Network Working Group
Request for Comments: 3063
Category: Experimental

Y. Ohba
Y. Katsube
Toshiba
E. Rosen
Cisco Systems
P. Doolan
Ennovate Networks

February 2001

MPLS Loop Prevention Mechanism

Status of this Memo

Copyright Notice

Abstract

This paper presents a simple mechanism, based on "threads", which can be used to prevent Multiprotocol Label Switching (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 ordered downstream-on-demand allocation or ordered downstream allocation. 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.

When an LSR finds that it has a new next hop for a particular FEC (Forwarding Equivalence Class) [1], 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.

For a given LSP, once a thread is extended to a particular next hop, no other thread is extended to that next hop unless there is a change in the hop count from the furthest upstream node. The only state information that needs to be associated with a particular next hop for a particular LSP is the thread color and hop count.

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 and 4 provide 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 hop count, but when the thread has been rewound, they need only remember its hop count.

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 ordered downstream on-demand label allocation or ordered downstream unsolicited label allocation [2,3]. The mechanism can also be applicable to loop detection, old path retention, and load-splitting.

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 network size. 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 (see section 9).

2. Basic definitions

LSP

We will use the term LSP to refer to a multipoint-to-point tree whose root is the egress node. See section 3.5 of [3].

In the following, we speak as if there were only a single LSP being set up in the network. This allows us to talk of incoming and outgoing links without constantly saying something like "for the same LSP.

Incoming Link, Upstream Link
Outgoing Link, Downstream Link

At a given node, a given LSP will have one or more incoming, or upstream links, and one outgoing or downstream link. A "link" is really an abstract relationship with an "adjacent" LSR; it is an "edge" in the "tree", and not necessarily a particular concrete entity like an "interface".

Leaf Node, Ingress Node

A node which has no upstream links.

Eligible Leaf Node

A node which is capable of being a leaf node. For example, a node is not an eligible leaf node if it is not allowed to directly inject L3 packets created or received at the node into its outgoing link.

Link Hop Count

Every link is labeled with a "link hop count". This is the number of hops between the given link and the leaf node which is furthest upstream of the given link. At any node, the link hop count for the downstream link is one more than the largest of the hop counts associated with the upstream links.

We define the quantity "Hmax" at a given node to be the maximum of all the incoming link hop counts. Note that, the link hop count of the downstream link is equal to Hmax+1. At a leaf node, Hmax is set to be zero.

An an example of link hop counts is shown in Fig.1.

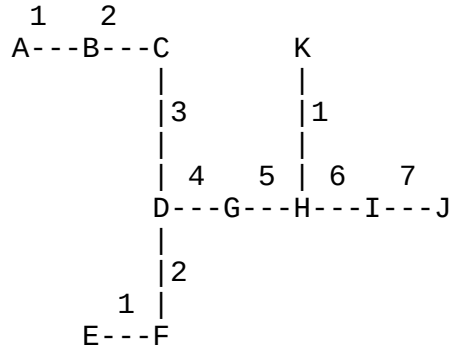


Fig.1 Example of link hop counts

Next Hop Acquisition

Node N thought that FEC F was unreachable, but now has a next hop for it.

Next Hop Loss

Node N thought that node A was the next hop for FEC F, but now no longer has the next hop for FEC F. A node loses a next hop whenever the next hop goes down.

Next Hop Change

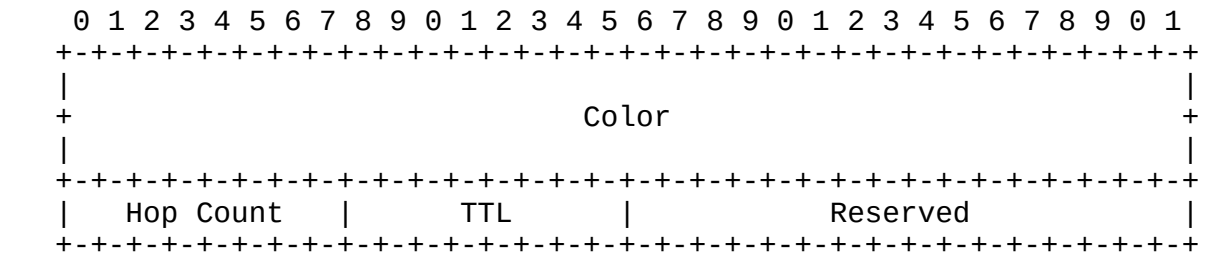
At node N, the next hop for FEC F changes from node A to node B, where A is different than B. A next hop change event can be seen as a combination of a next hop loss event on the old next hop and a next hop acquisition event on the new next hop.

3. Thread basics

A thread is a sequence of messages used to set up an LSP, in the "ordered downstream-on-demand" (ingress-initiated ordered control) style.

3.1. Thread attributes

There are three attributes related to threads. They may be encoded into a single thread object as:



Thread Color

Every time a path control message is initiated by a node, the node assigns a unique "color" to it. This color is to be unique in both time and space: its encoding consists of an IP address of the node concatenated with a unique event identifier from a numbering space maintained by the node. The path setup messages that the node sends downstream will contain this color. Also, when the node sends such a message downstream, it will remember the color, and this color becomes the color of the downstream link.

When a colored message is received, its color becomes the color of the incoming link. The thread which consists of messages of a certain color will be known as a thread of that color.

special color value "transparent"(=all 0's) is reserved.

One possible method for unique color assignment is, starting the event identifier from its initial value, and incrementing it by one (modulo its maximum value) each time a color is assigned. In this method, the initial event identifier is either selected at random or assigned to be larger than the largest event identifier used on the previous system incarnation.

Thread Hop Count

In order to maintain link hop counts, we need to carry hop counts in the path control messages. For instance, a leaf node would assign a hop count of 1 to its downstream link, and would store that value into a path setup message it sends downstream. When a path setup message is sent downstream, a node would assign a hop count which is one more than the largest of the incoming link hop counts, to its downstream link, and would store that value into a path setup message it sends downstream. Once the value is stored in a path control message, we may refer to it has a "thread hop count".

A special hop count value "unknown"(=0xff), which is larger than any other known value, is used when a loop is found. Once the thread hop count is "unknown", it is not increased any more as the thread is extended.

Thread TTL

To avoid infinite looping of control messages in some cases, a thread TTL is used. When a node creates a path control message and sends it downstream, it sets a TTL to the message, and the TTL is decremented at each hop. When the TTL reaches 0, the message is not forwarded any more. Unlike the thread hop counts and the thread colors, the thread TTLs do not needs to be stored in incoming links.

3.2. Thread loop

When the same colored thread is received on multiple incoming links, or the received thread color was assigned by the receiving node, it is said that the thread forms a loop. A thread creator can tell whether it assigned the received thread color by checking the IP address part of the received thread color.

3.3. Primitive thread actions

Five primitive actions are defined in order to prevent LSP loops by using threads: "extending", "rewinding", "withdrawing", "merging", and "stalling". This section describes only each primitive action and does not describe how these primitive actions are combined and how the algorithm totally works. The main body of the algorithm is described in section 4.

Thread Extending

When a node starts to send a path setup message to its next hop with a set of thread attributes, it is said that "the node creates a thread and extends it downstream". When a node receives a path setup message from an upstream node with a set of thread attributes and forwards it downstream, it is said that "the node receives a thread and extends it downstream". The color and hop count of the thread become the color and hop count of the outgoing link. Whenever a thread is received on a particular link, the color and hop count of that thread become the color and hop count of that incoming link, replacing any color and hop count that the link may have had previously.

For example, when an ingress node initiates a path setup, it creates a thread and extends it downstream by sending a path setup message. The thread hop count is set to be 1, and the thread color is set to be the ingress node's address with an appropriate event identifier, and the thread TTL is set to be its maximum value.

When a node receives a thread and extends it downstream, the node either (i) extends the thread without changing color, or (ii) extend the thread with changing color. The received thread is extended with changing color if it is received on a new incoming link and extended on an already existing outgoing link, otherwise, it is extended without changing color. When a thread is extended with changing color, a new colored thread is created and extended.

Thread creation does not occur only at leaf nodes. If an intermediate node has an incoming link, it will create and extend a new thread whenever it acquires a new next hop.

When a node notifies a next hop node of a decrease of the link hop count, if it is not extending a colored thread, a transparent thread is extended.

Thread Merging

When a node which has a colored outgoing link receives a new thread, it does not necessarily extend the new thread. It may instead 'merge' the new threads into the existing outgoing thread. In this case, no messages are sent downstream. Also, if a new incoming thread is extended downstream, but there are already other incoming threads, these other incoming threads are considered to be merged into the new outgoing thread.

Specifically, a received thread is merged if all the following conditions hold:

- A colored thread is received by node N, AND
- The thread does not form a loop, AND
- N is not an egress node, AND
- N's outgoing link is colored, AND
- N's outgoing link hop count is at least one greater than the hop count of the newly received thread.

When an outgoing thread rewinds (see below), any incoming threads which have been merged with it will rewind as well.

Thread Stalling

When a colored thread is received, if the thread forms a loop, the received thread color and hop count are stored on the receiving link without being extended. This is the special case of thread merging applied only for threads forming a loop and referred to as the "thread stalling", and the incoming link storing the stalled thread is called "stalled incoming link". A distinction is made between stalled incoming links and unstalled incoming links.

Thread Rewinding

When a thread reaches a node which satisfies a particular loop-free condition, the node returns an acknowledgment message back to the message initiator in the reverse path on which the thread was extended. The transmission of the acknowledgment messages is the "rewinding" of the thread.

The loop-free condition is:

- A colored thread is received by the egress node, OR
- All of the following conditions hold:
  (a) A colored thread is received by node N, AND
  (b) N's outgoing link is transparent, AND
  (c) N's outgoing link hop count is at least one greater than the hop count of the newly received thread.

When a node rewinds a thread which was received on a particular link, it changes the color of that link to transparent.

If there is a link from node M to node N, and M has extended a colored thread to N over that link, and M determines (by receiving a message from N) that N has rewound that thread, then M sets the color of its outgoing link to transparent. M then continues rewinding the thread, and in addition, rewinds any other incoming thread which had been merged with the thread being rewound, including stalled threads.

Each node can start label switching after the thread colors in all incoming and outgoing links becomes transparent.

Note that transparent threads are threads which have already been rewound; hence there is no such thing as rewinding a transparent thread.

Thread Withdrawing

It is possible for a node to tear down a path. A node tears down the portion of the path downstream of itself by sending teardown messages to its next hop. This process is known as the "thread withdrawing".

For example, suppose a node is trying to set up a path, and then experiences a next hop change or a next hop loss. It will withdraw the thread that it had extended down its old next hop.

If node M has extended a thread to node N, and node M then withdraws that thread, N now has one less incoming link than it had before. If N now has no other unstalled incoming links and N is not an eligible leaf node, it must withdraw its outgoing thread. If N still has an unstalled incoming link or N is an eligible leaf node, it may (or may not) need to change the hop count of the outgoing link.

N needs to change the outgoing hop count if:

- The incoming link hop count that was just removed had a larger hop count than any of the remaining incoming links, AND
- One of the following conditions holds:
  (a) The outgoing link is transparent, OR
  (b) The outgoing link has a known hop count.

If the outgoing link is transparent, it remains transparent, but the new hop count needs to be sent downstream. If the outgoing link is colored, a new thread (with a new color) needs to be created and extended downstream.

3.4. Examples of primitive thread actions

The following notations are used to illustrate examples of primitive actions defined for threads.

A pair of thread attributes stored in each link is represented by "(C,H)", where C and H represent the thread color and thread hop count, respectively.

A thread marked "+" indicates that it is created or received now. A thread marked "-" indicates that it is withdrawn now.

A link labeled with squared brackets (e.g., "[a]") indicates that it is an unstalled link. A link labeled with braces (e.g., "{a}") indicates that it is a stalled link.

Fig. 2 shows an example in which a leaf node A creates a blue thread and extends it downstream.

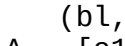


Fig.2 Thread extending at leaf node

Fig.3 shows an example of thread extending without changing color at intermediate node. Assume that a node B has no incoming and outgoing link before receiving a blue thread. When node B receives the blue thread of hop count 1 on a new incoming link i1, it extends the thread downstream without changing color (Fig.3(a)). After the blue thread is extended, node B receives a red thread of hop count unknown on incoming link i1 again (Fig.3(b)). The red thread is also extended without changing its color, since both i1 and o1 already exists.

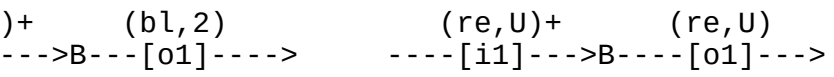


Fig.3(a)

Fig.3(b)

Fig.3 Thread extending without changing color

Fig.4 shows an example of thread extending with changing color. There are single incoming link i1 and single outgoing link o1 in Fig.4(a). Then a red thread of hop count 3 is received on a new incoming link i2. In this case, the received thread is extended with changing color, i.e., a new green thread is created and extended (Fig.4(b)), since o1 already exists.

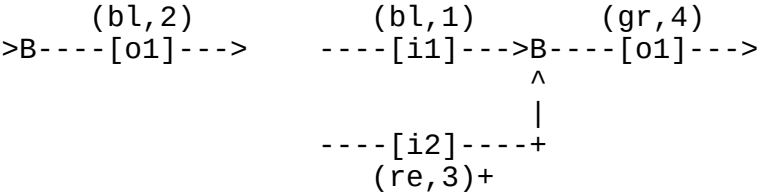


Fig.4(a)

Fig.4(b)

Fig.4 Thread extending with changing color

Fig.5 shows an example of thread merging. When a node B receives a red thread of hop count 3, the received thread is not extended since the outgoing link hop count is at least one greater than the received thread hop count. Both the red and blue threads will be rewound when the blue thread on outgoing link o1 is rewound.

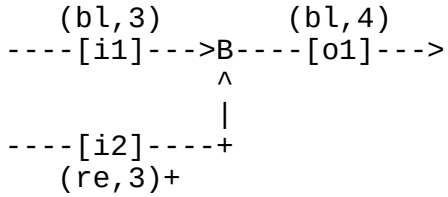


Fig.5 Thread merging

Figs 6 and 7 show examples of thread stalling. When a node B receives a blue thread of hop count 10 on incoming link i2 in Fig.6, it "stalls" the received thread since the blue thread forms a loop. In Fig.7, a leaf node A finds the loop of its own thread.

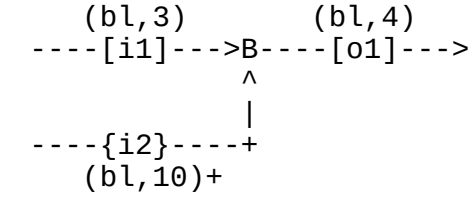


Fig.6 Thread stalling (1)

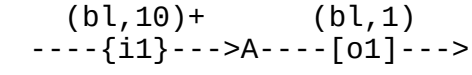


Fig.7 Thread stalling (2)

Fig.8 shows an example of thread rewinding. When the yellow thread which is currently being extended is rewound (Fig.8(a)), the node changes all the incoming and outgoing thread color to transparent, and propagates thread rewinding to upstream nodes (Fig.8(b)).

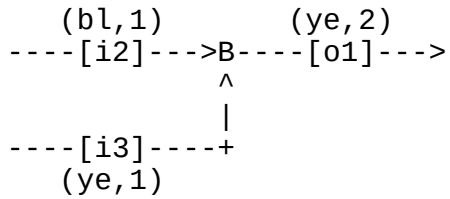


Fig.8(a)

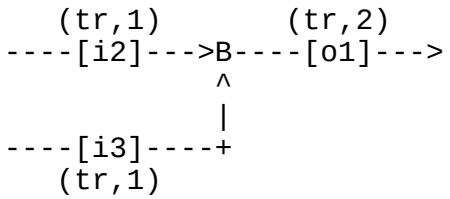


Fig.8(b)

Fig.8 Thread rewinding

Fig.9 shows an example of thread withdrawing. In Fig.9(a), the red thread on incoming link i2 is withdrawn. Then Hmax decreases from 3 to 1, and node B creates a new green thread and extends it downstream, as shown in Fig.9(b).

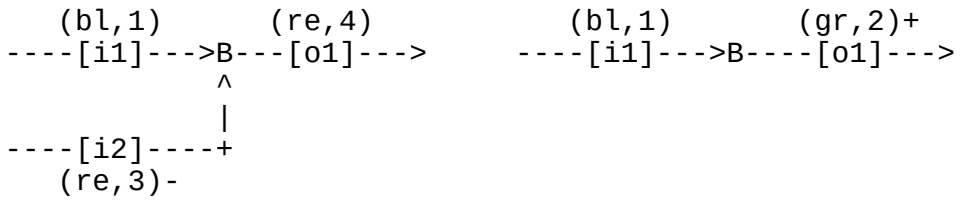


Fig.9(a) Fig.9(b)

Fig.9 Thread withdrawing (1)

Fig.10 shows another example of thread withdrawing. In Fig.10(a), the red thread on incoming link i3 is withdrawn. In this case, Hmax decreases from unknown to 1, however, no thread is extended as shown in Fig.10(b), since the outgoing link has a colored thread and the hop count is unknown.

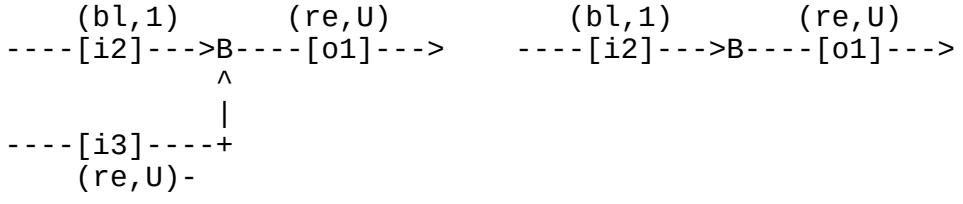


Fig.10(a) Fig.10(b)

Fig.10 Thread withdrawing (2)

Fig.11 shows another example of thread withdrawing. In Fig.11(a), the transparent thread on incoming link i3 is withdrawn. In this case, a transparent thread is extended (Fig.11(b)), since Hmax decreases and the outgoing link is transparent.

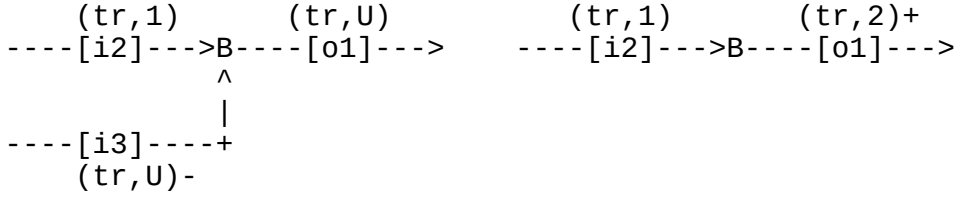


Fig.11(a) Fig.11(b)

Fig.11 Thread withdrawing (3)

4. Thread algorithm

The ordered downstream-on-demand allocation is assumed here, however, the algorithm can be adapted to the ordered downstream allocation, as shown in section 5.

In the algorithm, a next hop change event will be separated into two events: a next hop loss event on the old next hop and a next hop acquisition event on the new next hop, in this order.

The following notations are defined:

```
Hmax: the largest incoming link hop count
Ni:   the number of unstalled incoming links
```

The thread algorithm is described as follows.

When a node acquires a new next hop, it creates a colored thread and extends it downstream.

When a node loses a next hop to which it has extended a thread, it may withdraw that thread. As described in section 3, this may or may not cause the next hop to take some action. Among the actions the next hop may take are withdrawing the thread from its own next hop, or extending a new thread to its own next hop.

A received colored thread is either stalled, merged, rewound, or extended. A thread with TTL zero is never extended.

When a received thread is stalled at a node, if Ni=0 and the node is not an eligible leaf node, initiate a thread withdrawing. Otherwise, if Ni>0 and the received thread hop count is not unknown, a colored thread of hop count unknown is created and extended. If the received thread hop count is unknown, no thread is extended and no further action is taken.

When a thread being extended is rewound, if the thread hop count is greater than one more than Hmax, a transparent thread of hop count (Hmax+1) is extended downstream.

When a node that has an transparent outgoing link receives a transparent thread, if Hmax decreases the node extends it downstream without changing color.

5. Applicability of the algorithm

The thread algorithm described in section 4 can be applied to various LSP management policies.

5.1. LSP Loop prevention/detection

The same thread algorithm is applicable to both LSP loop prevention and detection.

In loop prevention mode, a node transmits a label mapping (including a thread object) for a particular LSP only when it rewinds the thread for that LSP. No mapping message is sent until the thread rewinds.

On the other hand, if a node operates in loop detection mode, it returns a label mapping message without a thread object immediately after receiving a colored thread. A node which receives a label mapping message that does not have a thread object will not rewind the thread.

5.2. 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 achieved simply by holding the label assigned to the downstream link on the old path until the thread being extended on the new route gets rewound. This is an implementation choice.

5.3. How to deal with ordered downstream allocation

The thread mechanism can be also adapted to ordered downstream allocation mode (or the egress-initiated ordered control) by regarding the event of newly receiving of a label mapping message [4] from the next hop as a next hop acquisition event.

Note that a node which doesn't yet have an incoming link 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.

5.4. How to realize load splitting

A leaf node can easily perform load splitting by setting up two different LSPs for the same FEC. The downstream links for the two LSPs are simply assigned different colors. The thread algorithm now prevents a loop in either path, but also allows the two paths to have a common downstream node.

If some intermediate node wants to do load splitting, the following modification is made. Assume that there are multiple next hops for the same FEC. If there are n next hops for a particular FEC, the set of incoming links for that FEC's LSP can be partitioned into n subsets, where each subset can be mapped to a distinct outgoing link.

This provides n LSPs for the FEC. Each such LSP uses a distinct color for its outgoing link. The thread algorithm now prevents a loop in any of the paths, but also allows two or more of the paths to have a common downstream node.

In this case, an interesting situation may happen. Let's say that in Fig.12, node B has two incoming links, i1 and i2, and two outgoing links, o1 and o2, such that i1 is mapped to o1, while i2 is mapped to o2.

If a blue thread received on i1 and extended on o1 is again received at node B on i2, the blue thread is not regarded as forming a loop, since i1 and i2 are regarded as belonging to different subsets. Instead, the blue thread received on i2 is extended on o2. If the thread extended on o2 is rewound, a single loop-free LSP which traverses node B twice is established.

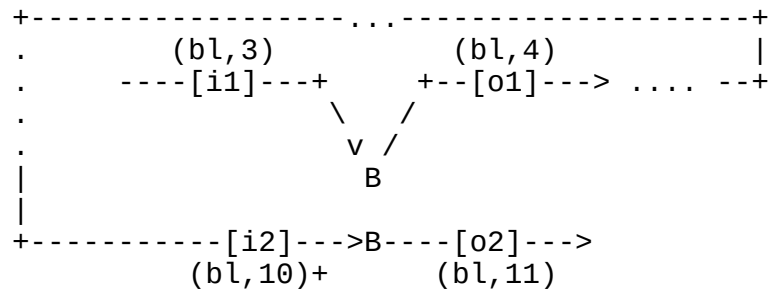


Fig.12 Load splitting at intermediate node

There is another type of load splitting, in which packets arrived at single incoming link can be label switched to any one of multiple outgoing links. This case does not seem to be a good load-splitting scheme, since the packet order in the same FEC is not preserved. Thus, this document does not focus on this case.

Whether that's a good type of load splitting or not, the fact remains that ATM-LSRs cannot load split like this because ATM switches just don't have the capability to make forwarding decisions on a per-packet basis.

6. Why this works

6.1. Why a thread with unknown hop count is extended

In the algorithm, a thread of unknown hop count is extended when a thread loop is detected. This reduces the number of loop prevention

messages by merging threads (of known hop count) that are flowing inside or outside the loop. See Appendix A.12.

6.2. Why a rewound thread cannot contain a loop

6.2.1. Case1: LSP with known link hop counts

How can we be sure that an established path does not contain a loop when the outgoing link hop count 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.)

Suppose that the thread hop count of the link between R1 and R2 is k. Then by the above procedures, the hop counts between Rn and R1 must be k+n-1. But the algorithm also ensures that if a node has an incoming hop count of j, its outgoing link hop count must be at least of j+1. Hence, if we assume that the LSP established as a result of thread rewinding contains a loop, the hop counts 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.

6.2.1. Case2: LSP with unknown link hop counts

An established path does not contain a loop as well, when the outgoing link hop count is "unknown". This is because a colored thread of unknown hop count is never rewound unless it reaches egress.

6.3. Why L3 loop is detected

Regardless of whether the thread hop count is known or unknown, if there is a loop, then some node in the loop will be the last node to receive a thread over a new incoming link. This thread will always arrive back at that node, without its color having changed. Hence the loop will always be detected by at least one of the nodes in the loop.

6.4. Why L3 loop is not mis-detected

Since no node ever extends the same colored thread downstream twice, a thread loop is not detected unless there actually is an L3 routing loop.

6.5. How a stalled thread automatically recovers from loop

Once a thread is stalled in a loop, the thread (or the path setup request) effectively remains in the loop, so that a path reconfiguration (i.e., thread withdrawing on the old path and thread extending on the new path) can be issued from any node that may receive a route change event so as to break the loop.

6.6. Why different colored threads do not chase each other

In the algorithm, multiple thread color and/or hop count updates may happen if several leaf nodes start extending threads at the same time. How can we prevent multiple threads from looping unlimitedly?

First, when a node finds that a thread forms a loop, it creates a new thread of hop count "unknown". All the looping threads of a known hop count which later arrive at the node would be merged into this thread. Such a thread behaves like a thread absorber.

Second, the "thread extending with changing color" prevents two threads from chasing each other.

Suppose that a received thread were always extended without changing color. Then we would encounter the following situation.

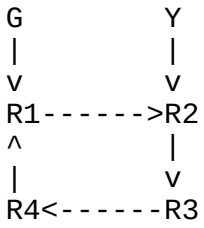


Fig.13 Example of thread chasing

In Fig.13, (1) node G acquires R1 as a next hop, and starts to extend a green thread of hop count 1, (2) node Y acquires R2 as a next hop, and starts to extend a yellow thread of hop count 1, and (3) both node G and node Y withdraws their threads before these threads go round.

In this case, the yellow and green threads would go round and get back to R2 and R1, respectively. When the threads get back to R2 and R1, however, the incoming links that store the yellow and green colors no longer exist. As a result, the yellow and green threads would chase each other forever in the loop.

However, since we have the "extending with changing color" mechanism, this does not actually happen. When a green thread is received at R2, R2 extends the thread with changing color, i.e., creates a new red thread and extends it. Similarly, when a yellow thread is received at R1, R1 creates a new purple thread and extends it. Thus, the thread loop is detected even after node G and node Y withdraw threads. This ensures that a thread is extended around the loop which has a color assigned by some node that is in the loop.

There is at least one case even the "extending with changing color" mechanism cannot treat, that is, the "self-chasing" in which thread extending and thread withdrawing with regard to the same thread chase each other in a loop. This case would happen when a node withdraw a thread immediately after extending it into an L3 loop.

A heuristics for self-chasing is to delay the execution of thread withdrawing at an initiating node of the thread withdrawing. Anyway, the thread TTL mechanism can eliminate any kind of thread looping.

7. Loop prevention examples

In this section, we show two examples to show how the algorithm can prevent LSP loops in given networks.

We assume that the ordered downstream-on-demand allocation is employed, that all the LSPs are with regard to the same FEC, and that all nodes are VC-merge capable.

7.1. First example

Consider an MPLS network shown in Fig.14 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.

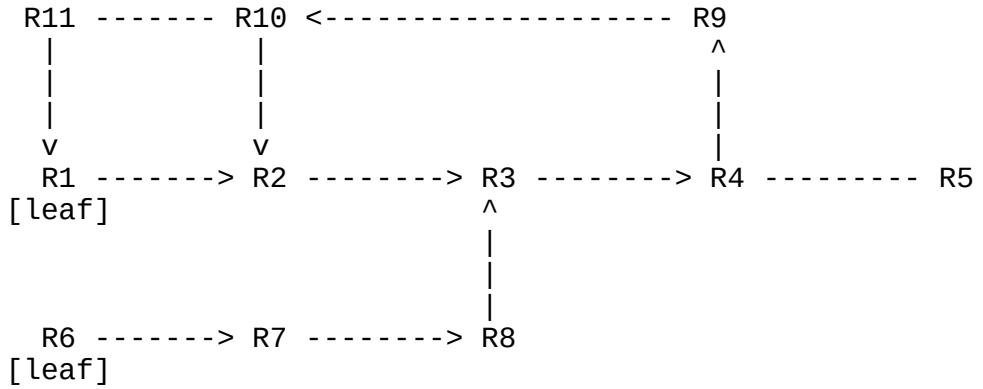


Fig. 14 Example MPLS network (1)

Assume that R1 and R6 send a label request message at the same time, and that the initial thread TTL is 255. First we show an example of how to prevent LSP loops.

A set of thread attributes is represented by (color, hop count, TTL).

The request from R1 and R6 contains (re,1,255) and (bl,1,255), respectively.

Assume that R3 receives the request originated from R1 before receiving the request originated from R6. When R3 receives the first request with red thread, R3 forwards it with (re,3,253) without changing thread color, since both the receiving incoming link and the outgoing link are newly created. Then R3 receives the second request with blue thread. In this time, the outgoing link is already exists. Thus, R3 performs thread extending with changing color, i.e., creates a new brown thread and forwards the request with (br,4,255).

When R2 receives the request from R10 with (re,6,250), it finds that the red thread forms a loop, and stalls the red thread. Then, R2 creates a purple thread of hop count unknown and extends it downstream by sending a request with (pu,U,255) to R3, where "U" represents "unknown".

After that, R2 receives another request from R10 with (br,7,252). The brown thread is merged into purple thread. R2 sends no request to R3.

On the other hand, the purple thread goes round without changing color through existing links, and R2 finds the thread loop and stalls the purple thread. Since the received thread hop count is unknown, no thread is created any more. In this case no thread rewinding occurs. The current state of the network is shown in Fig.15.

*: location of thread stalling

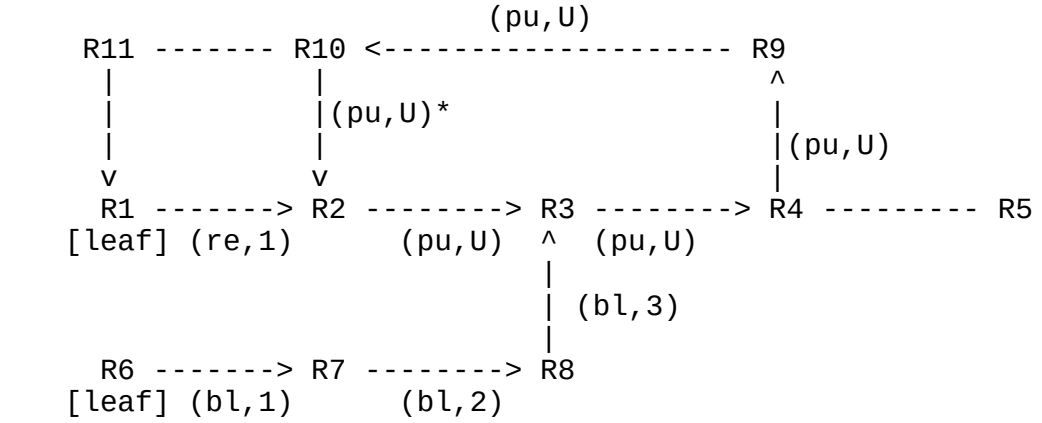


Fig.15 The network state

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 path teardown message to the old next hop R2 for withdrawing the purple thread. Next, it creates a green thread of hop count unknown and sends a request with (gr,U,255) to R11.

When R2 receives the teardown message from R10, R2 removes the stalled incoming link between R10 and R2.

On the other hand, the green thread reaches R1 and Hmax is updated from zero to unknown. In this case, R1 performs thread extending with changing color since the thread is received on a new incoming link but extended on the already existing outgoing link. As a result, R1 creates an orange thread of hop count unknown and extend it to R2.

The orange thread goes round through existing links without changing color, and finally it is stalled at R1.

The state of the network is now shown in Fig.16.

*: location of thread stalling

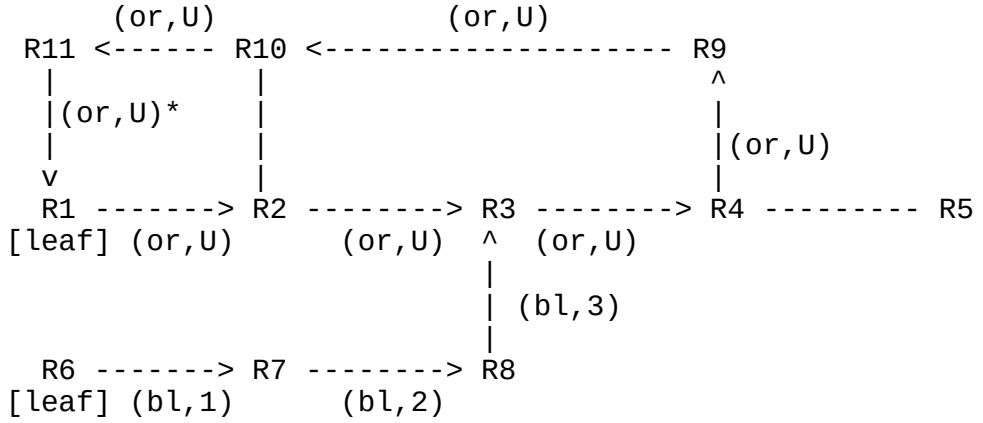


Fig.16 The network state

Then R4 changes its next hop from R9 to R5.

Since R4 is extending an orange thread, it first sends a teardown message to the old next hop R9 to withdraw the orange thread on the old route. Next, it creates a yellow thread of hop count unknown, and sends a request message with (ye,U,255) to R5.

Since R5 is the egress node, the yellow thread rewinding starts. R5 returns a label mapping message. The thread rewinding procedure is performed at each node, as the label mapping message is returned upstream hop-by-hop.

If R1 receives a label mapping message before receiving the orange thread's withdrawal from R11, R1 returns a label mapping message to R11. On receiving the orange thread's withdrawal, R1 will create a transparent thread and extend it by sending an update message with (tr,1,255) in order to notify downstream of the known hop count.

Otherwise, if R1 receives the orange thread's withdrawal before receiving a label mapping message, R1 removes the stalled incoming orange link and waits for rewinding of the outgoing orange thread. Finally, when R1 receives a label mapping message from R2, it creates a transparent thread (tr,1,255) and extend it downstream.

In both cases, a merged LSP ((R1->R2),(R6->R7->R8))->R3->R4->R5) is established and every node obtains the correct link hop count. The final network state is shown in Fig.17.

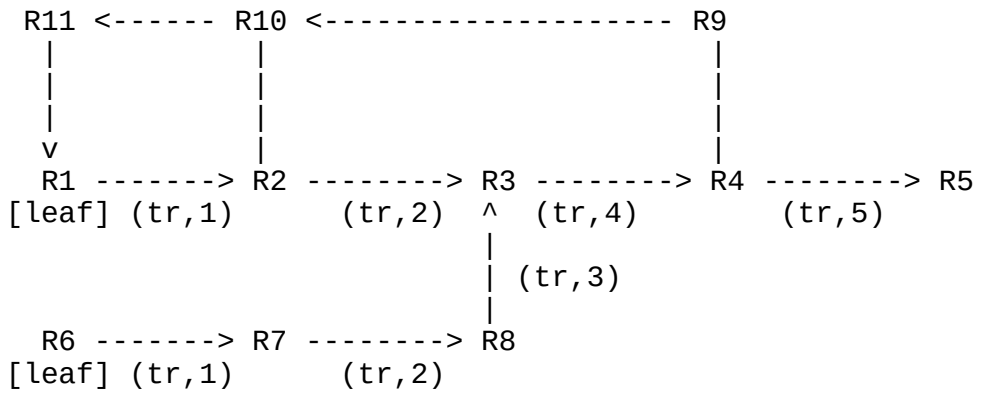


Fig.17 The final network state

7.2. Second example

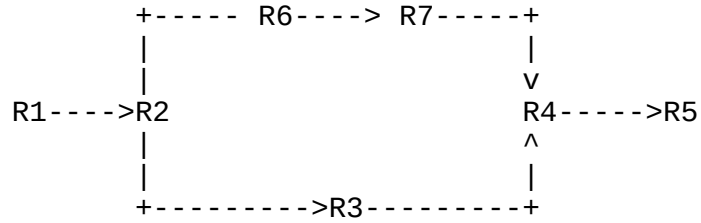


Fig.18 Example MPLS network (2)

Assume that in Fig.18, 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 a red thread (re,2,255). When the request with (re,4,253) reaches R4, it extends the thread to R5 with changing color. Thus, a new green thread is created at R4 and extended to R5 by sending an update message with (gr,5,255).

When R5 receives the update, it updates the incoming link hop count to 5 and returns an ack (or a notification message with a success code) for the update. When R4 receives the ack for the update, it returns a label mapping message to R7.

When R2 receives the label mapping message on the new route, it sends a teardown message to R3. When R4 receives the teardown message, it does not sends an update to R5 since Hmax 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.

R2 sends a request with a blue thread (bl,2,255) to R3. R3 forwards the request with (bl,3,254) to R4.

When R4 receives the request, it immediately returns a label mapping message to R3 since Hmax does not change.

When R2 receives the label mapping message on the new route, it sends a teardown message to R6. The teardown message reaches R4, triggering an update message with a transparent thread (tr,4,255) to R5, since Hmax decreases from 4 to 3. R5 updates the incoming link hop count to 4 without returning an ack.

8. Thread control block

A thread control block (TCB) is maintained per LSP at each node and may contain the following information:

- FEC
- State
- Incoming links
  Each incoming link has the following attributes:
  - o neighbor: upstream neighbor node address
  - o color: received thread color
  - o hop count: received thread hop count
  - o label
  - o S-flag: indicates a stalled link
- Outgoing links
  Each outgoing link has the following attributes:
  - o neighbor: downstream neighbor node address
  - o color: received thread color
  - o hop count: received thread hop count
  - o label
  - o C-flag: indicates the link to the current next hop

If a transparent thread is received on an incoming link for which no label is assigned yet or a non-transparent color is stored, discard the thread without entering the FSM. An error message may be returned to the sender.

Whenever a thread is received on an incoming link, the following actions are taken before entering the FSM: (1) Store the received thread color and hop count on the link, replacing the old thread color and hop count, and (2) set the following flags that are used for an event switch within "Recv thread" event (see section 8.1).

- Color flag (CL-flag):
  Set if the received thread is colored.
- Loop flag (LP-flag):
  Set if the received thread forms a loop.
- Arrived on new link flag (NL-flag):
  Set if the received thread arrives on a new incoming link.

If LP-flag is set, there must be an incoming link L, other than the receiving link, which stores the same thread color as the received one. The TCB to which link L belongs is referred to as the "detecting TCB". If the receiving LSR is VC-merge capable, the detecting TCB and the receiving TCB is the same, otherwise, the two TCBs are different.

Before performing a thread extending, the thread TTL is decremented by one. If the resulting TTL becomes zero, the thread is not extended but silently discarded. Otherwise, the thread is extended and the extended thread hop count and color are stored into the outgoing link.

When a node receives a thread rewinding event, if the received thread color and the extending thread color are different, it discards the event without entering the FSM.

8.1. Finite state machine

An event which is "scheduled" by an action in an FSM must be passed immediately after the completion of the action.

The following variables are used in the FSM:

- Ni: number of unstalled incoming links
- Hmax: largest incoming hop count
- Hout: hop count of the outgoing link for the current next hop
- Hrec: hop count of the received thread

In the FSM, if Hmax=unknown, the value for (Hmax+1) becomes the value reserved for unknown hop count plus 1. For example, if Hmax=unknown=255, the value (Hmax+1) becomes 256.

A TCB has three states; Null, Colored, and Transparent. When a TCB is in state Null, there is no outgoing link and Ni=0. The state Colored means that the node is extending a colored thread on the outgoing link for the current next hop. The state Transparent means that the node is the egress node or the outgoing link is transparent.

The flag value "1" represents the flag is set, "0" represents the flag is not set, and "*" means the flag value is either 1 or 0.

The FSM allows to have one transparent outgoing link on the old next hop and one colored outgoing link on the current next hop. However, it is not allowed to have a colored outgoing link on the old next hop.

State Null:

Event	Recv thread Flags			Action	New state
	CL	LP	NL		
	0	*	*	Do nothing.	No change
	1	0	*	If the node is egress, start thread rewinding and change the color of the receiving link to transparent.	Transparent
				Otherwise, extend the received thread without changing color.	Colored
	1	1	*	Stall the received thread; if Hrec<unknown, schedule "Reset to unknown" event for the detecting TCB.	No change
Next hop acquisition				If eligible-leaf, create a colored thread and extend it.	Colored
Others				Silently ignore the event.	No change

State Colored:

Event	Recv thread Flags			Action	New state
	CL	LP	NL		
	0	*	*	If Hmax+1<Hout<unknown, create a colored thread and extend it. Otherwise, do nothing.	No change
	1	0	*	If Hmax<Hout, merge the received thread. Otherwise, extend the thread with (if NL=1) or without (if NL=0) changing color.	No change
	1	1	*	Stall the received thread.	
				If Ni=0 and the node is not an eligible leaf, initiate thread withdrawing.	Null
				If Ni>0 and Hrec<unknown, schedule "Reset to unknown" event for the detecting TCB.	No change
				Otherwise, do nothing.	No change

Rewound	Propagate thread rewinding to previous hops that are extending a colored thread; change the colors stored in all incoming and outgoing links to transparent; if Hmax+1<Hout, extend transparent thread. Withdraw the thread on the outgoing link for which C-flag=0.	Transparent
Withdrawn	Remove the corresponding incoming link.	
	If Ni=0 and the node is not an eligible leaf, propagate thread withdrawing to all next hops.	Null
	Otherwise, if Hmax+1<Hout<unknown, create a colored thread and extend it.	No change
	Otherwise, do nothing.	No change
Next hop acquisition	If there is already an outgoing link for the next hop, do nothing. (This case happens only when the node retains the old path.)	Transparent
	Otherwise, create a colored thread and extend it.	No change
Next hop loss	If the outgoing link is transparent and the node is allowed to retain the link and the next hop is alive, do nothing.	No change
	Otherwise, take the following actions. Initiate thread withdrawing for the next hop; if the node becomes a new egress, schedule "Rewound" event for this TCB.	
	If Ni=0, move to Null.	Null
	Otherwise, do nothing.	No change
Reset to unknown	Create a colored thread of hop count unknown and extend it.	No change
Others	Silently ignore the event.	No change

State Transparent:

Event Recv thread Flags CL	LP	NL	Action	New state
0	*	*	If Hmax+1<Hout, extend a transparent thread.	No change
1	0	*	If the node is egress or if Hmax<Hout, change the color of the receiving link to transparent and start thread rewinding.	No change
			Otherwise, extend the thread with (if NL=1) or without (if NL=0) changing color.	Colored

Withdrawn	Remove the corresponding incoming link.	
	If Ni=0 and the node is not an eligible leaf, propagate thread withdrawing to next hops.	Null
	Otherwise, if Hmax+1<Hout, create a transparent thread and extend it.	No change
	Otherwise, do nothing.	No change
Next hop acquisition	Create a colored thread and extend it.	Colored
Next hop loss	If the node is allowed to retain the outgoing link and the next hop is alive, do nothing.	No change
	Otherwise, take the following actions.	
	Initiate thread withdrawing.	
	If Ni=0, move to Null.	Null
	Otherwise, do nothing.	No change
Others	Silently ignore the event.	No change

9. 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 ordered downstream-on-demand allocation and ordered downstream allocation. The path-vector/diffusion algorithm, however, is tightly coupled with the ordered downstream allocation.

- 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.

10. Security Considerations

The use of the procedures specified in this document does not have any security impact other than that which may generally be present in the use of any MPLS procedures.

11. 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.

12. Acknowledgments

We would like to thank Hiroshi Esaki, Bob Thomas, Eric Gray, and Joel Halpern for their comments.

13. Authors' Addresses

Yoshihiro Ohba
Toshiba Corporation
1, Komukai-Toshiba-cho, Saiwai-ku
Kawasaki 210-8582, Japan

EMail: yoshihiro.ohba@toshiba.co.jp

Yasuhiro Katsube
Toshiba Corporation
1, Toshiba-cho, Fuchu-shi,
Tokyo, 183-8511, Japan

EMail: yasuhiro.katsube@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 Rd
Marlborough MA 01719

EMail: pdoolan@ennovatenetworks.com

14. References

[1] Callon, R., et al., "A Framework for Multiprotocol Label Switching", Work in Progress.

[2] Davie, B., Lawrence, J., McCloghrie, K., Rosen, E., Swallow, G., Rekhter, Y. and P. Doolan, "MPLS using LDP and ATM VC Switching", RFC 3035, January 2001.

[3] Rosen, E., et al., "A Proposed Architecture for MPLS", Work in Progress.

[4] Andersson, L., Doolan, P., Feldman, N., Fredette, A. and B. Thomas, "LDP Specification", RFC 3036, January 2001.

Appendix A - Further discussion of the algorithm

The purpose of this appendix is to give a more informal and tutorial presentation of the algorithm, and to provide some of the motivation for it. For the precise specification of the algorithm, the FSM should be taken as authoritative.

As in the body of the document, we speak as if there is only one LSP; otherwise we would always be saying "... of the same LSP". We also consider only the case where the algorithm is used for loop prevention, rather than loop detection.

A.1. Loop Prevention the Brute Force Way

As a starting point, let’s consider an algorithm which we might call "loop prevention by brute force". In this algorithm, every path setup attempt must go all the way to the egress and back in order for the path to be setup. This algorithm is obviously loop-free, by virtue of the fact that the setup messages actually made it to the egress and back.

Consider, for example, an existing LSP B-C-D-E to egress node E. Now node A attempts to join the LSP. In this algorithm, A must send a message to B, B to C, C to D, D to E. Then messages are sent from E back to A. The final message, from B to A, contains a label binding, and A can now join the LSP, knowing that the path is loop-free.

Using our terminology, we say that A created a thread and extended it downstream. The thread reached the egress, and then rewound.

We needn’t assume, in the above example, that A is an ingress node. It can be any node which acquires or changes its next hop for the LSP in question, and there may be nodes upstream of it which are also trying to join the LSP.

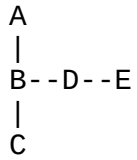
It is clear that if there is a loop, the thread never reaches the egress, so it does not rewind. What does happen? The path setup messages just keep traveling around the loop. If one keeps a hop count in them, one can ensure that they stop traveling around the loop when the hop count reaches a certain maximum value. That is, when one receives a path setup message with that the maximum hop count value, one doesn’t send a path setup message downstream.

How does one recover from this situation of a looping thread? In order for L3 routing to break the loop, some node in the loop MUST experience a next hop change. This node will withdraw the thread

from its old next hop, and extend a thread down its new next hop. If there is no longer a loop, this thread now reaches the egress, and gets rewound.

A.2. What's Wrong with the Brute Force Method?

Consider this example:



If A and C both attempt to join the established B-D-E path, then B and D must keep state for both path setup attempts, the one from A and the one from C. That is, D must keep track of two threads, the A-thread and the C-thread. In general, there may be many more nodes upstream of B who are attempting to join the established path, and D would need to keep track of them all.

If VC merge is not being used, this isn't actually so bad. Without VC merge, D really must support one LSP for each upstream node anyway. If VC merge is being used, however, supporting an LSP requires only that one keep state for each upstream link. It would be advantageous if the loop prevention technique also required that the amount of state kept by a node be proportional to the number of upstream links which thenode has, rather than to the number of nodes which are upstream in the LSP.

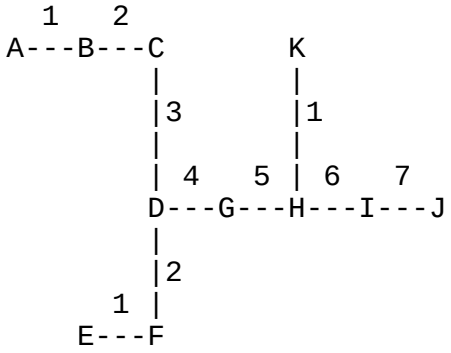
Another problem is that if there is a loop, the setup messages keep looping. Even though a thread has traversed some node twice, the node has no way to tell that a setup message it is currently receiving is part of the same thread as some setup message it received in the past.

Can we modify this brute force scheme to eliminate these two problems? We can. To show how to do this, we introduce two notions: thread hop count, and thread color.

A.3. Thread Hop Count

Suppose every link in an LSP tree is labeled with the number of hops you would traverse if you were to travel backwards (upstream) from that link to the leaf node which is furthest upstream of the link.

For example, the following tree would have its links labeled as follows:



Call these the "link hop counts".

Links AB, EF, KH are labeled one, because you can go only one hop upstream from these links. Links BC, and FD are labeled 2, because you can go 2 hops upstream from these links. Link DG is labeled 4, because it is possible to travel 4 hops upstream from this link, etc.

Note that at any node, the hop count associated with the downstream link is one more than the largest of the hop counts associated with the upstream links.

Let’s look at a way to maintain these hop counts.

In order to maintain the link hop counts, we need to carry hop counts in the path setup messages. For instance, a node which has no upstream links would assign a hop count of 1 to its downstream link, and would store that value into the path setup messages it sends downstream. Once the value is stored in a path setup message, we may refer to it has a "thread hop count".

When a path setup message is received, the thread hop count is stored as the link hop count of the upstream link over which the message was received.

When a path setup message is sent downstream, the downstream link’s hop count (and the thread hop count) is set to be one more than the largest of the incoming link hop counts.

Suppose a node N has some incoming links and an outgoing link, with hop counts all set properly, and N now acquires a new incoming link. If, and only if, the link hop count of the new incoming link is greater than that of all of the existing incoming links, the downstream link hop count must be changed. In this case, control messages must be sent downstream carrying the new, larger thread hop count.

If, on the other hand, N acquires a new incoming link with a link hop count that is less than or equal to the link hop count of all existing incoming links, the downstream link hop count remains unchanged, and no messages need be sent downstream.

Suppose N loses the incoming link whose hop count was the largest of any of the incoming links. In this case, the downstream link hop count must be made smaller, and messages need to be sent downstream to indicate this.

Suppose we were not concerned with loop prevention, but only with the maintenance of the hop counts. Then we would adopt the following rules to be used by merge points:

A.3.1 When a new incoming thread is received, extend it downstream if and only if its hop count is the largest of all incoming threads.

A.3.2 Otherwise, rewind the thread.

A.3.3 An egress node would, of course, always rewind the thread.

A.4. Thread Color

Nodes create new threads as a result of next hop changes or next hop acquisitions. Let’s suppose that every time a thread is created by a node, the node assigns a unique "color" to it. This color is to be unique in both time and space: its encoding consists of an IP address of the node concatenated with a unique event identifier from a numbering space maintained by the node. The path setup messages that the node sends downstream will contain this color. Also, when the node sends such a message downstream, it will remember the color, and this color becomes the color of the downstream link.

When a colored message is received, its color becomes the color of the incoming link. The thread which consists of messages of a certain color will be known as a thread of that color.

When a thread is rewound (and a path set up), the color is removed. The links become transparent, and we will sometimes speak of an established LSP as being a "transparent thread".

Note that packets cannot be forwarded on a colored link, but only on a transparent link.

Note that if a thread loops, some node will see a message, over a particular incoming link, with a color that the node has already seen before. Either the node will have originated the thread of that color, or it will have a different incoming link which already has

that color. This fact can be used to prevent control messages from looping. However, the node would be required to remember the colors of all the threads passing through it which have not been rewound or withdrawn. (I.e., it would have to remember a color for each path setup in progress.)

A.5. The Relation between Color and Hop Count

By combining the color mechanism and the hop count mechanism, we can prevent loops without requiring any node to remember more than one color and one hop count per link for each LSP.

We have already stated that in order to maintain the hop counts, a node needs to extend only the thread which has the largest hop count of any incoming thread. Now we add the following rule:

A.5.1 When extending an incoming thread downstream, that thread's color is also passed downstream (I.e., the downstream link's color will be the same as the color of the upstream link with largest hop count.)

Note that at a given node, the downstream link is either transparent or it has one and only one color.

A.5.2 If a link changes color, there is no need to remember the old color.

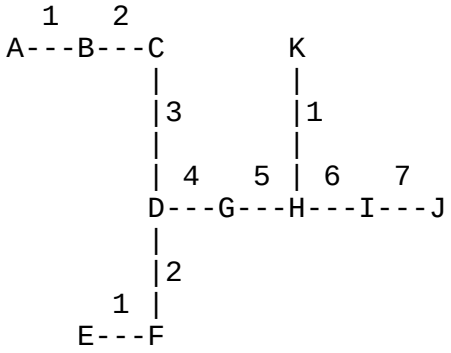
We now define the concept of "thread merging":

A.5.2 Suppose a colored thread arrives at a node over an incoming link, the node already has an incoming thread with the same or larger hop count, and the node has an outgoing colored thread. In this case, we may say that the new incoming thread is "merged" into the outgoing thread.

Note that when an incoming thread is merged into an outgoing thread, no messages are sent downstream.

A.6. Detecting Thread Loops

It can now be shown that if there is a loop, there will always either be some node which gets two incoming threads of the same color, or the colored thread will return to its initiator. In this section, we give several examples that may provide an intuitive understanding of how the thread loops are detected.



Returning to our previous example, let's set what would happen if H changed its next hop from I to E. H now creates a new thread, and assigns it a new color, say, red. Since H has two incoming link, with hop counts 1 and 5 respectively, it assigns hop count 6 to its new downstream link, and attempts a path setup through E.

E now has an incoming red thread with hop count 6. Since E's downstream link hop count is now only 1, it must extend the red thread to F, with hop count 7. F then extends the red thread to D with hop count 8, D to G with hop count 9, and G to H with hop count 10.

The red thread has now returned to its initiator, and the loop is detected.

Suppose though that before the red thread makes it back to H, G changes its next hop from H to E. Then G will extend the red thread to E. But E already has an incoming red link (from H), so the loop is detected.

Let's now define the notion of a "stalled thread". A stalled thread is a thread which is merged into the outgoing thread, even though the outgoing thread has a smaller link hop count.

When a thread loop is detected, the thread becomes stalled.

A.6.1 When a loop is detected due to a thread of a particular color traversing some node twice, we will say that the thread is "stalled" at the node. More precisely, it is the second appearance of the thread which is stalled. Note that we say that a thread is traversing a node twice if the thread is received by that node on an incoming link, but either there is another incoming link with the same color, or the color is one that was assigned by the node itself.

A.7. Preventing the Setup of Looping LSPS

The mechanism to be used for preventing the setup of looping LSPs should now be obvious. If node M is node N's next hop, and N wishes to set up an LSP (or to merge into an LSP which already exists at M), then N extends a thread to M.

M first checks to see if the thread forms a loop (see Appendix A.6), and if so, the thread is stalled. If not, the following procedure is followed.

A.7.1 If M receives this thread, and M has a next hop, and either:

- M has no outgoing thread

- the incoming thread hop count is larger than the hop count of all other incoming threads,

then M must extend the thread downstream.

A.7.2 On the other hand, if M receives this thread, and M has a next hop and there is another incoming thread with a larger hop count, then:

A.7.2.1 if the outgoing thread is transparent, M rewinds the new incoming thread.

A.7.2.2 if the outgoing thread is colored, M merges the new incoming thread into the outgoing thread, but does not send any messages downstream.

A.7.3 If M has not already assigned a label to N, it will assign one when, and only when, M rewinds the thread which N has extended to it.

A.7.4 If M merges the new thread into an existing colored outgoing thread, then the new incoming thread will rewind when, and only when, the outgoing thread rewinds.

A.8. Withdrawing Threads

A.8.1 If a particular node has a colored outgoing thread, and loses or changes its next hop, it withdraws the outgoing thread.

Suppose that node N is immediately upstream of node M, and that N has extended a thread to M. Suppose further that N then withdraws the thread.

A.8.2 If M has another incoming thread with a larger hop count, then M does not send any messages downstream.

A.8.3 However, if the withdrawn thread had the largest hop count of any incoming thread, then M's outgoing thread will no longer have the proper hop count and color. Therefore:

A.8.3.1 M must now extend downstream the incoming thread with the largest hop count. (This will cause it to forget the old downstream link hop count and color.)

A.8.3.2 The other incoming threads are considered to be merged into the thread which is extended.

A.8.4 When the last unstalled incoming thread is withdrawn, the outgoing thread must be withdrawn.

A.9. Modifying Hop Counts and Colors of Existing Threads

We have seen the way in which the withdrawal of a thread may cause hop count and color changes downstream. Note that if the hop count and/or color of an outgoing thread changes, then the hop count and color of the corresponding incoming thread at the next hop will also change. This may result in a color and/or next hop change of the outgoing thread at that next hop.

A.9.1 Whenever there is a hop count change for any incoming thread, a node must determine whether the "largest hop count of any incoming thread" has changed as a result. If so, the outgoing thread's hop count, and possibly color, will change as well, causing messages to be sent downstream.

A.10. When There is No Next Hop

A.10.1 If a particular node has a colored incoming thread, but has no next hop (or loses its next hop), the incoming thread is stalled.

A.11. Next Hop Changes and Pre-existing Colored Incoming Threads

It is possible that a node will experience a next hop change or a next hop acquisition at a time when it has colored incoming threads. This happens when routing changes before path setup is complete.

A.11.1 If a node has a next hop change or a next hop acquisition at a time when it has colored incoming threads, it will create a thread with a new color, but whose hop count is one more than the largest of the incoming link hop counts. It will then extend this thread downstream.

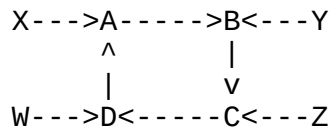
A.11.2 When this new thread is created and extended downstream, all incoming threads are merged into it. Any incoming threads that were previously stalled are now considered to be "merged" rather than "stalled".

That is, even though the outgoing thread has a different color than any of the incoming threads, the pre-existing incoming threads are all considered to have been merged into the new outgoing thread. This means that when the outgoing thread rewinds, the incoming threads will too.

Note: it is still required to distinguish stalled incoming links from unstalled incoming links when thread withdrawing is performed.

A.12. How Many Threads Run Around a Loop?

We have seen that when a loop is detected, the looping thread stalls. However, considering the following topology:



In this example, there is a loop A-B-C-D-A. However, there are also threads entering the loop from X, Y, Z, and W. Once the loop is detected, there really is no reason why any other thread should have to wrap around the loop. It would be better to simply mark presence of the loop in each node.

To do this, we introduce the notion of the "unknown" hop count, U. This hop count value is regarded as being larger than any other hop count value. A thread with hop count U will be known as a "U-thread".

A.12.1 When an incoming thread with a known hop count stalls, and there is an outgoing thread, we assign the hop count U to the outgoing thread, and we assign a new color to the outgoing thread as well.

As a result, the next hop will then have an incoming U-thread, with the newly assigned color. This causes its outgoing thread in turn to be assigned hop count U and the new color. The rules we have already given will then cause each link in the loop to be assigned the new color and the hop count U. When this thread either reaches its originator, or any other node which already has an incoming thread of the same color, it stalls.

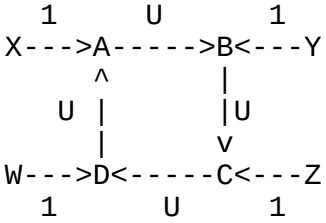
In our example above, this will cause the links AB, BC, CD, and DA to be given hop count U.

Now let's add one more rule:

A.12.2 When a thread with a known hop count reaches a node that has a colored outgoing U-thread, the incoming thread merges into the outgoing thread. (Actually, this is just a consequence of a rule which has already been given, since U is greater than any known hop count.)

Then if W, X, Y, or Z attempt to extend a thread to D, A, B, or C respectively, those threads will immediately stall. Once all the links are marked as being within a loop, no other threads are extended around the loop, i.e., no other setup messages will traverse the loop.

Here is our example topology with the link hop counts that would exist during a loop:



A.13. Some Special Rules for Hop Count U

When a U-thread encounters a thread with known hop count, the usual rules apply, remembering that U is larger than any known hop count value.

However, we need to add a couple of special rules for the case when a U-thread encounters a U-thread. Since we can't tell which of the two U-threads is really the longer, we need to make sure that each of the U-threads is extended.

A.13.1 If an incoming colored U-thread arrives at a node which already has an incoming U-thread of that color, or arrives at the node which created that U-thread, then the thread stalls.

(Once a loop is detected, there is no need to further extend the thread.)

A.13.2 If an incoming colored U-thread arrives at a node which has a transparent outgoing U-thread to its next hop, the incoming thread is extended.

A.13.3 If an incoming colored U-thread arrives at a node which has a colored outgoing U-thread, and if the incoming link over which the thread was received was already an incoming link of the LSP, the thread is extended.

A.13.4 If an incoming colored U-thread arrives at a node which has a colored outgoing U-thread, and if the incoming link over which the thread was received was NOT already an incoming link of the LSP, a new U-thread is created and extended. All the incoming threads are merged into it. This is known in the main body of this document as "extending the thread with changing color".

These rules ensure that an incoming U-thread is always extended (or merged into a new U-thread which then gets extended), unless it is already known to form a loop.

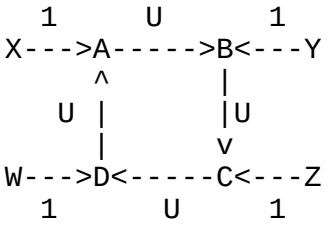
What is the purpose of rule A.13.4? There are certain cases where a loop can form, but where the node which created the looping thread is not part of the loop. Rule A.13.4 ensures that when there is a loop, there will be a looping thread which was created by some node which is actually in the loop. This in turn ensures that the loop will be detected well before the thread TTL expires.

The rule of "extending the thread with changing color" is also applied when extending a thread with a known hop count.

A.13.5 When a received colored thread with a known hop count is extended, if the node has an outgoing thread, and if the incoming link over which the thread was received was NOT already an incoming link of the LSP, a new thread is created and extended. All the incoming threads are merged into it. This is an exceptional case of A.5.1.

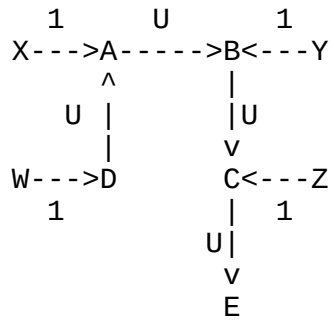
A.14. Recovering From a Loop

Here is our example topology again, in the presence of a loop.

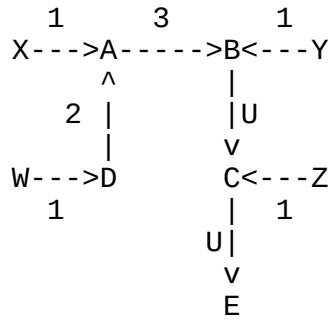


Suppose now that C's next hop changes from D to some other node E, thereby breaking the loop. For simplicity, we will assume that E is the egress node.

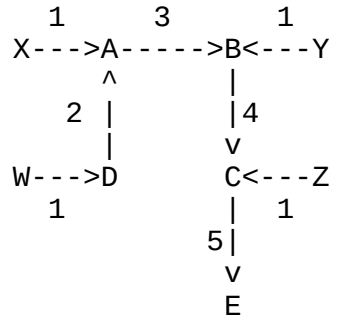
C will withdraw its outgoing U-thread from D (9.1). It will also create a new thread (12.1), assign it a new color, assign it hop count U (the largest hop count of C's incoming threads), merge its two other incoming threads into the new thread (12.2), and extend the new thread to E, resulting the following configuration:



When the thread from C to E rewinds, the merged threads also rewind (8.4). This process of rewinding can now proceed all the way back to the leafs. While this is happening, of course, D will note that its outgoing thread hop count should be 2, not U, and will make this change (9.3). As a result, A will note that its outgoing hop count should be 3, not U, and will make this change. So at some time in the future, we might see the following:



After a short period, we see the following:



with all threads transparent, and we have a fully set up non-looping path.

A.15. Continuing to Use an Old Path

Nothing in the above requires that any node withdraw a transparent thread. Existing transparent threads (established paths) can continue to be used, even while new paths are being set up.

If this is done, then some node may have both a transparent outgoing thread (previous path) and a colored outgoing thread (new path being set up). This would happen only if the downstream links for the two threads are different. When the colored outgoing thread rewinds (and becomes transparent), the previous path should be withdrawn.

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.