Multi-Protocol Label Switching WG
Internet-Draft
Expiration Date: January 1999

Yoshihiro Ohba
Yasuhiro Katsube
Toshiba

Eric Rosen
Cisco Systems

Paul Doolan
Ennovate Networks

July 1998

MPLS Loop Prevention Mechanism

<draft-ohba-mpls-loop-prevention-01.txt>

Status of this Memo

Abstract

This paper presents a simple mechanism, based on 'threads', which can be used to prevent MPLS from setting up label switched path (LSPs) which have loops. The mechanism is compatible with, but does not require, VC merge. The mechanism can be used with either the ingress-initiated ordered control or the egress-initiated ordered control. The amount of information that must be passed in a protocol message is tightly bounded (i.e., no path-vector is used). When a node needs to change its next hop, a distributed procedure is executed, but only nodes which are downstream of the change are involved.

Table of contents

1. Introduction

This paper presents a simple mechanism, based on "threads", which can be used to prevent MPLS from setting up label switched paths (LSPs) which have loops. The thread mechanism is a generalization of [1].

When an LSR finds that it has a new next hop for a particular FEC, it creates a thread and extends it downstream. Each such thread is assigned a unique "color", such that no two threads in the network can have the same color.

Only a single thread for an LSP is ever extended to a particular next hop as long as the thread length does not change. The only state information that needs to be associated with a particular next hop for a particular LSP is the thread color and length.

The procedures for determining just how far downstream a thread must be extended are given in section 3.

If there is a loop, then some thread will arrive back at an LSR through which it has already passed. This is easily detected, since each thread has a unique color.

Section 3 provides procedures for determining that there is no loop. When this is determined, the threads are "rewound" back to the point of creation. As they are rewound, labels get assigned. Thus labels

are NOT assigned until loop freedom is guaranteed.

While a thread is extended, the LSRs through which it passes must remember its color and length, but when the thread has been rewound,

they need only remember its length.

The thread mechanism works if some, all, or none of the LSRs in the LSP support VC-merge. It can also be used with either the ingress-initiated ordered control or the egress-initiated ordered control [2,3].

The state information which must be carried in protocol messages, and which must be maintained internally in state tables, is of fixed size, independent of the length of the LSP. Thus the thread mechanism is more scalable than alternatives which require that path-vectors be carried.

To set up a new LSP after a routing change, the thread mechanism requires communication only between nodes which are downstream of the point of change. There is no need to communicate with nodes that are upstream of the point of change. Thus the thread mechanism is more robust than alternatives which require that a diffusion computation be performed.

The thread mechanism contains a number of significant improvements when compared to the mechanism described in the previous version of this internet draft. In particular:

- The thread mechanism allows a node whose next hop changes to continue using the old LSP while setting up the new LSP (or while waiting for the L3 routing to stabilize, so that a new loop-free LSP can be set up)

- When a loop is detected, path setup is delayed, but it is automatically resumed when the L3 routing stabilizes and the loop disappears. No retry timers are needed.

- "Color" only has to be remembered while a path is being set up. Once it is set up, the "color" (though not the length) can be forgotten.

In this paper, we assume unicast LSPs. The loop prevention for multicast LSPs is for further study.

2. Definitions

An LSP for a particular Forwarding Equivalence Class (FEC) [4] can be thought of as a tree whose root is the egress LSR for that FEC. With respect to a given node in the tree, we can speak of its "downstream link" as the link which is closest to the root; the node's other edges are "upstream links".

The term "link", as used here, refers to a particular relationship

on the tree, rather than to a particular physical relationship. In the remainder of this section, when we will speak of the upstream and downstream links of a node, we are speaking with reference to a single LSP tree for a single FEC.

In the case of non-VC-merging, multiple links of the same FEC between the neighboring nodes must be identified by identifiers that are locally assigned by the upstream node of the links.

A leaf node is a node which has no upstream links.

A "trigger node" is any node which (a) acquires a new next hop (i.e., either changes its next hop, or gets a next hop when it previously had none) for a particular FEC, and (b) needs to have an LSP for that FEC.

An LSP length at a node is represented by a hop count from the furthest leaf node to that node. The LSP length at a leaf node is zero.

In the remainder of the paper, we assume the "downstream-on-demand" is used as the label allocation method between neighboring nodes, although the thread mechanism is applicable to the upstream allocation method.

3. Thread mechanism

3.1. Thread

A thread is an object used for representing a loop-prevention process which extends downstream. The downstream end of a thread is referred to as the "thread head".

A thread has a color that is assigned at the node that creates the thread. The color is globally unique in the FEC.

A thread is always associated with a particular LSP (for a particular FEC). The "length" of a thread is the number of hops from the thread head to the node which is furthest upstream of it on the LSP. An "unknown" length which is greater than any other known length is used in a certain situation (see section 3.2).

A thread has a TTL which is decremented by one (except for a special "infinity" value, see section 4) as the thread is extended without changing its color.

For a given LSP, at a given LSR, there can be one "incoming thread" for each upstream neighbor, and one "outgoing thread" to the downstream neighbor. That is, one of the incoming threads is extended downstream. If a node is the creator of a thread, the thread becomes a "virtual incoming thread" whose upstream neighbor is the node itself. A non-virtual incoming thread is referred to as an "actual incoming thread".

When a thread is extended, it retains its color, but its length becomes the maximum incoming thread length plus 1.

If a thread head of a given color reaches a node which already has a thread of that color, then a loop has been detected.

When a node changes the color of its outgoing thread, it notifies its downstream neighbor by means of LDP messages. The downstream neighbor must process these messages in order.

3.2. Loop prevention algorithm

The ingress-initiated ordered control is assumed here, however, the algorithm can be adapted to egress-initiated ordered control (see section 3.4).

When a trigger node requests a label assignment to its downstream neighbor, it creates a thread and extends it downstream.

The thread is given an initial length corresponding to the number of hops between the trigger node and the furthest upstream leaf. It is given a color which consists of the trigger node's IP address, prepended to an event identifier which is assigned by the trigger node. The trigger node will never reuse an event identifier until sufficient time has passed so that we can be sure that no other node in the network has any memory of the corresponding color.

A colored thread is extended downstream until one of the following events occurs:

(i) the thread head reaches an egress node;

(ii) the thread head reaches a node where there is already an ESTABLISHED LSP for the thread, with a KNOWN length which is no less than the thread length;

(iii) the thread head reaches a node which already has an actual or a virtual incoming thread of that color;

(iv) the thread TTL becomes zero;

(v) the thread head reaches a node where the maximum incoming thread length is not updated and there is another actual incoming thread.

o In the case of (i) or (ii), the thread is assured to reach the egress node without forming a loop. Therefore the thread is "rewound". When a thread is rewound, each node takes the following actions. For each upstream link, it assigns a label to the LSP and distributes that label LSP upstream, if needed. It resets all incoming and outgoing thread colors to "transparent". It sets the longest length among actual incoming

threads to the LSP length. If the outgoing thread length is "unknown" and the obtained LSP length becomes known, it notifies downstream of the LSP length (by using a "transparent" thread).

When the thread is rewound back to the trigger node, the LSP setup completes.

- In the case of (iii) and (iv), the thread is neither extended nor rewound. It is blocked at the node. In the case of (iii), the following actions are taken. If the node is not the creator of the thread, it creates a new thread with "unknown" length and extends it downstream. Otherwise, if it is not a leaf node and there is no other actual incoming thread, it withdraws the outgoing thread (this will cause a thread reconstruction, see section 3.3).

- In the case of (v), the received thread is "merged" into the outgoing thread and no message is sent to the downstream neighbor.

When a trigger node is attempting to set up a new LSP, it also tells its old next hop that it is withdrawing the thread that goes through it to the old next hop. This will cause the old next hop to withdraw one of its incoming threads.

When an incoming thread is withdrawn, if there is no actual incoming thread, the outgoing thread is also withdrawn unless the node becomes a new leaf node. Otherwise, if it is the one currently being extended, a new thread is created and extended.

A transparent thread is extended when a node notifies the downstream neighbor on an established LSP of an LSP length update or a thread withdrawal without releasing the LSP. No rewinding is needed for transparent threads.

A virtual incoming thread is removed when the corresponding outgoing thread is replaced or withdrawn.

3.3. Why this works

The above procedures ensure that once a looping thread is detected, path setup along the LSRs in that thread is effectively stalled until the L3 routing changes so as to remove the loop.

How can we be sure that the any L3 loop will be detected by these procedures when a thread length is NOT "unknown"?

Consider a sequence of LSRs <R1, ..., Rn>, such that there is a loop traversing the LSRs in the sequence. (I.e., packets from R1 go to R2, then to R3, etc., then to Rn, and then from Rn to R1.)

Remember that after a routing change, a path cannot be set up (i.e., labels cannot be assigned) until the thread resulting from the

routing change is rewound, and the act of rewinding the thread causes the thread lengths to be set consistently along the path.

Suppose that the thread length of the link between R1 and R2 is k.

Then by the above procedures, the length of the link between Rn and R1 must be k+n-1. But the above procedures also ensure that if a node has an incoming thread of length j, its outgoing thread must be at least of length j+1. Hence, if we assume that the loop is not detected by the above procedure, the thread length of the link between R1 and R2 must be at least k+n. From this we may derive the absurd conclusion that n=0, and we may therefore conclude that there is no such sequence of LSRs.

When a thread of "unknown" length gets into an L3 loop, however, there is a situation in which the thread is merged into another thread of "unknown" length. In this case, the L3 loop would not be explicitly detected, but the thread is effectively stalled in the loop until the L3 routing changes so as to remove the loop.

Inversely, how can we be sure that no loop detection occurs when there is no loop?

Since every new path setup or release attempt that changes an LSP length causes the use of a new color, condition (iii) cannot obtain unless there actually is an L3 routing loop.

Next, why thread reconstructions are needed?

When a thread loop is detected, imagine a thread tree whose root is the thread head. If there is a leaf which is not an LSP leaf in that tree, then the thread will not disappear even after all LSP leaf node withdraw their threads. The thread reconstruction is performed to change the location of the thread head to the proper node where any leaf of the thread tree is an LSP leaf node.

In the above procedure, multiple thread updates may happen if several leaf nodes start extending threads at the same time. How can we prevent multiple threads from looping unlimitedly?

In the procedure, when a node detects a thread loop by condition (iii), it creates a new thread of "unknown" length. All the looping threads which later arrive at the node would be merged into this thread. Such a thread of "unknown" length behaves like a thread absorber. Furthermore, the thread TTL mechanism can eliminate any kind of thread looping.

3.4. Using old path while looping on new path

When a route changes, one might want to continue to use the old path if the new route is looping. This is archived simply by holding the label assigned to the downstream link on the old path until the thread head on the new route returns. This is an implementation choice.

3.5. How to deal with egress-initiated ordered control

The thread mechanism can be also adapted to the egress-initiated ordered control by originating a thread when a node newly receives an advertisement message [5] from the downstream node.

Note that a node which has no upstream link for the LSP yet behaves as a leaf. In the case where the tree is being initially built up (e.g., the egress node has just come up), each node in turn will behave as a leaf for a short period of time.

3.6. How to realize load splitting from the ingress node

The load splitting from the ingress node can be easily realized by assigning a different colored thread to each downstream link.

4. Modification to LDP specification

A new advertisement class message, Update is added to the current specification [5]. In addition, two new objects, Thread object and Request ID object are defined.

When a thread of a particular LSP is extended on a downstream link, if a label is not still allocated for that link, a Request message is used for carrying the thread as well as for requesting a label, otherwise, an Update message is used for extending the thread on the downstream link where a label already exists.

When a node wants to withdraw an outgoing thread as well as the downstream link, a Release message is used.

When a Mapping message is returned to an upstream node in response to a Request message, it is treated as an indication of thread rewinding (i.e., acknowledgments for loop-prevention check).

For an Update message, an ACK message is returned and treated as an indication of thread rewinding, except for an Update containing a special "transparent" thread. (See section 4.1.2.)

4.1. LDP objects

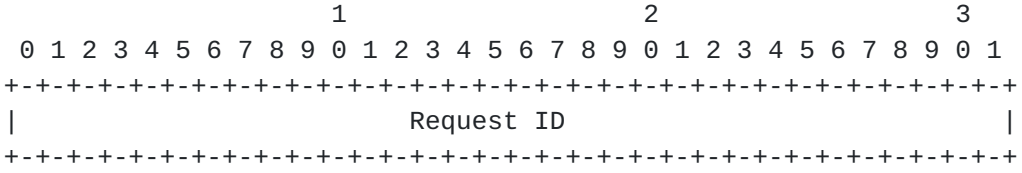
4.1.1. Request ID object

The object type of Request ID object is TBA.

OBJECT	Type	Subtype(s)	Length
Request ID	TBA	0x01 Default	4

The Request ID object contains the information for identifying multiple LSPs of the same SMD.

SubType = 0x01 Default



Request ID
This four-octet quantity contains a request-id which is locally assigned by an upstream neighbor of a link.

4.1.2. Thread object

The object type of Loop Prevention object is TBA.

OBJECT	Type	Subtype(s)	Length
Thread	TBA	0x01 Default	12

The Thread object contains the information required for the thread mechanism.

SubType = 0x01 Default

Color
This eight-octet quantity contains a color of the thread. The first four-octet is the thread creator's IP address. The last four-octet is the local-id which is unique within the thread creator's IP address.

If a node does not require a loop prevention check but only requires an LSP length update, the special color "transparent" is defined by setting all zero's to the Color field. No acknowledgment is needed for transparent threads.

Length

This one octet quantity contains a thread length which is represented by a hop count from the furthest leaf node to the thread head. The value 0xff is assigned for "unknown" thread

length.

TTL (Time To Live)
This one octet quantity contains a thread TTL which is decremented by one (except for TTL="infinity") when a thread is extended without changing its color. When the TTL becomes zero, the extending procedure must be stopped. The value 0xff is assigned for "infinity" which is never decremented.

4.2 Advertisement class messages

4.2.1. Request message

Mandatory Objects
At least one of each mandatory object with associated object headers.

MANDATORY OBJECT	Type
SMD	0x02
Class of Service	0x04

Optional Objects
Zero or more optional objects with associated object headers.

OPTIONAL OBJECT	Type
Request ID	TBA
Thread	TBA

4.2.2. Mapping message

Mandatory Objects
At least one of each mandatory object with associated object headers.

MANDATORY OBJECT	Type
SMD	0x02
Label	0x03

Optional Objects
Zero or more optional objects with associated object headers.

OPTIONAL OBJECT	Type
Class of Service	0x04
Hop Count	0x06
MTU	0x07
Stack	0x08
Request ID	TBA

4.2.3. Update message

The message type of Update message is TBA.

Mandatory Objects
At least one of each mandatory object with associated object headers.

MANDATORY OBJECT	Type
SMD	0x02
Class of Service	0x04
Thread	TBA

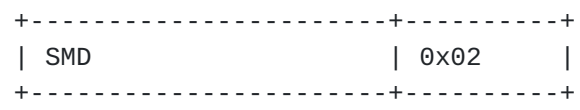
Optional Objects
Zero or more optional objects with associated object headers.

OPTIONAL OBJECT	Type
Request ID	TBA

4.2.4. Release message

Mandatory Objects
At least one of each mandatory object with associated object headers.

MANDATORY OBJECT	Type


SMD
0x02

Optional Objects
Zero or more optional objects with associated object headers.

OPTIONAL OBJECT	Type
Label	0x03
Request ID	TBA

4.2.5. Ack/Nak message

Mandatory Objects
At least one of each mandatory object with associated object headers.

MANDATORY OBJECT	Type
SMD	0x02
Error	0x01

Optional Objects

Zero or more optional objects with associated object headers.

OPTIONAL OBJECT	Type
Label	0x03
Request ID	TBA

5. Examples

In the following examples, we assume that the ingress-initiated ordered control is employed, that all the LSPs are with regard to the same FEC, and that all nodes are VC-merge capable.

5.1. First example

Consider an MPLS network shown in Fig. 1 in which an L3 loop exists. Each directed link represents the current next hop of the FEC at each node. Now leaf nodes R1 and R6 initiate creation of an LSP.

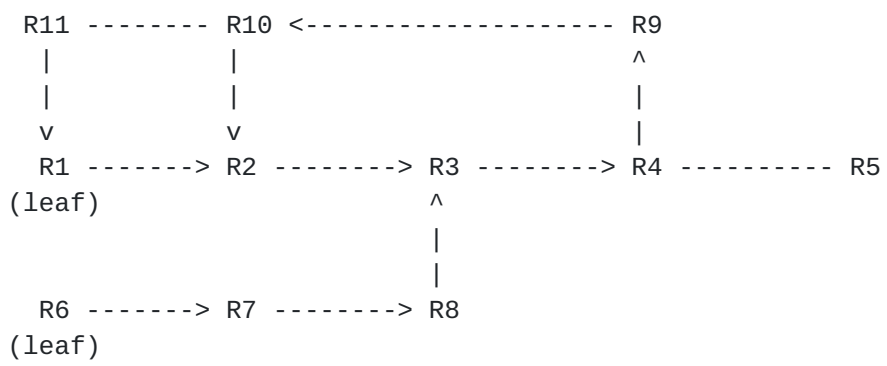


Fig. 1 Example MPLS network (1)

Assume that R1 and R6 sends Request messages at the same time, and that the initial thread TTL is 254 (255 represents "infinity"). First we show an example of how to prevent LSP loops before thread TTL becomes zero.

The Request message from R1 contains a thread of (red,1,254), where a thread is identified by (color,length,TTL). The Request message from R6 contains a thread of (blue,1,254).

Assume that R3 receives the Request originated from R1 before the Request originated from R6. Then R3 forwards the Request with the thread of (red,3,252) and then the Request with (blue,4,251) in this order.

When R2 receives the Request from R10 with the thread of (red,6,249), it detects a loop of the red thread. In this case, R2 creates a new purple thread of "unknown" length and extends it downstream by sending a Request with (purple,?,254) to R3, where "?" represents "unknown".

After that, R2 receives another Request for the same LSP from R10 with (blue,7,248). The blue thread is merged into the purple thread since the purple thread length (="unknown") is longer than the blue thread length (=7). R2 sends nothing to R3.

On the other hand, the purple thread goes round and R2 detects the loop of its own purple thread.

In this case, neither a thread is rewound nor a Mapping is returned. The current state of the network is shown in Fig. 2. Note that thread TTL information is not shown here.

Bl(L): blue thread with length L
Re(L): red thread with length L
Pu(L): purple thread with length L
*: position of thread head

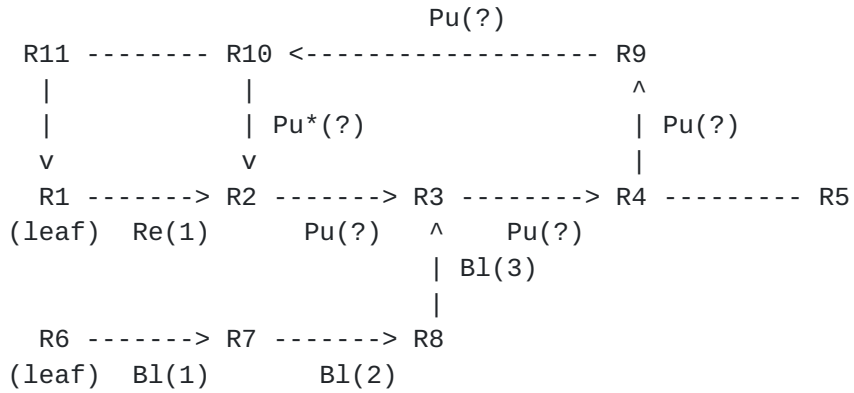


Fig. 2 The network state

Then R10 changes its next hop from R2 to R11.

Since R10 has a purple thread on the old downstream link, it first sends a Release message to the old next hop R2 for removing the purple thread. Next, it creates a new green thread for which the purple thread length(="unknown") is used, and sends a Request with (green,?,254) to R11.

When R2 receives the Release from R10, the upstream link between R10 and R2 is removed.

On the other hand, the green thread goes round to R10 without being merged.

When R10 receives the green thread, it sends a Release message to R11 to withdraw the green thread, since it is the creator of the green thread and there is no other actual incoming thread.

When R1 removes the green thread, it creates a new orange thread and resends a Request with (orange,0,254) to R2. The orange thread goes round to R1, replacing the green thread on the path. Finally, R1 detects the loop of its own orange thread.

The state of the network is now shown in Fig. 3.

Or(L): orange thread with length L
Bl(L): blue thread with length L
*: position of thread head

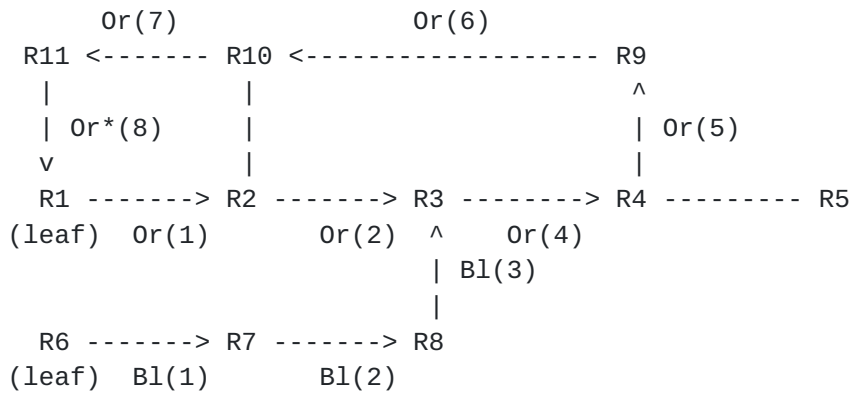


Fig. 3 The network state

Then R4 changes its next hop from R9 to R5.

Since R4 has the orange thread, it first sends a Release message to the old next hop R9 to withdraw the orange thread on the old route. Next, it creates a yellow thread of length 4, and sends a Request with (yellow,5,254) to R5.

Since R5 is the egress node, the received thread is assured to be loop-free, and R5 returns a Mapping message with a label. R5 sets the LSP length to 5.

The thread rewiding procedure is performed at each node, as the Mapping is returned upstream hop-by-hop.

Finally, when each of R1 and R6 receives a Mapping message, a merged LSP ((R1->R2),(R6->R7->R8))->R3->R4->R5) is established and all the colored threads disappear from the network.

5.2. Second example

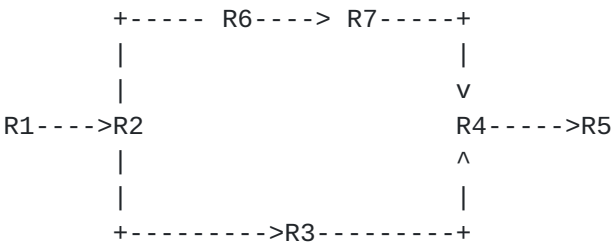


Fig. 4. Example MPLS network (2)

Assume that in Fig. 4, there is an established LSP R1->R2->R3->R4->R5, and the next hop changes at R2 from R3 to R6. R2 sends a Request to R6 with (red,2,254). When the Request with (red,4,252) reaches R4, it sends an Update message to R5 with (red,5,251) since the received thread length (=4) is longer than the current LSP length (=3).

When R5 receives the Update, it updates the LSP length to 5 and returns an ACK for the Update. When R4 receives the ACK for the Update, it returns an Mapping to R7.

When R2 receives the Mapping on the new route, it sends a Release to R3. When R4 receives the Release, it does not sends an Update to R5 since the LSP length does not change. Now an established LSP R1->R2->R6->R7->R4->R5 is obtained.

Then, the next hop changes again at R2 from R6 to R3.

R1 sends a Request with (blue,2,254) to R3. R3 forwards the Request with (blue,3,253) to R4.

When R4 receives the Request, it immediately returns a Mapping to R3 since the received thread length (=3) is not longer than the current LSP length (=4).

When R2 receives the Mapping on the new route, it sends a Release to R6. The Release reaches R4, triggering an Update message with a transparent thread (0,4,255) to R5, since the LSP length at R4 decreases from 4 to 3. R5 updates the LSP length to 4 without returning an ACK.

6. Comparison with path-vector/diffusion method

- Whereas the size of the path-vector increases with the length of the LSP, the sizes of the threads are constant. Thus the size of messages used by the thread algorithm are unaffected by the network size or topology. In addition, the thread merging

capability reduces the number of outstanding messages. These lead to improved scalability.

- In the thread algorithm, a node which is changing its next hop for a particular LSP must interact only with nodes that are between it and the LSP egress on the new path. In the path-vector algorithm, however, it is necessary for the node to initiate a diffusion computation that involves nodes which do not lie between it and the LSP egress.

  This characteristic makes the thread algorithm more robust. If a diffusion computation is used, misbehaving nodes which aren't even in the path can delay the path setup. In the thread algorithm, the only nodes which can delay the path setup are those nodes which are actually in the path.

- The thread algorithm is well suited for use with both the ingress-initiated ordered control and the egress-initiated ordered control. The path-vector/diffusion algorithm, however, is tightly coupled with the egress-initiated ordered control.

- The thread algorithm is retry-free, achieving quick path (re)configuration. The diffusion algorithm tends to delay the path reconfiguration time, since a node at the route change point must to consult all its upstream nodes.

- In the thread algorithm, the node can continue to use the old path if there is an L3 loop on the new path, as in the path-vector algorithm.

7. Security considerations

Security considerations are not discussed in this document.

8. Intellectual property considerations

Toshiba and/or Cisco may seek patent or other intellectual property protection for some of the technologies disclosed in this document. If any standards arising from this document are or become protected by one or more patents assigned to Toshiba and/or Cisco, Toshiba and/or Cisco intend to disclose those patents and license them on reasonable and non-discriminatory terms.

9. References

[1] Y. Ohba, et al., "Flow Attribute Notification Protocol Version 2 (FANPv2) Ingress Control Mode," Internet Draft,

draft-ohba-csr-fanpv2-icmode-00.txt, Dec. 1997.

[2] B. Davie, et al., "Use of Label Switching With ATM," Internet Draft, draft-davie-mpls-atm-01.txt, July 1998.

[3] E. Rosen, et al., "A Proposed Architecture for MPLS,"
Internet Draft, draft-ietf-mpls-arch-01.txt, July 1998.

[4] R. Callon, et al., "A Framework for Multiprotocol Label
Switching," Internet Draft, draft-ietf-mpls-framework-02.txt,
Nov. 1997.

[5] L. Andersson, et al., "Label Distribution Protocol," Internet Draft,
draft-mpls-ldp-spec-00.txt, March 1998.

Authors' Addresses

Yoshihiro Ohba
R&D Center, Toshiba Corp.
1, Komukai-Toshiba-cho, Saiwai-ku
Kawasaki, 210, Japan
Email: ohba@csl.rdc.toshiba.co.jp

Yasuhiro Katsube
R&D Center, Toshiba Corp.
1, Komukai-Toshiba-cho, Saiwai-ku
Kawasaki, 210, Japan
Email: katsube@isl.rdc.toshiba.co.jp

Eric Rosen
Cisco Systems, Inc.
250 Apollo Drive
Chelmsford, MA, 01824
Email: erosen@cisco.com

Paul Doolan
Ennovate Networks
330 Codman Hill Road
Boxborough, MA
Email: pdoolan@ennovatenetworks.com