ALTO New Transport using HTTP/2

draft-schott-alto-new-transport-01

Roland Schott
Y. Richard Yang
Kai Gao
Jensen Zhang

March 23, 2022

IETF 113
Motivation and requirements
ALTO/H2 design
Discussions and open issues
Motivation

- ALTO base protocol [RFC7285] is an HTTP/1.x client-pull protocol
- ALTO/SSE [RFC8895] adds incremental server push using Server-Sent-Event, but is based on HTTP/1.x
  - Need additional control connection
  - Updates must be serialized
- RFC8895 IESG review
  - Consider HTTP/2

---

ALTO SSE Example

```
+--------+ +--------+ 1. init request +--------+
|         |         | <---------->
<p>| 3.add/  | 1'. control uri |
| remove  |         |         |</p>
<table>
<thead>
<tr>
<th>resource</th>
<th>Stream</th>
<th>Update</th>
</tr>
</thead>
</table>
| Control| private|Stream| 2a. data update|Client|--
| Server | Stream| Update| messages |      |
| --------|--------|-------|----------|      |
| response|        | Server| 2b.control update|    |
|         |        |       | messages     |      |
| --------|--------|-------|-------------|      |
```
ALTO/H2 Design Requirements

• From ALTO base protocol [RFC 7285]
  – R0: Client can request any ALTO resource using the connection, just as using ALTO base protocol using HTTP/1.x

• From ALTO SSE [RFC 8895]
  – R1: Client can request the addition (start) of incremental updates to a resource
  – R2: Client can request the deletion (stop) of incremental updates to a resource
  – R3: Server can signal to the client the start or stop of incremental updates to a resource
  – R4: Server can choose the type of each incremental update encoding, as long as the type is indicated to be acceptable by the client

• From ALTO base framework [RFC 7285]
  – R5: Design follows basic HTTP Representational State Transfer architecture if possible
    • Can use only a limited number of verbs (GET, POST, PUT, DELETE, HEAD)
  – R6: Design takes advantage of HTTP/2 design features such as parallel transfer and respects HTTP/2 semantics [PUSH_PROMISE]

• Allow flexible deployment
  – R7: Capability negotiation
ALTO/H2 Design Requirements addressed by daft

- From ALTO base protocol [RFC 7285]
  - R0: Client can request any ALTO resource using the connection, just as using ALTO base protocol using HTTP/1.x

- From ALTO SSE [RFC 8895]
  - R1: Client can request the addition (start) of incremental updates to a resource
  - R2: Client can request the deletion (stop) of incremental updates to a resource
  - R3: Server can signal to the client the start or stop of incremental updates to a resource
  - R4: Server can choose the type of each incremental update encoding, as long as the type is indicated to be acceptable by the client

- From ALTO base framework [RFC 7285]
  - R5: Design follows basic HTTP Representational State Transfer architecture if possible
    - Can use only a limited number of verbs (GET, POST, PUT, DELETE, HEAD)
  - R6: Design takes advantage of HTTP/2 design features such as parallel transfer and respects HTTP/2 semantics [PUSH_PROMISE]

- Allow flexible deployment
  - R7: Capability negotiation
Outline

- Motivation and requirements
- ALTO/H2 design
  - Overview
ALTO/H2 Transport Information Structure

ALTO Server

**Static resource such as NetworkMap**

**Filterable resource such as FilteredCostMap**

Information resource

- ir1
- ir2

Transport queue

- tq1
- tq2
- tq3

Incremental updates queue

- tq1/uq
- tq1/rs

Receiver set

- Client 1
- Client 2
- Client 3

Adapt ALTO SSE to HTTP/2:
Single HTTP connection between Server and client

A single HTTP/2 connection between each client and the server

Static resource such as NetworkMap

Filterable resource such as FilteredCostMap

IETF 113: ALTO New Protocol (Transport)
ALTO/H2 Transport Information Structure

- Client opens a connection to the server
- Client opens/identifies a transport queue tq
  - Client requests transport queue status
  - Client requests an element in the message queue
  - Client becomes a receiver
  - Client receives push updates
- Client closes the transport queue
- Client closes connection
Outline

- Motivation and requirements
- ALTO/H2 design
  - Overview
  - Transport queue
Transport Queue

- **Basic operations (CRUD):** create, read (get status), delete
- **Client creates transport queue**
  - POST to transport queues path
    - Request reuses ALTO/SSE input
      - HTTP :method=post with AddUpdateReq [RFC8895]
    - Response
      - <transport-queue>

```json
object {
    ResourceID resource-id;
    [JSONString tag;]
    [Boolean incremental-changes;]
    [Object input;]
} AddUpdateReq;
```
Transport Queue Example (Create)

- Client -> Server request

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

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

- Server -> Client response

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

DATA
- END_STREAM
{"tq": "/tqs/2718281828459"}
```
Transport Queue

- Basic operations (CRUD): create, read (get status), delete
- Client creates transport queue
  - POST to transport queues path
    - Request reuses ALTO/SSE input
      - HTTP :method=post with AddUpdateReq [RFC8895]
    - Response
      - <transport-queue>
- Client reads transport queue: GET <transport-queue>
- Client closes transport queue:
  - Explicit: DELETE <transport-queue>
    - Delete from local view (server may still maintain the transport queue for other client connections)
  - Implicit: Transport queue for a client is ephemeral: close of connection or stream deletes the transport queue from the client’s view --- when the client reconnects, the client MUST NOT assume that the queue is still valid
Transport Queue Example (Read)

• Client -> Server request

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

„uq“ = incremental updates queue
„rs“ = receiver set

• Server -> Client response

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

{ "uq": [
  {
    "seq": 101,
    "media-type": "application/alto-costmap+json",
    "tag": "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
  },
  {
    "seq": 102,
    "media-type": "application/merge-patch+json",
    "tag": "cdf0222x59740b0b2e3f8eb1d4785acd42231bfe"
  },
  {
    "seq": 103,
    "media-type": "application/merge-patch+json",
    "tag": "8eb1d4785acd42231bfecd0222x59740b0b2e3f",
    "link": "/tqs/2718281828459/snapshot/2e3f"
  },
  {
    "rs": ["self"]
  }
]  }
```
Outline

• Motivation and requirements
• ALTO/H2 design
  – Overview
  – Transport queue
  – Incremental updates queue
Incremental Updates Queue

- Incremental updates queue basic operations (CRUD): read (get status)
  - Client cannot create, update, or delete incremental updates queue directly---it is read only, and associated with transport queue automatically
  - Read:
    - Input: <tq>/uq
    - Response: updates queue state
    - Note
      - Server determines the state (window of history and type of each update) in the update queue [R4]
      - Read of updates queue status allows client to know
        - backlog status
        - workload to catch up (HEAD)
        - potential direct link

Request

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

Response data

```
DATA
- END_STREAM
 {
  ["seq": 101,
   "media-type": "application/alto-costmap+json",
   "tag": "a10ce8b059740b0b2e3f8eb1d4785acd42231bfe"
  ],
  ["seq": 102,
   "media-type": "application/merge-patch+json",
   "tag": "cdf0222x59740b0b2e3f8eb1d4785acd42231bfe"
  ],
  ["seq": 103,
   "media-type": "application/merge-patch+json",
   "tag": "8eb1d4785acd42231bferc0222x59740b0b2e3f",
   "link": "/tqs/2718281828459/snapshot/2e3f"
  ]
} 
```
Outline

- Motivation and requirements
- ALTO/H2 design
  - Overview
  - Transport queue
  - Incremental updates queue
  - Individual updates
Individual Incremental Updates

- Individual incremental updates operations (CRUD): pull read or push read
  - Client pull
    - GET <update-uri>
**Client Pull Example**

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

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

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

- Individual incremental updates operations (CRUD): pull read or push read
  - Client pull
    - GET <update-uri>
  - Server push
    - Initialization:
      - The first update pushed from the server to the client MUST be the later of the following two
        - The last independent update in the incremental updates queue
        - The following entry of the entry that matches the tag when the client creates the transport queue
      - The client MUST set SETTINGS_ENABLE_PUSH to be consistent
    - State: the server maintains the last entry pushed to the client and schedules next update push
      - Per client, connection state
    - Client MUST NOT cancel (RST_STREAM) a PUSH_PROMISE
      - To avoid complex server state management
Server Push Initialization Example

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

- Each pushed update is indicated first in a PUSH_PROMISE

Server send PUSH_PROMISE

```plaintext
Server -> client PUSH_PROMISE in stream 3

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

```
Server -> client content Stream 4

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

DATA
+ END_STREAM
{
  "meta": {
    "dependent-vtags": [{
      "resource-id": "my-network-map",
      "tag": "da65eca2eb7a0ce8b059740b0b2e3f8eb1d4785"
    }],
    "cost-type": {
      "cost-mode": "numerical",
      "cost-metric": "routingcost"
    },
    "vtag": {
      "resource-id": "my-routingcost-map",
      "tag": "3ee2cb7e8d63d9fab71b9b34cbf764436315542e"
    }
  },
  "cost-map": {
    "PID1": { "PID1": 1, "PID2": 5, "PID3": 10 },
    "PID2": { "PID1": 5, "PID2": 1, "PID3": 15 },
    "PID3": { "PID1": 20, "PID2": 15 }
  }
}
```
Outline

• Motivation and requirements
• ALTO/H2 design
  – Overview
  – Transport queue
  – Incremental updates queue
  – Individual updates
  – Receiver set
Receiver Set

• Receiver set operations (CRUD): read (get status), delete (self only)

• By default, a client can see only itself in the receiver set
  – Appearance of self in the receiver set (read does not return “not exists”) is an indication that push starts

• A client can delete itself (stops receiving push):
  – Explicit: DELETE <transport-queue>/rs/self
  – Implicit: Transport queue is connection ephemeral: close of connection or stream for the transport queue deletes the transport queue (from the view) for the client
Outline

• Motivation and requirements
• ALTO/H2 design
  – Overview
  – Transport queue
  – Incremental updates queue
  – Individual updates
  – Receiver set
  – Stream management
ALTO/H2 Stream Management: Objectives

- Objectives
  - Allow stream concurrency to reduce latency
  - Minimize the number of streams created
  - Enforce dependency among streams (so that if A depends on B, then A should be sent before B)
    - Encode dependency to enforce semantics (correctness)
ALTO/H2 Stream Management: Specification

- Client -> Server [Create transport queue]
  - Each request to create a transport queue (POST) MUST choose a new client selected stream ID (SID_tq)
  - Stream Identifier of the frame is a new client-selected stream ID; Stream Dependency in HEADERS is 0 (connection) for an independent resource, the other transport queue if the dependency is known
  - Invariant: Stream keeps open until close or error

```
+---------------------------+  +---------------------------+  
|                           |  |                           |
| Length (24)               |  | Pad Length? (8)           |
|                           |  | E| Stream Dependency? (31)  |
| Type (8)                  |  | Weight? (8)               |
|| Flags (8)                |  | Header Block Fragment (*) |
| R| Stream Identifier (31)   |  | Padding (*)               |
|                           |  | ...                        |
| Frame Payload (0...)      |  | ...                        |
```

Figure 1: Frame Layout

Figure 7: HEADERS Frame Payload
ALTO/H2 Stream Management: Specification

- Client -> Server [Close transport queue]
  - DELETE to close a transport queue (SID_tq) MUST be sent in SID_tq
  - Stream Identifier of the frame is SID_tq; Stream Dependency in HEADER is 0 (connection)
    - So that a client cannot close a different stream
    - Indicates END_STREAM; server response also close stream

![Frame Layout](image1.png)
![HEADERS Frame Payload](image7.png)

Figure 1: Frame Layout
Figure 7: HEADERS Frame Payload
ALTO/H2 Stream Management: Specification

- Client -> Server [Request on data of a transport queue SID_tq, e.g., read message]
  - Stream Identifier of the frame is a new client-selected stream ID, Stream Dependency in HEADERS MUST be SID_tq
  - So that a client cannot issue request on a closed transport queue
  - Request indicates END_STREAM; response also indicates end of stream

Figure 1: Frame Layout

Figure 7: HEADERS Frame Payload
ALTO/H2 Stream Management: Specification

- Server -> Client PUSH_PROMISE for transport queue SID_tq
  - PUSH_PROMISE sent in stream SID_tq to serialize to allow the client to know the push order
  - Each PUSH_PROMISE chooses a new server-selected stream ID
- Stream is closed after push

---

Figure 1: Frame Layout

---

Figure 11: PUSH_PROMISE Payload Format
Concurrent Streams Management

• Controlled by SETTINGS_MAX_CONCURRENT_STREAMS

• Client -> Server
  – There is one stream for each open transport queue
    • A client can always close a transport queue (it uses the open stream) and hence can open -> can close, without issue of deadlock

• Server -> Client push
  – Each push needs to open a new stream
Outline

• Motivation and requirements
• ALTO/H2 design
  – Overview
  – Transport queue
  – Incremental updates queue
  – Individual updates
  – Receiver set
  – Stream management
• Discussions and open issues
Transport and Pub/sub

• What is missing
  – The design does not allow creation of generic message queues
  – Only the server can be the publisher
    • Clients cannot publish info to be shared with other clients
  – The design does not have the capability of Exchange (message router)

• Way forward: Keep simple
  – Broker for further discussion

Capability Negotiation

- Capability Negotiation is not fully specified
  - Instead of fix stream management, client server can negotiate
Additional Information about Transport Queue

• Calendar semantics
  – Tell the client ALTO information (e.g., cost) for a future time point
  – Tell the client when the next information will be released, it is the time that the info is released is distributed, not the value [support]
Thank you!

Questions?