ALTO WG K. Gao
Internet-Draft Tsinghua University
Intended status: Standards Track X. Wang
Expires: September 3, 2018 Tongji University
Q. Xiang
Tongji/Yale University
C. Gu
Tongji University
Y. Yang
Yale University
G. Chen
Huawei
March 2, 2018

Compressing ALTO Path Vectors
draft-gao-alto-routing-state-abstraction-08.txt

Abstract

The path vector extension [I-D.ietf-alto-path-vector] has extended the base ALTO protocol [RFC7285] with the ability to represent a more detailed view of the network which contains not only end-to-end costs but also information about shared bottlenecks.

However, the view computed by straw man algorithms can contain redundant information and result in unnecessary communication overhead. The situation gets even worse when certain ALTO extensions are enabled, for example, the incremental update extension [I-D.ietf-alto-incr-update-sse] which continuously pushes data changes to ALTO clients. Redundant information can trigger unnecessary updates.

In this document, several algorithms are described which can effectively reduce the redundancy in the network view while still providing the same information as in the original path vectors.

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].

Copyright Notice

Table of Contents

1. Introduction

The path vector extension [I-D.ietf-alto-path-vector] has extended the base ALTO protocol [RFC7285] with the ability to present more complex network views than the simple abstraction used by Cost Map or Endpoint Cost Service. ALTO clients can query more sophisticated information such as shared bottlenecks, and schedule their flows properly to avoid congestion and to better utilize network resources.

Meanwhile, the extension itself does not specify how an ALTO server should respond to a path-vector query. A straw man approach, as in the context of Software Defined Networking (SDN) where network providers have a global view, can compute the path vectors by retrieving the paths for all requested flows and returning the links on those paths as abstract network elements. However, this approach has several drawbacks:

- The resultant network view may lead to privacy leaks. Since the paths constitute a sub-graph of the global network topology, they may contain sensitive information without further processing.

- The resultant network view may contain redundant information. The path vector information is primarily used to avoid network bottlenecks. Thus, if a link cannot become the bottleneck, as demonstrated in Section 4, it is considered as redundant. Redundant links not only increase the communication overhead of the path vector extension, but also trigger false-positive data change events when the incremental update extension [I-D.ietf-alto-incr-update-sse] is activated.

To overcome these limitations, this document describes equivalent transformation algorithms that identify redundant abstract network elements and reduce them as much as possible. The algorithm can be integrated with any implementation of the path vector extension as a post-processing step. As the name suggests, this algorithm conducts equivalent transformations on the original path vectors, removes redundant information and obtains a more compact view.

This document is a supplement to the path vector extension and can be optionally turned on and off without affecting the correctness of responses. A crucial part of the equivalent transformation algorithm is how to find redundant abstract network elements. By tuning the redundancy check algorithm, one can make different trade-off

decisions between efficiency and privacy. A reference implementation of redundancy check algorithm is also described in this document.

This document is organized as follows. Section 4 gives a concrete example to demonstrate the importance of compressing path vectors. The compression algorithms are specified in Section 5 and Section 6 discusses how one can use these algorithms on existing path vector responses. Finally, Section 7 and Section 8 discuss security and IANA considerations.

2. Changes Since Version -03, -04, -05, -06 and -07

In early versions of this draft, a lot of contents are shared with the path vector draft. From version -04, the authors have adjusted the structure and target this document as a supplement of the path vector extension with

o practical compression algorithms which can effectively reduce the leaked information and the communication overhead; and

o detailed instructions on how an original path vector response can be processed by these algorithms.

The -06 version fixed some minor issues in -04 and -05. The -07 version has focused on improving the clarity of the algorithms with more examples. The -08 version has improved the overall quality of the draft, especially the clarity of the algorithms using simpler symbols.

3. Terminology

This document uses the same terms as in [I-D.ietf-alto-path-vector].

4. Compressing Path Vectors

We use the example shown in Figure 1 to demonstrate the importance of compressing path vectors. The network has 6 switches (sw1 to sw6) forming a dumbbell topology where switches sw1/sw3 provide access on the left hand side, s2/s4 provide access on the right hand side, and sw5/sw6 form the backbone. End hosts eh1 to eh4 are connected to access switches sw1 to sw4 respectively. Assume that the bandwidth of each link is 100 Mbps, and that the network is abstracted with 4 PIDs each representing a host at one access switch.

```
PID1 +-----+                                        +-----+  PID2
eh1__|     |_                                   ____|     |__eh2
     | sw1 | \   +------+         +------+ /    | sw2 |
     +-----+  \  |      |         |      |/     +-----+
               \_| sw5  +---------+ sw6  |
PID3 +-----+   / |      |         |      |\     +-----+  PID4
eh3__|     |__/  +------+         +------+ \____|     |__eh4
     | sw3 |                                    | sw4 |
     +-----+                                    +-----+
```

Figure 1: Raw Network Topology

Link	Description
link1	sw1 <==> sw5
link2	sw2 <==> sw6
link3	sw3 <==> sw5
link4	sw4 <==> sw6
link5	sw5 <==> sw6

Table 1: Description of the Links

Three cases are identified when path vectors can be further compressed and an example is provided for each case.

4.1. Equivalent Aggregation

Consider an application which schedules the traffic consisting of two flows, eh1 -> eh2 and eh3 -> eh4. The application can query the path vectors and a straw man implementation will return all 5 links (abstract network elements) as shown in Figure 2.

```
path vectors:
    eh1: [ eh2: [ane:l1, ane:l5, ane:l2]]
    eh3: [ eh4: [ane:l3, ane:l5, ane:l4]]

abstract network element property map:
    ane:l1 : 100 Mbps
    ane:l2 : 100 Mbps
    ane:l3 : 100 Mbps
    ane:l4 : 100 Mbps
    ane:l5 : 100 Mbps
```

Figure 2: Path Vectors Returned by a Straw Man Implementation

The resultant path vectors represent the following linear constraints on the available bandwidth for the two flows:

```
bw(eh1->eh2)                <= 100 Mbps (ane:l1)
bw(eh1->eh2)                <= 100 Mbps (ane:l2)
               bw(eh3->eh4) <= 100 Mbps (ane:l3)
               bw(eh3->eh4) <= 100 Mbps (ane:l4)
bw(eh1->eh2) + bw(eh3->eh4) <= 100 Mbps (ane:l5)
```

Figure 3: Linear Constraints Represented by the Path Vectors

It can be seen that the constraints of ane:l1 and ane:l2 are exactly the same, and so are those of ane:l3 and ane:l4. Intuitively, we can replace ane:l1 and ane:l2 with a new abstract network element ane:1, and similarly replace ane:l3 and ane:l4 with ane:2. The new path vectors are shown in Figure 4.

```
path vectors:
    eh1: [ eh2: [ane:1, ane:l5]]
    eh3: [ eh4: [ane:2, ane:l5]]

abstract network element property map:
    ane:1  : 100 Mbps
    ane:2  : 100 Mbps
    ane:l5 : 100 Mbps
```

Figure 4: Path Vectors after Merging ane:l1/ane:l2 and ane:l3/ane:l4

4.2. Redundant Network Elements

Consider the same case as in Section 4.1. Taking a deeper look at Figure 3, one can conclude that constraints of ane:1 (ane:l1/ane:l2) and ane:2 (ane:l3/ane:l4) can be implicitly derived from that of ane:l5. Thus, these constraints are considered _redundant_ and the path vectors in Figure 4 can be further reduced. We replace ane:l5 with a new ane:3 and the new path vectors are shown in Figure 5.

```
path vectors:
    eh1: [ eh2: [ane:3]]
    eh3: [ eh4: [ane:3]]

abstract network element property map:
    ane:3 : 100 Mbps
```

Figure 5: Path Vectors after Removing Redundant Elements

It is clear that the new path vectors (Figure 5) are much more compact than the original path vectors (Figure 2) but they contain

just as much information. Meanwhile, the application can hardly infer anything about the original topology with the compact path vectors.

4.3. Equivalent Decomposition

However, it is not always possible to directly remove all redundant network elements. For example, consider the case when both bandwidth and routingcost are requested, and the values are as shown in Figure 6. Note that we have changed the bandwidth for ane:l5 for demonstration purpose.

```
              path vectors:
                  eh1: [ eh2: [ane:l1, ane:l5, ane:l2]]
                  eh3: [ eh4: [ane:l3, ane:l5, ane:l4]]

              abstract network element property map:
                  ane:l1 : 100 Mbps, 1
                  ane:l2 : 100 Mbps, 2
                  ane:l3 : 100 Mbps, 1
                  ane:l4 : 100 Mbps, 1
                  ane:l5 : 200 Mbps, 1
```

Figure 6: Path Vectors Returned by a Straw Man Implementation

```
            bw(eh1->eh2)                <= 100 Mbps (ane:l1)
            bw(eh1->eh2)                <= 100 Mbps (ane:l2)
                           bw(eh3->eh4) <= 100 Mbps (ane:l3)
                           bw(eh3->eh4) <= 100 Mbps (ane:l4)
            bw(eh1->eh2) + bw(eh3->eh4) <= 200 Mbps (ane:l5)
```

Figure 7: Bandwidth Constraints in the Original Path Vectors

```
       rc(eh1->eh2) = rc(ane:l1) + rc(ane:l2) + rc(ane:l5) = 4
       rc(eh3->eh4) = rc(ane:l3) + rc(ane:l4) + rc(ane:l5) = 3
```

Figure 8: Routingcost Information in the Original Path Vectors

Figure 7 and Figure 8 demonstrate the bandwidth and routingcost information one can obtain from the original path vector. Again, ane:l1/ane:l2 and ane:l3/ane:l4 can still be aggregated in a similar way as in Figure 4 by setting the routingcost of ane:1 and ane:2 to 3 and 2 respectively. However, we cannot remove the redundant network element (ane:l5 in this case) directly because the resultant path vectors (Figure 9) would not provide the same routingcost information as in the original path vector.

```
                  path vectors:
                      eh1: [ eh2: [ane:1]]
                      eh3: [ eh4: [ane:2]]

                  abstract network element property map:
                      ane:1  : 100 Mbps, 3
                      ane:2  : 100 Mbps, 2
```

Figure 9: Path Vectors after Removing Redundant Network Element

A further observation is that since the bandwidth constraint of ane:l5 is redundant, it can be equally represented as two abstract network elements ane:a5 and ane:b5, as shown in Figure 10.

```
                  path vectors:
                      eh1: [ eh2: [ane:1, ane:a5]]
                      eh3: [ eh4: [ane:2, ane:b5]]

                  abstract network element property map:
                      ane:1  : 100 Mbps, 3
                      ane:2  : 100 Mbps, 2
                      ane:a5 : 200 Mbps, 1
                      ane:b5 : 200 Mbps, 1
```

Figure 10: Path Vectors after Decomposing ane:l5

Since ane:1/ane:a5 and ane:2/ane:b5 can be aggregated as ane:3 and ane:4 respectively, the final path vectors only contain two network elements, as shown in Figure 11.

```
                  path vectors:
                      eh1: [ eh2: [ane:1]]
                      eh3: [ eh4: [ane:2]]

                  abstract network element property map:
                      ane:1  : 100 Mbps, 4
                      ane:2  : 100 Mbps, 3
```

Figure 11: Path Vectors after Merging ane:1/ane:a5 and ane:2/ane:b5

One can verify that this path vector response has just the same information as in Figure 6 but contains much less contents.

5. Compression Algorithms

To provide a guideline on how path vectors MIGHT be compressed, this section describes the details of the algorithms for the three aforementioned cases:

1. Equivalent aggregation (EQUIV_AGGR), which compresses the original path vectors by aggregating the network elements with the same set of pairs as shown in Section 4.1;

2. Identification of redundant constraints (IS_REDUNDANT), which compresses the original path vectors by removing the network elements that provide only redundant information as shown in Section 4.2;

3. Equivalent decomposition (EQUIV_DECOMP), which compresses the original path vectors by decomposing redundant network elements to obtain the same end-to-end routing metrics as shown in Section 4.3.

5.1. Equivalent Aggregation

5.1.1. Parameters and Variables

The equivalent aggregation algorithm takes 3 parameters: the set of network elements "V", the set of relevant host pairs "P" and the set of metrics "M".

Set of network elements V: The set of network elements consists of all the network elements that exists in the original path vectors. The "i"-th network element in "V" is denoted as "v_i".

Set of relevant host pairs P: The "i"-th element in "P" is denoted as "p_i". It represents the set of (src, dst) pairs whose paths traverse "v_i" in the original path vectors.

Set of metrics M: The "i-th" element in "M" is denoted as "m_i". It represents the set of metrics associated with network element "v_i".

The output of the equivalent aggregation algorithm is a new set of network elements "V'", a new set of relevant host pairs "P'" and a new set of metrics "M'", i.e., "V', P', M' = EQUIV_AGGR(V, P, M)".

5.1.2. Algorithm Description

1. Set "V'", "P'", "M'" to empty sets. Set "k" to 0. Go to step 2.

2. If "V" is empty, go to step 6. Otherwise, go to step 3.

3. Select an arbitrary element "v_i" from "V", remove "v_i" from "V" and go to step 4.

4. For any element "v_j" in "V", if "p_i = p_j", remove "v_j" from "V" and update "m_i" with "m_j", i.e., "m_i = UPDATE(m_i, m_j)" (which will be explained later). Go to step 5.

5. Increment "k" by 1, let "v'_k = v_i", "p'_k = p_i" and "m'_k = m_i". Go to step 2.

6. Return "V'", "P'", and "M'"

The process of update "m_i" with "m_j" depends on the metric types. For example, for routingcost and hopcount, the update is numerical addition, while for bandwidth, the update is calculating the minimum. The UPDATE function for some common metrics are listed in Table 2.

metric	UPDATE(x, y)	default
hopcount	x + y	0
routingcost	x + y	0
bandwidth	min(x, y)	+infinity
loss rate	1 - (1 - x) * (1 - y)	0

Table 2: UPDATE Function of Different Metrics

5.1.3. Example

Consider the path vectors in Figure 2 which can be represented as:

```
V     = { ane:l1, ane:l2, ane:l3, ane:l4, ane:l5 }

p_1   = { eh1->eh2 }
p_2   = { eh1->eh2 }
p_3   = { eh3->eh4 }
p_4   = { eh3->eh4 }
p_5   = { eh1->eh2, eh3->eh4 }

m_1   = 100 Mbps
m_2   = 100 Mbps
m_3   = 100 Mbps
m_4   = 100 Mbps
m_5   = 100 Mbps
```

As "p_1 = p_2" and "p_3 = p_4", the resultant attributes after the aggregation become:

```
V'    = { ane:1, ane:2, ane:l5 }

p'_1  = { eh1->eh2 } = p_1 = p_2
p'_2  = { eh3->eh4 } = p_3 = p_4
p'_3  = { eh1->eh2, eh3->eh4 } = p_5

m'_1  = 100 Mbps = UPDATE(m_1, m_2)
m'_2  = 100 Mbps = UPDATE(m_3, m_4)
m'_3  = 100 Mbps = m_5
```

5.2. Redundant Network Element Identification

5.2.1. Parameters and Variables

The redundant network element identification algorithm is based on the algorithm introduced by Telgen [TELGEN83]. It takes 3 parameters: the set of network elements "V", the set of relevant host pairs "P" and the set of available bandwidth values "B".

"V", "v_i", "P" and "p_i" are defined the same way as in Section 5.1.1.

Set of available bandwidth values B: The "i"-th element in "B" is denoted as "b_i". It represents the available bandwidth for network element "v_i".

The output of the IS_REDUNDANT function is a set of indices "R", which represents the indices of network elements whose bandwidth constraints are redundant, i.e., "R = IS_REDUNDANT(V, P, B)".

In addition to the parameters and output values, the algorithm also maintains the following variables:

Set of host pairs H: The "i"-th element of "H" is denoted as "h_i". It represents a (src, dst) pair ever appeared in the path vector query. "H" is the union of all "p_i" in "P".

Set of bandwidth constraints C: The "i"-th element of "C" is denoted as "c_i". It represents a linear bandwidth constraint on the flows between the end host pairs. The constraint "c_i" has the form of "a_i x <= b_i" where "a_i" is a row vector of 0-1 coefficients derived from "p_i", "x" is a column vector representing the bandwidth of all the host pairs, and "b_i" is the available bandwidth of "v_i".

5.2.2. Algorithm Description

1. The first step is to convert a network element to its bandwidth constraint "c_i". The bound "b_i" is directly obtained as the available bandwidth and the coefficients "a_i" are computed as:

```
        1  if h_j in p_i
a_ij =
        0  otherwise.
```

   Set "R" to an empty set. Go to step 2.

2. For each "i", solve the following linear programming problem:

```
            y_i = max a_i x
subject to:
    a_j x <= b_j, j = 1..|V|, i <> j
```

   Go to step 3.

3. For each "i", if "y_i <= b_i", "c_i" is redundant and we say "v_i" is redundant, "R = UNION(R, {i})". Go to step 4.

4. Return "R".

5.2.3. Example

Consider the path vectors in Figure 4 such that the input to the IS_REDUNDANT algorithm is as follows.

```
V     = { ane:1, ane:2, ane:l5 }

p_1   = { eh1->eh2 }
p_2   = { eh3->eh4 }
p_3   = { eh1->eh2, eh3->eh4 }

b_1   = 100 Mbps
b_2   = 100 Mbps
b_3   = 100 Mbps
```

With that information, one can follow the algorithm and get:

```
c_1:   x1        <= 100
c_2:        x2   <= 100
c_3:   x1 + x2   <= 100

y_1  = 100 Mbps <= b_1
y_2  = 100 Mbps <= b_2
y_3  = 200 Mbps >  b_3

R    = IS_REDUNDANT(V, P, B) = { 1, 2 }
```

5.3. Equivalent Decomposition

5.3.1. Parameters and Variables

The equivalent decomposition algorithm takes 4 parameters: the set of network elements "V", the set of relevant host pairs "P", the set of metrics "M" and the set of redundant network elements "R".

"V", "P" and "M" are as defined as in Section 5.1.1. If the "j"-th metric is bandwidth, we can construct the set of available bandwidth values "B" as "b_i = m_ij" and "R" is the output of the redundant network element identification procedure, i.e. "R = IS_REDUNDANT(V, P, B)". Otherwise, if bandwidth is not included in the metrics, "R" is {1, ..., |V|}.

The output of the function EQUIV_DECOMP is a new set of network elements "V'", a new set of relevant host pairs "P'", and a new set of metrics "M'", i.e., "V', P', M' = EQUIV_DECOMP(V, P, M, R)".

5.3.2. Algorithm Description

1. Set "V'", "P'", "M'" to empty sets. Set "k" to 0. Go to step 2.

2. For each "i" such that "i" in "R", go to step 3. After processing each "i", go to step 7.

3. For each "j" such that "j <> i", go to step 4. After processing each "j", go to step 6.

4. If "p_j" is a subset of "p_i", go to step 5. Otherwise go to step 3.

5. Let "p_i = p_i \ p_j" and "m_j = UPDATE(m_j, m_i)". Go to step 3.

6. If "p_i" is not empty, increment "k" by 1 and let "v'_k = v_i", "p'_k = p_i" and "m'_k = m_i". Go to step 2.

7. For each "i" such that "i" is not in "R", go to step 8. After processing each "i", go to step 9.

8. Increment "k" by 1 and let "v'_k = v_i", "p'_k = p_i", "m'_k = m_i". Go to step 7.

9. Return "V'", "P'" and "M'".

5.3.3. Example

Consider the case in Section 4.3. Before the decomposition, the input to the algorithm is as follows:

```
V     = { ane:1, ane:2, ane:l5 }

p_1   = { eh1->eh2 }
p_2   = { eh3->eh4 }
p_3   = { eh1->eh2, eh3->eh4 }

m_1   = { bw: 100 Mbps, rc: 3 }
m_2   = { bw: 100 Mbps, rc: 2 }
m_3   = { bw: 200 Mbps, rc: 1 }

R     = { 3 }
```

Since there is only one element in "R", "v_i = ane:l5".

After the first iteration of steps 3-5 with "v_j = ane:1":

```
V     = { ane:1, ane:2, ane:l5 }

p_1   = { eh1->eh2 }
p_2   = { eh3->eh4 }
p_3   = { eh3->eh4 }

m_1   = { bw: 100 Mbps, rc: 4 }
m_2   = { bw: 100 Mbps, rc: 2 }
m_3   = { bw: 200 Mbps, rc: 1 }

V'    = { }
k     = 0
```

After the second iteration of steps 3-5 with "v_j = ane:2":

```
V     = { ane:1, ane:2, ane:l5 }

p_1   = { eh1->eh2 }
p_2   = { eh3->eh4 }
p_3   = { }

m_1   = { bw: 100 Mbps, rc: 4 }
m_2   = { bw: 100 Mbps, rc: 3 }
m_3   = { bw: 200 Mbps, rc: 1 }

V'    = { }
k     = 0
```

After step 6, since "p_3" is now empty, it just goes back to step 2. At step 2, since all indices in "R" has been processed, it goes to step 7.

After the first iteration of steps 7-8 with "i = 1":

```
V     = { ane:1, ane:2, ane:l5 }

p_1   = { eh1->eh2 }
p_2   = { eh3->eh4 }
p_3   = { }

m_1   = { bw: 100 Mbps, rc: 4 }
m_2   = { bw: 100 Mbps, rc: 3 }
m_3   = { bw: 200 Mbps, rc: 1 }

V'    = { ane:1 }
k     = 1

p'_1  = { eh1->eh2 } = p_1

m'_1  = { bw: 100 Mbps, rc: 4 } = m_1
```

After the second iteration of steps 7-8 with "i = 2":

```
V      = { ane:1, ane:2, ane:l5 }

p_1    = { eh1->eh2 }
p_2    = { eh3->eh4 }
p_3    = { }

m_1    = { bw: 100 Mbps, rc: 4 }
m_2    = { bw: 100 Mbps, rc: 3 }
m_3    = { bw: 200 Mbps, rc: 1 }

V'     = { ane:1, ane:2 }
k      = 2

p'_1   = { eh1->eh2 }
p'_2   = { eh3->eh4 } = p_2

m'_1   = { bw: 100 Mbps, rc: 4 }
m'_2   = { bw: 100 Mbps, rc: 3 } = m_2
```

So the final output of EQUIV_DECOMP is:

```
V'     = { ane:1, ane:2 }

p'_1   = { eh1->eh2 }
p'_2   = { eh3->eh4 }

m'_1   = { bw: 100 Mbps, rc: 4 }
m'_2   = { bw: 100 Mbps, rc: 3 }
```

5.4. Execution Order

As the examples demonstrate, the three algorithms MUST be executed in the same order as they are introduced, i.e., one MUST conduct "EQUIV_AGGR" before "IS_REDUNDANT" or "EQUIV_DECOMP", and conduct "IS_REDUNDANT" before "EQUIV_DECOMP". Otherwise, the results of the compressed path vectors MAY NOT be correct.

6. Encoding/Decoding Path Vectors

The three algorithms work mostly with network elements. Existing path vectors must be decoded before they can be passed on to the algorithms and the compressed results must be encoded as path vectors before they are sent to the clients. The decoding and encoding processes are specified as below.

6.1. Decoding Path Vectors

6.1.1. Parameters and Variables

The decoding algorithm DECODE takes a path vector response, which consists of the path vector part "PV" and the element property part "E".

Path vectors PV: The path vector part has a format of a CostMap (EndpointCostMap) where the cost value is a list of abstract network element names. We say a PID (endpoint address) "i" is IN "PV" if and only if there is an entry "i" in the cost-map (endpoint-cost-map), and denote the entry value as "PV[i]". Similarly, we say a PID (endpoint address) "j" is IN "PV[i]" if and only if there is an entry "j" in the DstCosts of "i", whose value is denoted as "PV[i][j]".

Element property map E: The element property map "E" maps an abstract network element name to its properties. We denote "E[n]" as the properties of element with name "n" and "E[n][pn]" as the value of property "pn".

The algorithm returns the set of elements "V", the set of relevant host pairs "P", the set of metrics "M" and the available bandwidth "B", as defined in Section 5.1.1 and Section 5.2.1. The algorithm uses a "SET" function which transforms a list into a set, and uses a "NAME" function which maps an integer in [1, K] to a unique property name where there are K properties in "E".

6.1.2. Algorithm Description

1. Set "V", "P", "M" and "B" to empty sets. Set "k" to 0. Go to step 2.

2. For each "i IN PV", go to step 3. After processing each "i", go to step 8.

3. For each "j IN PV[i]", go to step 4. After processing each "j", go to step 2.

4. For each "n" in "SET(PV[i][j])", go to step 5. After processing each "n", go to step 3.

5. If "n" is not in "V", go to step 6. Otherwise, go to step 7.

6. Increment "k" by 1 and let "v_k = n", "p_k = { i->j }". Go to step 4.

7. Find the index of "n" in "V" denoted as "a", let "p_a = UNION(p_a, {i->j})". Go to step 4.

8. For each "i" from 1 to |V|, go to step 9. After processing all "i", go to step 11.

9. For each "j" from 1 to K, go to step 10. After processing all "j", go back to step 8.

10. If "NAME(j) = 'availbw'", let "b_i = E[v_i][NAME(j)]". Let "m_ij = E[v_i][NAME(j)]".

11. Return "V", "P", "M" and "B".

6.1.3. Example

Consider the following example:

```
HTTP/1.1 200 OK
Content-Length: [TBD]
Content-Type: multipart/related; boundary=example-2

--example-2
Content-Type: application/alto-endpointcost+json

{
  "meta": {
    "cost-types": [
      {"cost-mode": "array", "cost-metric": "ane-path"}
    ]
  }
  "endpoint-cost-map": {
    "ipv4:192.0.2.2": {
      "ipv4:192.0.2.89": [ "ane:L1", "ane:L3", "ane:L4" ],
      "ipv4:203.0.113.45": [ "ane:L1", "ane:L4", "ane:L5" ]
    }
  }
}
```

```
--example-2
Content-Type: application/alto-propmap+json

{
  "property-map": {
    "ane:L1": { "availbw": 50 },
    "ane:L3": { "availbw": 48 },
    "ane:L4": { "availbw": 55 },
    "ane:L5": { "availbw": 60 },
    "ane:L7": { "availbw": 35 }
  }
}

--example-2--
```

After the first iteration of Lines 2-5:

```
V      = { ane:L1, ane:L3, ane:L4 }

p_1    = { ipv4:192.0.2.2->ipv4:192.0.2.89 }
p_2    = { ipv4:192.0.2.2->ipv4:192.0.2.89 }
p_3    = { ipv4:192.0.2.2->ipv4:192.0.2.89 }.
```

After the second iteration of Lines 2-5:

```
V      = { ane:L1, ane:L3, ane:L4, ane:L5 }

p_1    = { ipv4:192.0.2.2->ipv4:192.0.2.89,
           ipv4:192.0.2.2->ipv4:203.0.113.45 }
p_2    = { ipv4:192.0.2.2->ipv4:192.0.2.89,
           ipv4:192.0.2.2->ipv4:203.0.113.45 }
p_3    = { ipv4:192.0.2.2->ipv4:192.0.2.89 }
p_4    = { ipv4:192.0.2.2->ipv4:203.0.113.45 }.
```

After the first iteration of Lines 6-9 with "i = 1":

```
m_1    = [50]

b_1    = 50
```

After all four iterations of Lines 6-9:

```
m_1    = [50]
m_2    = [48]
m_3    = [55]
m_4    = [60]

b_1    = 50
b_2    = 48
b_3    = 55
b_4    = 60
```

The decoded information can be passed on to "EQUIV_AGGR", "IS_REDUNDANT" and "EQUIV_DECOMP" for compression.

6.2. Encoding Path Vectors

6.2.1. Parameters and Variables

The algorithm ENCODE is the reverse process of DECODE. It takes the parameters "V", "P", "M" and constructs the path vector results.

The parameters are defined as in Section 5.1.1 and Section 5.2.1.

The algorithm also uses the NAME function in Section 6.1.1 which MUST return the same results in a paired ENCODE/DECODE process, and the "APPEND(L, e)" function which adds element "e" to list "L".

6.2.2. Algorithm Description

1. Set "PV={}", "E = {}". Go to step 2.

2. For each "v_i" in "V", go to step 3. If all "v_i" is processed, go to step XX.

3. For each "a->b" in "p_i", go to step 4. If all such "a->b" is processed, go to step 6.

4. If "a" is not in "PV", let "PV[a] = {}". Go to step 5.

5. If "b" is not in "PV[a]", let "PV[a][b] = [v_i]". Otherwise, let "PV[a][b] = APPEND(PV[a][b], v_i)". Go to step 2.

6. For each index "k" in [1, K], go to step 7. If all "k" is processed, go to step 1.

7. Set "E[v_i][NAME(k)] = m_ik". Go to step 6.

8. Return "PV" and "E".

6.2.3. Example

We consider the encoding of the decoded example in Section 6.1.3.

```
V      = { ane:L1, ane:L3, ane:L4, ane:L5 }

p_1    = { ipv4:192.0.2.2->ipv4:192.0.2.89,
           ipv4:192.0.2.2->ipv4:203.0.113.45 }
p_2    = { ipv4:192.0.2.2->ipv4:192.0.2.89,
           ipv4:192.0.2.2->ipv4:203.0.113.45 }
p_3    = { ipv4:192.0.2.2->ipv4:192.0.2.89 }
p_4    = { ipv4:192.0.2.2->ipv4:203.0.113.45 }

m_1    = [50]
m_2    = [48]
m_3    = [55]
m_4    = [60]
```

After the first iteration of steps 2-7:

```
PV[ipv4:192.0.2.2][ipv4:192.0.2.89  ] = [ane:L1]
PV[ipv4:192.0.2.2][ipv4:203.0.113.45] = [ane:L1]

E[ane:L1]["availbw"]                = 50
```

After the second iteration:

```
PV[ipv4:192.0.2.2][ipv4:192.0.2.89  ] = [ane:L1, ane:L3]
PV[ipv4:192.0.2.2][ipv4:203.0.113.45] = [ane:L1, ane:L3]

E[ane:L1]["availbw"]                = 50
E[ane:L3]["availbw"]                = 48
```

After the third iteration:

```
PV[ipv4:192.0.2.2][ipv4:192.0.2.89  ] = [ane:L1, ane:L3, ane:L4]
PV[ipv4:192.0.2.2][ipv4:203.0.113.45] = [ane:L1, ane:L3]

E[ane:L1]["availbw"]                = 50
E[ane:L3]["availbw"]                = 48
E[ane:L4]["availbw"]                = 55
```

After the fourth iteration:

```
PV[ipv4:192.0.2.2][ipv4:192.0.2.89  ] = [ane:L1, ane:L3, ane:L4]
PV[ipv4:192.0.2.2][ipv4:203.0.113.45] = [ane:L1, ane:L3, ane:L5]

E[ane:L1]["availbw"]                = 50
E[ane:L3]["availbw"]                = 48
E[ane:L4]["availbw"]                = 55
E[ane:L5]["availbw"]                = 60
```

Eventually, one can use the previous information to construct the endpoint cost service response.

```
HTTP/1.1 200 OK
Content-Length: [TBD]
Content-Type: multipart/related; boundary=example-2

--example-2
Content-Type: application/alto-endpointcost+json
```


```
{
  "meta": {
    "cost-types": [
      {"cost-mode": "array", "cost-metric": "ane-path"}
    ]
  }
  "endpoint-cost-map": {
    "ipv4:192.0.2.2": {
      "ipv4:192.0.2.89": [ "ane:L1", "ane:L3", "ane:L4" ],
      "ipv4:203.0.113.45": [ "ane:L1", "ane:L4", "ane:L5" ]
    }
  }
}
```


```
--example-2
Content-Type: application/alto-propmap+json
```


```
{
  "property-map": {
    "ane:L1": { "availbw": 50 },
    "ane:L3": { "availbw": 48 },
    "ane:L4": { "availbw": 55 },
    "ane:L5": { "availbw": 60 },
  }
}
```


```
--example-2--
```

6.3. Compatibility

When the path vector extension is used with other extensions, such as [I-D.ietf-alto-cost-calendar] and [I-D.ietf-alto-multi-cost], the decoding and the encoding MUST only apply on the path vector part and leave the other attributes as they are.

Hence, this extension does not change the compatibility between the original path vector extension and other extensions.

7. Security Considerations

This document does not introduce any privacy or security issue on ALTO servers not already present in the base ALTO protocol or in the path vector extension.

The algorithms specified in this document can even help protect the privacy of network providers by conducting irreversible transformations on the original path vector.

8. IANA Considerations

This document does not define any new media type or introduce any new IANA consideration.

9. Acknowledgments

The authors would like to thank Dr. Qiao Xiang, Mr. Jingxuan Zhang (Tongji University), Prof. Jun Bi (Tsinghua University) and Dr. Andreas Voellmy (Yale University) for their early engagement and discussions.

10. References

10.1. Normative References

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

10.2. Informative References

[I-D.ietf-alto-cost-calendar]
Randriamasy, S., Yang, Y., Wu, Q., Lingli, D., and N. Schwan, "ALTO Cost Calendar", draft-ietf-alto-cost-calendar-01 (work in progress), February 2017.

[I-D.ietf-alto-incr-update-sse]
Roome, W. and Y. Yang, "ALTO Incremental Updates Using Server-Sent Events (SSE)", draft-ietf-alto-incr-update-sse-02 (work in progress), April 2016.

[I-D.ietf-alto-multi-cost]
Randriamasy, S., Roome, W., and N. Schwan, "Multi-Cost ALTO", draft-ietf-alto-multi-cost-10 (work in progress), April 2017.

[I-D.ietf-alto-path-vector]
Bernstein, G., Chen, S., Gao, K., Lee, Y., Roome, W., Scharf, M., Yang, Y., and J. Zhang, "ALTO Extension: Path Vector Cost Mode", draft-ietf-alto-path-vector-00 (work in progress), May 2017.

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

[TELGEN83]
Telgen, J., "Identifying Redundant Constraints and Implicit Equalities in Systems of Linear Constraints", Management Science , Volume 29, Issue 10, DOI 10.1287/mnsc.29.10.1209, 1983, <http://pubsonline.informs.org/doi/abs/10.1287/mnsc.29.10.1209>.

Authors' Addresses

Kai Gao
Tsinghua University
30 Shuangqinglu Street
Beijing 100084
China

Email: gaok12@mails.tsinghua.edu.cn

Xin (Tony) Wang
Tongji University
4800 CaoAn Road
Shanghai 210000
China

Email: xinwang2014@hotmail.com

Qiao Xiang
Tongji/Yale University
51 Prospect Street
New Haven, CT
USA

Email: qiao.xiang@cs.yale.edu

Chen Gu
Tongji University
4800 CaoAn Road
Shanghai 210000
China

Email: gc19931011jy@gmail.com

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

Email: yry@cs.yale.edu

G. Robert Chen
Huawei
Nanjing
China

Email: chenguohai@huawei.com