MUST be no more than 64 characters, and MUST NOT contain any character below U+0021 or above U+007E. Because they are used as authentication tokens, stream ids SHOULD NOT be predictable.

- 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: 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 a client sends a "stop-updates" request for this stream, the ALTO Server MUST send a confirmation event on the stream. The event's resource-id is that of the Update Stream resource, and the media-type is the same as the input parameters for an Update Stream request ("application/alto-updatestreamparams+json", as defined in Section 6.3). The data is the "stop-updates" request sent by the client. This informs the client that the server will not send subsequent updates for those resources. If the "stop-updates" array is zero-length, updates for all remaining resources will be closed. When there are no more resources left, the server MUST close the stream.

6.6.3. Cross-Stream Consistency Requirements

If several distinct Update Stream resources offer updates for the same resource-id, the ALTO Server MUST send the same update data on all of those Update Streams. Similarly, the server MUST send the same updates to all clients connected to the that stream. 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 open the same Cost Map Update Stream, 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.

6.7. 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.

6.8. 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.

6.9. 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 a line-at-a-time. However, an Update Stream sends full cost maps as single events, and a cost map may involve megabytes, of 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 when sending a full network map or cost map, an ALTO server SHOULD insert a new-line character periodically. Approximately every 2,000 characters should be sufficient for most SSE clients.

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.

6.10. 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 8. The client requests updates for the Network Map and "routingcost" Cost Map, but not for the "hopcount" Cost Map. 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.

Note that the server may "chunk" the returned data (see [RFC2616]); for simplicity, we have omitted those details.

```
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: ###

{ "start-updates": {
    "my-network-map": {},
      "my-routingcost-map": {}
    }
}


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

event: update-my-costs,application/alto-updatestreamparams+json
data: {"stream-id":
data: "3141592653589793238462643383279502884197169399375105820"}

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

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

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

6.11. 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 does not send the full Network Map update event at the beginning of the stream. 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: ###

{ "start-updates": {
    "my-network-map": {
      "tag": "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
    },
    "my-routingcost-map": {}
    "my-hopcount-map": {}
  }
}


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

event: update-my-costs,application/alto-updatestreamparams+json
data: {"stream-id":
data: "097494459230781640628620899862803482534211706798214 8"}

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

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

   (pause)

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

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

If the client wishes to stop receiving updates for the "hopcount" Cost Map, the client can send a "stop-updates" request on a different HTTP connection:

```
  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: ###

  {
    "stream-id":
      "097494459230781640628620899862803482534211706798214 8",
    "stop-updates": [ "my-hopcount-map" ]
  }


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

      (stream closed without sending data content)
```

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

```
  event: update-my-costs,application/alto-updatestreamparams+json
  data: {"stream-id":
  data: "097494459230781640628620899862803482534211706798214 8",
  data: "stop-updates": [ "my-hopcount-map" ]
  data: }
```

If the client no longer needs any updates, and wishes to shut the Update Stream down gracefully, the client can send a "stop-updates" request with an empty array:

```
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: ###

{
  "stream-id":
    "0974944592307816406286208998628034825342117067982148",
  "stop-updates": []
}


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

    (stream closed without sending data content)
```

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

```
event: update-my-costs,application/alto-updatestreamparams+json
data: {"stream-id":
data: "0974944592307816406286208998628034825342117067982148",
data: "stop-updates": []
data: }

    (server closes stream)
```

6.12. Example: Endpoint Property Updates

As another example, here is how a client can request updates for the property "priv:ietf-bandwidth" for a set of endpoints. The ALTO Server immediately sends a full-replacement message 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: ###

{ "start-updates":
   { "my-props": {
```

```
            "input": {
               "properties" : [ "priv:ietf-bandwidth" ],
               "endpoints" : [
                  "ipv4:1.0.0.1",
                  "ipv4:1.0.0.2",
                  "ipv4:1.0.0.3"
               ]
            }
         }
      }
   }


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

   event: update-my-props,application/alto-updatestreamparams+json
   data: {"stream-id":
   data: "086513282306647093844609550582231725359408128481117 45"}

   event: my-props,application/alto-endpointprops+json
   data: { "endpoint-properties": {
   data:     "ipv4:1.0.0.1" : { "priv:ietf-bandwidth": "13" },
   data:     "ipv4:1.0.0.2" : { "priv:ietf-bandwidth": "42" },
   data:     "ipv4:1.0.0.3" : { "priv:ietf-bandwidth": "27" }
   data:  } }

      (pause)

   event: my-props,application/merge-patch+json
   data: { "endpoint-properties":
   data:   {"ipv4:1.0.0.1" : {"priv:ietf-bandwidth": "3"}}
   data: }

      (pause)

   event: my-props,application/merge-patch+json
   data: { "endpoint-properties":
   data:   {"ipv4:1.0.0.3" : {"priv:ietf-bandwidth": "38"}}
   data: }
```

7. 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 6.6.2).

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.

8. 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 6.7, 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"
        }
      }
    }
```

9. Design Decisions and Discussions

9.1. HTTP2 Server-Push

An alternative would be to use HTTP 2 Server-Push [RFC7540], instead of SSE over HTTP 1.1, as the transport mechanism for update messages. That would have several advantages: HTTP 2 Server-Push is designed to allow a server to send asynchronous messages to the client, and HTTP library packages should make it simple for servers to send those asynchronous messages, and for clients to receive them.

The disadvantage is HTTP 2 is a new protocol, and it is considerably more complicated than HTTP 1.1. While there is every reason to

expect that HTTP library packages will eventually support HTTP 2, we do not want to delay deployment of an ALTO incremental update mechanism until that time.

Hence we have chosen to base ALTO updates on HTTP 1.1 and SSE. When HTTP 2 support becomes ubiquitous, a future extension of this document may define updates via HTTP 2 Server-Push.

9.2. 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.

9.3. 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 discussions in the next section.

9.4. 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.

10. Security Considerations

10.1. Denial-of-Service Attacks

Allowing persistent update stream connections enables a new class of Denial-of-Service attacks. An ALTO Server MAY choose to limit the number of active streams, and reject new requests when that threshold is reached. In this case 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.

10.2. Spoofed "stop-updates" Requests

An outside party which can read the update stream response can obtain the stream-id and use that to send a fraudulent "stop-updates" request, thus disabling updates for the valid client. This can be

avoided by encrypting the stream (see Section 15 of [RFC7285]). Also, the ALTO Server sends any "stop-updates" requests on the update stream, so the valid client can detect unauthorized "stop-update" requests.

10.3. Privacy

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

11. IANA Considerations

This document defines a new media-type, "application/alto-updatestreamparams+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 10 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).

12. References

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Burners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[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.

[RFC7386] Hoffman, P. and J. Snell, "JSON Merge Patch", RFC 7386, 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
Alcatel-Lucent/Bell Labs
600 Mountain Ave, Rm 3B-324
Murray Hill, NJ 07974
USA

Phone: +1-908-582-7974
Email: w.roome@alcatel-lucent.com

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

Email: yang.r.yang@gmail.com