

# Improved Fault Recovery for Core Based Trees

Loren Schwiebert

Department of Electrical and Computer Engineering

Wayne State University

5050 Anthony Wayne Drive

Detroit, MI 48202-3902

Email: loren@ece.eng.wayne.edu

Phone: (313) 577-3990

Fax: (313) 577-1101

Raghunath Chintalapati

Softalia

2527 FarmCrest Dr, Apt #401

Herndon, VA 20171

Email: raghunath\_c@hotmail.com

## Abstract

The demand for multicast communication in wide-area networks, such as the internet, is increasing. Core based trees is one protocol that has been proposed to support scalable multicasting for sparse groups. When faults occur in the network nodes or links of the tree, the tree can become disconnected. In this paper, we propose an efficient protocol for recovering from faults in a core based tree. One of the key ideas is a technique for restructuring the disconnected subtree so that a loop-free path to the core can be found. The correctness of this protocol is also proved.

**Keywords:** multicast protocol, sparse-mode multicasting, core based tree, fault recovery.

## 1 Introduction

Multicasting is widely used for group communication. In a multicast, the sender transmits a single message that is replicated within the network and delivered to multiple recipients. For this reason, a multicast message typically requires less total bandwidth than a separate unicast message to each recipient. The sender and the receivers form the group sharing this information. Multicast protocols define how these groups are maintained and the route each message takes to reach all members of the group [8].

Multicast protocols currently in use, such as the Distance Vector Multicast Routing Protocol (DVMRP) [5, 9], which is used for the backbone of the Internet Multicast Backbone (MBONE), rely on flood and prune mechanisms to maintain multicast group membership. It is generally acknowledged that this approach generates too much network traffic for deployment in large networks when the set of participants is relatively sparse. As the demand for collaborative applications grows, an increase in multicast traffic is expected. More efficient use of the network bandwidth will be required to support these group applications.

Core based trees [1, 2, 3] have been proposed as a protocol for supporting sparse multicast groups over the internet. One of the objectives is to provide a multicasting protocol that scales to large networks. The protocol defines a central node, the *core*, which is the root of a tree that contains all group members.

When a node wishes to transmit a message to the multicast group, it sends the message toward the core. The message is distributed to the group members along the path to the core and once the message reaches the core, it is distributed to the remaining group members. Requests to join or leave the multicast group are processed by sending the request toward the core. When the message reaches the core, the node is added or removed and the tree is pruned of unused links if appropriate. If the message reaches an *on-tree router* along the path to the core, the request is instead processed at that router.

## 2 Problem Statement

For reliable multicasting, some mechanism is required to recover from faults. If one of the group members fails or one of the nodes or links used in the multicast tree fails, the multicast tree can become disconnected. It is then impossible for messages from the core to reach the members of the disconnected subtree.

The original specification of core based trees [4] included a mechanism for repairing the tree when a network node or link failed. The root of the disconnected subtree would send a REJOIN\_REQUEST toward the core using the appropriate unicast routing protocol. It is possible that routing around the fault sends the message through one of the children of this root. If the REJOIN\_REQUEST reached one of the children, a loop would be formed in the subtree when the child sent a REJOIN\_ACK in response to the REJOIN\_REQUEST. In order to avoid this situation, this root would flush the subtree (using FLUSH\_TREE) if it detected that the new path went through one of its children. However, it is also possible that the REJOIN\_REQUEST is routed through one of the descendants of the root, other than its child. In this case, the REJOIN\_ACK is ignored and the REJOIN\_REQUEST is retried at a later time.

Subsequently, Shields and Garcia-Luna-Aceves [10] showed that the original specification could result in the creation of incorrect multicast trees. Loops could be created. Also, the subtree might remain permanently disconnected, as the root of the disconnected subtree continually retried the same path that routes through one of its descendants.

Figure 1 depicts an example of a loop forming in a subtree when attempting to reconnect the subtree with the core. The edges currently used in the tree have arrowheads. The core and the root of a disconnected subtree are shown. There is a link failure immediately preceding the node labeled root. The root would like to reconnect to the core using the path indicated by the dashed

line. Unfortunately, this path routes through one of the children of the root. When this child receives the REJOIN\_REQUEST, it forwards the request up the current tree to root. Thus, a loop forms and root is unable to reconnect with the core. This loop is shown with the thick line from root back to itself.

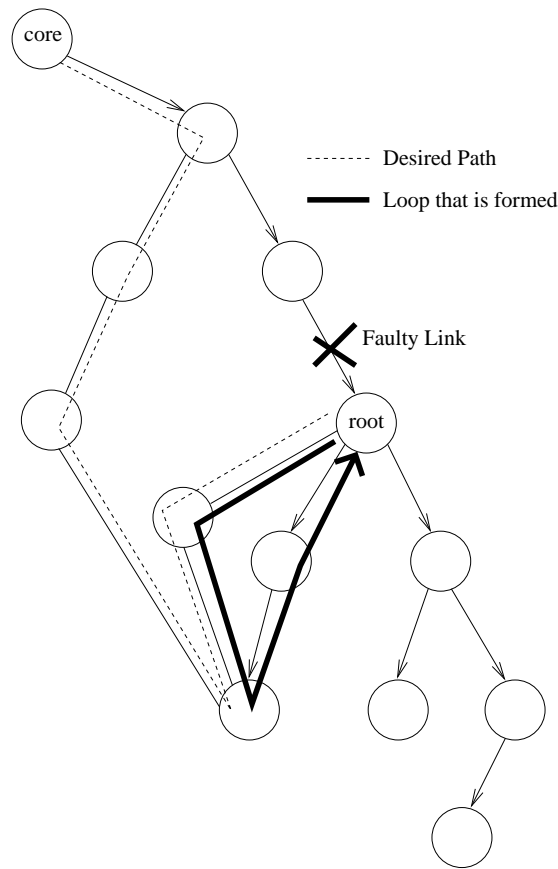


Figure 1: Loop in a Rejoining Subtree

In order to resolve the problem of loops forming after faults, the protocol specification of core based trees [2, 3] was modified to eliminate the possibility of generating loops when faults are detected. Rather than trying to reconnect the subtree, the subtree is flushed and all group members in the subtree attempt to rejoin the tree individually. This eliminates the problem of loop formation when rejoining the tree, however, there are three drawbacks to this approach. The first is that there could be a substantial delay in rebuilding the tree, as distant members of the subtree first receive a FLUSH\_TREE message and must then initiate a JOIN\_REQUEST with the associated delay. During the rebuilding phase, no messages are received from the core. The second disadvantage is that a subtree with many members could experience a substantial increase in network traffic as the control messages are propagated through the network. Finally, overhead at the on-tree routers may be high when processing many simultaneous requests to join the group.

A better protocol for resolving faults would be beneficial. Although faults may seem to be uncommon events, the chance of faults is higher than one might expect. For example, it was

recently observed [7] that the internet occasionally experiences periods of routing instability, also known as routing flaps, when the network can temporarily lose connectivity as floods of routing updates are processed. This network instability could lead to timeouts that result in the flushing of subtrees due to these transient faults. The added overhead of rebuilding the core based tree will generate additional load on an already overloaded router. In addition, networked mobile hosts are becoming increasingly common and including these hosts in multicast groups is desirable. However, mobile hosts are likely to change connectivity on a more frequent basis. If there are delays in hand-offs, it is possible that a faulty node or link will be assumed. Finally, the use of core based trees in multi-hop ad hoc mobile networks has been proposed [6]. As mobile hosts move in this environment, the connectivity of the network is likely to change and the structure of the tree may break. (A similar situation could arise in a mobile network with mobile base stations.) With the current core based tree specification [2, 3], frequent flushing and rebuilding of the tree may be required.

In this paper, we propose an improved protocol for fault recovery in core based trees that guarantees correct multicast trees. A formal description of the additional message types and processing required by our protocol is presented in appendix A. These trees are free from loops and in almost all cases require fewer control messages to reconnect the disconnected group members than are required by flushing and rebuilding the subtree.

### 3 New Protocol

In the original specification of core based trees [4], the REJOIN\_REQUEST message was used to reconnect disconnected trees. This message would succeed unless the request was routed through one of the descendants of the root of the disconnected subtree (or the network was disconnected). If the message routes through one of the descendants, a loop is formed and the tree is not reconnected. The problem of loop formation was first pointed out by Shields and Garcia-Luna-Aceves [10]. In order to resolve this problem, the specification of core based trees was modified. As described above, the REJOIN\_REQUEST message is no longer used to reconnect the disconnected subtree to the core. Instead, the subtree is flushed and each multicast group member in the subtree rejoins separately. This approach avoids the formation of loops, but in this paper we propose a better way of resolving this problem. Our approach rarely flushes the subtree and is more efficient in almost all cases.

Note that a request from a node that routes through one of its descendants is readily detected, because once the request reaches an on-tree router, the request is forwarded up the tree toward the core. So, the root of the subtree will eventually see its own REJOIN\_REQUEST and thus detect the loop. On the other hand, if the REJOIN\_REQUEST finds a valid path, a REJOIN\_ACK is returned to the root of the subtree by the on-tree router. The subtree is thus reconnected and communication can proceed.

In order to simplify our protocol and guarantee correctness, a slight modification is made to the REJOIN\_REQUEST in the original protocol. Instead of having the on-tree router send the REJOIN\_ACK, the REJOIN\_ACK is sent by the core of the core based tree. Note that since the core already received every REJOIN\_REQUEST in the original specification, the additional overhead is minimal. This implementation guarantees that no router in the tree receives a REJOIN\_ACK unless

the REJOIN\_REQUEST is successful.

Since the failure of the original protocol to work correctly is easily detected, it makes sense to use this approach if the REJOIN\_REQUEST succeeds most of the time. Although it is difficult to precisely quantify the success rate of the REJOIN\_REQUEST, Shields and Garcia-Luna-Aceves [10] obtained a failure rate of only 11.3% in their simulations. This translates into a success rate of nearly 90%. Intuitively, this seems reasonable. Since the route from the root of the disconnected subtree to the core did not originally go through one of the descendants, it seems likely that there is a low probability that the new route will go through one of the descendants.

In section 4, a protocol will be presented that can reconnect the subtree even if the path is through one of the descendants of the subtree. In section 5, unreliable message delivery is addressed.

## 4 Enhanced Protocol

In this section, an enhanced version of our protocol is presented that can resolve most of the REJOIN\_REQUESTS that fail because the route is through one of the descendants. This protocol also allows the disconnected subtree to be reconnected even if the path from the root is through one of its children. The change is relatively straightforward and the overhead in terms of additional control message traffic should be minimal. The modified protocol works as follows:

### 4.1 Operation

When the first on-tree router, denoted  $OTR_1$ , receives the REJOIN\_REQUEST, it embeds its IP address (or other suitable unique identifier) in the packet and forwards the request toward the core. If the REJOIN\_REQUEST reaches the core without routing through the root of the disconnected subtree, denoted  $R_s$ , then proceed as in section 3.

On the other hand, if the REJOIN\_REQUEST returns to  $R_s$ , then a potential loop has been formed. In this case, corrective action is required to reconnect the subtree. *The key idea is that by reversing the tree edges from  $R_s$  to  $OTR_1$ ,  $OTR_1$  becomes the root of the disconnected subtree and can then attempt to reconnect to the core based tree.* An illustration of this property can be seen in figures 2 and 3.

In order to reverse the edges of the subtree,  $R_s$  sends a LOOP\_FORMED unicast message to  $OTR_1$ . This message may follow the path within the multicast tree or the path that was previously taken from  $R_s$  when attempting to reach the core. When  $OTR_1$  receives the LOOP\_FORMED message, it sends a REVERSE\_EDGES message up the tree to  $R_s$ . The REVERSE\_EDGES message causes the routers along the path to swap the upstream and downstream links. More precisely, the upstream link is swapped with the downstream link on which the REVERSE\_EDGES message was received. (As described later, the reversal is postponed until a REVERSE\_ACK is received.) If a router along the path is unable to reverse the edges for some reason, it should discard the REVERSE\_EDGES message and send a REVERSE\_NACK to  $OTR_1$  and  $R_s$ .  $R_s$  should flush the subtree (using FLUSH\_TREE) when it receives the REVERSE\_NACK. The forwarding process is terminated when the REVERSE\_EDGES message reaches  $R_s$ .  $R_s$  then sends a REVERSE\_ACK unicast message toward  $OTR_1$  and reverses its upstream and downstream links. The REVERSE\_ACK

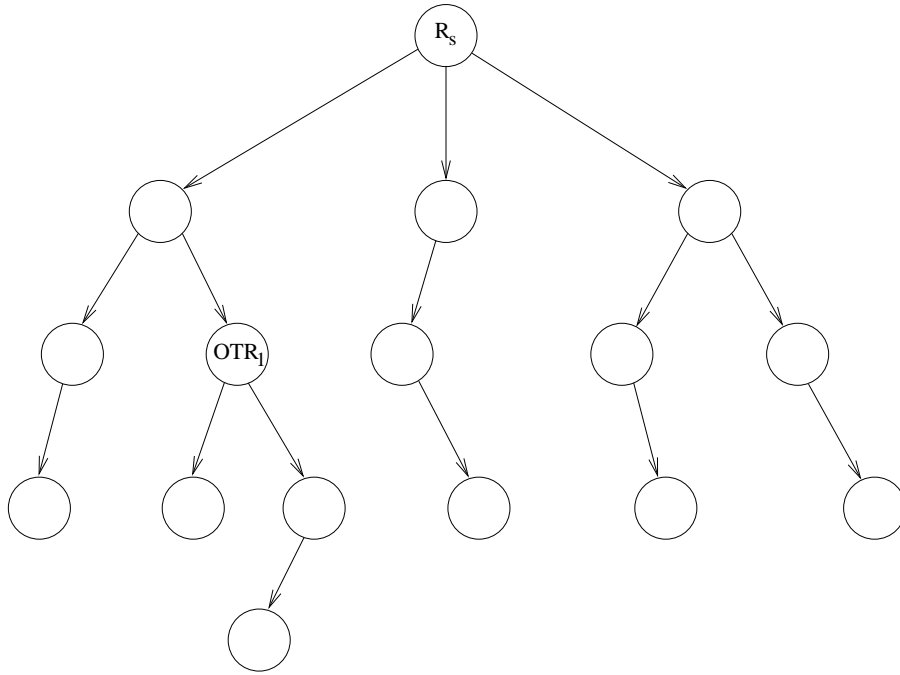


Figure 2: Original Subtree

follows the tree edges to  $OTR_1$  and each router along the path forwards the message and reverses its edges. When  $OTR_1$  receives the `REVERSE_ACK`, it reverses the edges as well and becomes the root of the disconnected subtree. A new `REJOIN_REQUEST` is initiated by  $OTR_1$  and the protocol repeats the above steps.

As an optimization, if the path from the root to the core begins with a child of the root, these two routers can immediately exchange `LOOP_FORMED` and `REVERSE_EDGES` messages. The child then becomes the root of the subtree and initiates a `REJOIN_REQUEST`.

## 4.2 Analysis

The advantage of restructuring the subtree and retrying the `REJOIN_REQUEST` depends on the success rate. The success rate is difficult, if not impossible, to precisely quantify. However, both simulation results and intuition suggest this number should be reasonably high. For example, Shields and Garcia-Luna-Aceves [10] obtained a failure rate of only 11.3% in their simulations, so 88.7% of the `REJOIN_REQUEST` attempts were successful. Their simulations proceeded by causing a single link failure in the sample network and determining if the original core based tree protocol was able to correctly reconnect the tree [10]. The failure rates were determined by running multiple simulations, one for each link in the network, and determining the percentage of link failures that were correctly resolved.

Assuming a 90% rate of success for a `REJOIN_REQUEST`, the likelihood that this will succeed in two attempts is 99%. Ideally, the process can be repeated several times to increase the probability of success. If each reconfiguration of the subtree ends with a new root closer to the core than the

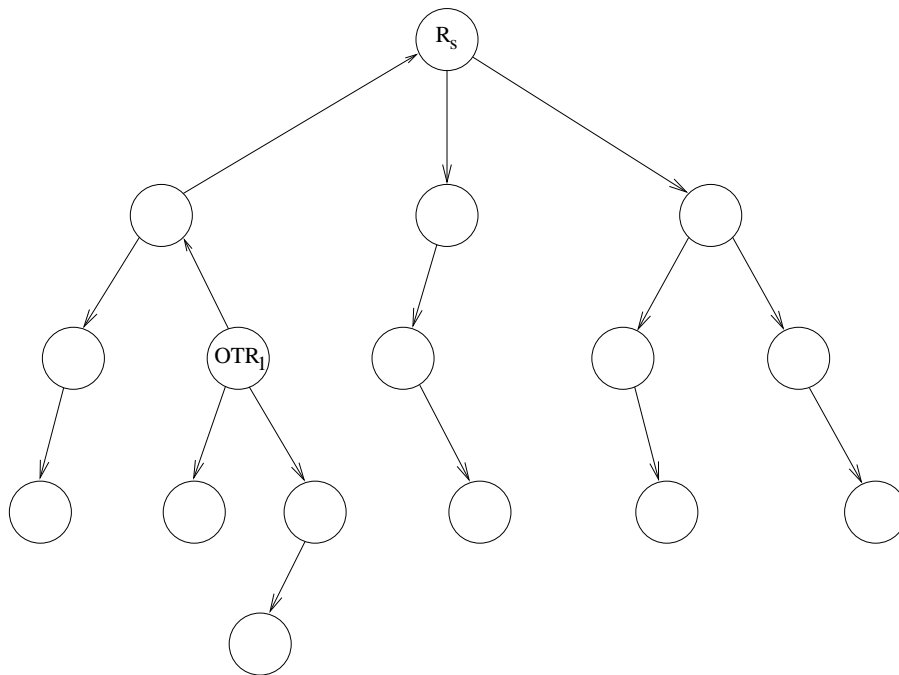


Figure 3: Subtree with Reversed Edges

previous root of the subtree, and the network is connected, the process will eventually terminate with a reconnected tree. However, since the number of trees successfully reconnected diminishes after each attempt and routing instability may render the assumption of moving closer to the core incorrect, it may be better to flush the subtree after a fixed number of reconfigurations. Given a 90% chance of success, three attempts yields a success rate of 99.9%, so three attempts seems a reasonable number. Similarly, an 80% chance of success implies that three attempts yields a success rate of 99.2%. Even a mere 60% chance of success means a 93.6% chance of success after three attempts. In this case, extending the protocol to four attempts improves the success rate by only 3.8% to 97.4%.

### 4.3 Uni-directional Trees

By default, a JOIN\_REQUEST creates bi-directional state information, which makes swapping the upstream and downstream links trivial. It is also possible to construct a uni-directional path. In this case, data cannot flow up the tree. However, there is no prohibition on sending *control* information up the tree. Note that a bi-directional flow of control information is required when nodes join the core based tree. Thus, the upstream and downstream information can be swapped correctly even in the uni-directional case.

Although it is technically feasible to apply this technique even in the case of uni-directional trees, it is not completely clear whether or not this is advisable. If these paths are uni-directional simply because these members will not be senders of multicast messages, then reversing the edges maintains this status and there are no negative side effects. On the other hand, if reversing the flow

of information on a link is not possible for some router, a REVERSE\_NACK can be transmitted to both  $OTR_1$  and  $R_s$ . The root of the subtree should then flush the subtree and the group members can reconnect individually as in the current protocol specification [2, 3].

## 5 Unreliable Message Transmission

Using this protocol, the root of the subtree receives its own REJOIN\_REQUEST if a loop has been formed in the route and receives a REJOIN\_ACK if the reconnection request was successful. A timeout can be used to detect the loss of a message and a REJOIN\_REQUEST can be retransmitted a finite number of times in an attempt to recover the lost message. If an on-tree router receives a REJOIN\_REQUEST and it has already forwarded a prior REJOIN\_ACK from the core for that request, it can respond with a REJOIN\_ACK directly and not forward the REJOIN\_REQUEST toward the core.

Similarly, when reversing the edges using the LOOP\_FORMED and REVERSE\_EDGES messages, along with a REVERSE\_ACK or REVERSE\_NACK response as appropriate, a message could be lost. By using timeouts and retrying a finite number of times and then flushing the subtree after these retries have all timed out, the protocol will eventually terminate. In other words,  $R_s$  should resend the LOOP\_FORMED message if it has not received a REVERSE\_EDGES or REVERSE\_NACK before some timeout. Similarly,  $OTR_1$  should resend the REVERSE\_EDGES message if it has not received a REVERSE\_ACK or REVERSE\_NACK before some timeout. For REVERSE\_ACK and REVERSE\_NACK messages, resending a few times is also a reasonable option.

Since the actual reversal of links is postponed until a REVERSE\_ACK is received, the subtree can have an inconsistent state only while the REVERSE\_ACK is progressing from  $R_s$  to  $OTR_1$ . Having each router that received a REVERSE\_ACK forward it a few times may help avoid this problem. At the point where the message was lost, adjacent routers on the path will have one router with a reversed edge and one without.

If a REVERSE\_ACK is lost,  $OTR_1$  will resend the REVERSE\_EDGES message. The first router on the path that has successfully processed the REVERSE\_ACK message can immediately respond with a REVERSE\_ACK and not forward the message up the tree. After a few attempts, the REVERSE\_ACK should reach  $OTR_1$ . If so,  $OTR_1$  becomes the root of the subtree.

If the REVERSE\_ACK never succeeds, then there is a faulty link or node in the subtree. In this case the ECHO protocol should also detect the fault. (This can be guaranteed by piggy-backing the REVERSE\_ACK and an acknowledgment of the receipt of the REVERSE\_ACK on the ECHO\_REQUEST and ECHO\_REPLY messages, respectively.) In the case of a link failure, the original subtree splits into two separate subtrees. The adjacent routers, where one performed the REVERSE\_ACK and one did not, become the root of their respective subtrees. In the case of an on-tree router failure, the neighboring router that performed the REVERSE\_ACK and the children of the router that failed become the roots of distinct subtrees.

## 6 Correctness

In this section, it is proved that, eventually, the protocol terminates and the core based tree is in a correct state. A correct state means that there are no loops in the tree. An invocation of the

protocol terminates when the subtree is reconnected to the core based tree, a fault is detected and the subtree is divided, or the protocol is unsuccessful and the subtree is flushed. Given that the correctness of the current core based tree specification [2, 3] has been shown and the disconnected subtree is always flushed in that specification, the cases when the protocol resorts to flushing the tree obviously result in correct trees. First consider the case where no message loss occurs.

**Theorem 1** *Given reliable message delivery, the protocol terminates with a correct tree.*

**Proof.** A finite amount of time after the root of the subtree transmits a REJOIN\_REQUEST, it receives either a REJOIN\_ACK or a copy of its own REJOIN\_REQUEST. Since a REJOIN\_ACK is sent only after the REJOIN\_REQUEST reaches the core, receipt of a REJOIN\_ACK means that a path to the core has been found that does not route through this subtree. Therefore, the tree is obviously correct.

If the REJOIN\_REQUEST returns to the root, the path to the core lies through one of its descendants. The LOOP\_FORMED and REVERSE\_EDGES messages, following by a REVERSE\_ACK message, successfully restructure the subtree into a new valid tree. The REJOIN\_REQUEST process is then repeated. Since it is repeated only a few times before giving up and then flushing the tree, the protocol terminates in either case. A REVERSE\_ACK is not received by the root if one of the routers along the path cannot reverse its edges. In this case, a REVERSE\_NACK is received and the subtree is flushed. Thus, in all cases, the protocol terminates with a loop-free tree.  $\square$

**Theorem 2** *The protocol remains correct, even if messages are lost.*

**Proof.** The proof proceeds by examining the loss of messages at each stage of the protocol and showing that the loss of one or more messages does not lead to invalid trees.

The protocol starts with a REJOIN\_REQUEST from the root of the disconnected subtree. If this message is lost, the root will retry the REJOIN\_REQUEST after a timeout. After a few retries, the root will flush the subtree.

If the REJOIN\_REQUEST reaches the core, but the REJOIN\_ACK is lost, the root of the subtree will timeout and generate a new REJOIN\_REQUEST. This REJOIN\_REQUEST eventually reaches the core and a new REJOIN\_ACK is transmitted. As an optimization, the REJOIN\_ACK could be retransmitted by an on-tree router that has previously forwarded the REJOIN\_ACK, however, this does not affect the correctness. If the REJOIN\_ACK does not reach the root of the subtree after a few attempts, the subtree is flushed.

If the root receives the REJOIN\_ACK, the protocol terminates. If the root receives its own REJOIN\_REQUEST, then it sends a LOOP\_FORMED message to the on-tree router who first received the REJOIN\_REQUEST. If the LOOP\_FORMED message is lost, the root will not receive the corresponding REVERSE\_EDGES message (or a REVERSE\_NACK). After some period of time, the LOOP\_FORMED message will be retransmitted. After a few attempts, the subtree is flushed. Since no edges are reversed until the root receives a REVERSE\_ACK, the subtree can be correctly flushed.

If the LOOP\_FORMED message is received by the appropriate on-tree router, it initiates a REVERSE\_EDGES message. If the REVERSE\_EDGES message is lost, then no corresponding REVERSE\_ACK or REVERSE\_NACK is received. The on-tree router will then timeout and resend the REVERSE\_EDGES message. If the REVERSE\_EDGES message is lost several times, the subtree is

eventually flushed. Recall that after some period of time, the root of the subtree will flush the subtree if it does not receive a REVERSE\_EDGES or REVERSE\_NACK message, so no further action is required by the on-tree router.

If the REVERSE\_NACK message is lost and does not reach the on-tree router, the on-tree router resends the REVERSE\_EDGES message after a timeout. If the REVERSE\_NACK reaches the root, the subtree is flushed. If the REVERSE\_NACK reaches the on-tree router but not the root, the root will resend the LOOP\_FORMED message after a timeout. If the on-tree router resends the REVERSE\_EDGES message, the REVERSE\_NACK may reach the root. Alternatively, the on-tree router could forward the REVERSE\_NACK to the root or ignore the LOOP\_FORMED message. In either case, the tree is eventually flushed.

If the REVERSE\_ACK message is lost, the on-tree router will resend the REVERSE\_EDGES message. The last on-tree router to process the REVERSE\_ACK is currently the root of the subtree and should respond accordingly. Thus, if the REVERSE\_EDGES message reaches this router, it should immediately respond with another REVERSE\_ACK. If the REVERSE\_ACK never succeeds, this is recognized as well when the ECHO protocol between these two on-tree routers fails. A fault is thus detected and the subtree is further divided into multiple disconnected subtrees.

In all cases, a correct state is reached even after the loss of one or more messages. Thus, the protocol correctly recovers from lost messages.  $\square$

The two preceding proofs show that the protocol terminates correctly with a valid tree in a finite amount of time. This demonstrates that the protocol works correctly for a single disconnected subtree. In theorem 3, a proof is given that the protocol also works correctly when invoked concurrently by multiple disconnected subtrees.

**Theorem 3** *Concurrent execution of this protocol by multiple distinct subtrees does not lead to incorrect trees.*

**Proof.** If multiple subtrees invoke the protocol concurrently, no problems result if the subtrees do not interact. In other words, unless a message travels from one disconnected subtree to the other, there is no potential for interference. Note also that the only messages that must be routed *outside* of the subtree in order to reconnect to the core are the REJOIN\_REQUEST and the REJOIN\_ACK. However, the REJOIN\_REQUEST times out if a REJOIN\_ACK is not received and by definition, a disconnected subtree cannot forward the REJOIN\_REQUEST to the core. Thus, interactions between disconnected subtrees do not lead to the formation of invalid trees.  $\square$

## 7 Conclusions

In this paper, an improved fault recovery protocol has been proposed. The current core based tree specification [3] flushes an entire subtree whenever a network failure causes the multicast tree to become disconnected. This introduces additional network traffic and could lead to an extended reconnection time period. Our protocol eliminates this overhead and delay in most cases.

If each attempt offers even a modest chance of reconnecting a disconnected subtree to the original core based tree, then our protocol has a success rate exceeding 90%. The algorithm is easily tuned to an arbitrary probability of reconnecting, although the delay increases.

The correctness of the algorithm has been shown. The protocol either successfully reconnects the disconnected subtree, splits the tree if additional faults occur, or times out and flushes the subtree. If the subtree is flushed, each group member must reconnect to the existing tree individually. However, in most cases, flushing the tree should be unnecessary and the protocol should recover from faults quickly.

One natural extension of this work would be a careful simulation study to illustrate the advantages of this protocol under realistic situations. In order to do this, however, simulated networks would have to be designed that include realistic multicast groups in terms of size and distribution throughout the network. Realistic distributions of faults and failure rates are also required for such a study.

## Acknowledgments

The authors thank Golden Richard for a careful reading of a draft of this paper and numerous helpful comments on improving the presentation. The authors also thank the anonymous reviewers for several helpful suggestions.

## A Formal Description of the Protocol

The formal description is provided by first giving a detailed explanation of each new message type and its processing. A pseudo-code description of the operation of the protocol is also presented. Six new message types are introduced by this new protocol. These messages must be formatted consistently with the existing core based tree protocol messages. Details on the formatting of the messages (types of fields and their location in the header) can be found in [3]. Required fields such as message type are assumed and not repeated in the discussion below. Similarly, the source and destination addresses for each message are assumed to be present in the message header. Any of these messages can be repeated to improve reliability.

1. **REJOIN\_REQUEST** – A **REJOIN\_REQUEST** is sent from the root of the disconnected subtree toward the core. The **REJOIN\_REQUEST** contains the multicast group address, the address of the core, and the address of the root of this subtree. The first on-tree router to receive this message adds its address to the message. This message also supports two options: (1) a message ID for unique identification and (2) a retry counter to guarantee termination.
2. **REJOIN\_ACK** – A **REJOIN\_ACK** is transmitted from the core to the sender (root of the disconnected subtree). The **REJOIN\_REQUEST** succeeded and the tree has been reconnected. The **REJOIN\_ACK** header contains the unique message ID that was included with the **REJOIN\_REQUEST** message.
3. **LOOP\_FORMED** – If a **REJOIN\_REQUEST** returns to the root of the disconnected subtree, the root knows that the path to the core is through one of its children. The **LOOP\_FORMED** message is unicast to the first on-tree router the **REJOIN\_REQUEST** reached. This message informs that on-tree router that it should become the new root of this subtree.

4. REVERSE\_EDGES – When the on-tree router receives the LOOP\_FORMED message, it sends the REVERSE\_EDGES message up the tree toward the core. The routers on this path set transient state information so that the correct edges can be reversed when an acknowledgment is received. When this message reaches the root of the subtree, the acknowledgment is sent.
5. REVERSE\_ACK – The REVERSE\_ACK is sent from the root of the disconnected subtree to the new root of this subtree. The new root can then try to connect to the core. The REVERSE\_ACK travels along the same path as the REVERSE\_EDGES but in the opposite direction, using the transient information created for the REVERSE\_EDGES message. The edges of the multicast tree are reversed as this message travels toward the new root. When the REVERSE\_ACK message reaches the new root, the transfer is complete; the subtree has been successfully restructured.
6. REVERSE\_NACK – If an edge cannot be reversed for some reason, the node responds to the REVERSE\_EDGES message with a REVERSE\_NACK. This message is sent to both the root of the subtree and the sender of the REVERSE\_EDGES message. When the root receives this message, it flushes the subtree. (Sending the message to the source of the REVERSE\_EDGES message allows it to become aware that the tree will be flushed, so it should not resend the REVERSE\_EDGES message.)

Next, pseudo-code is presented to show how each of the routers in the network processes the messages in the new protocol. Only the pseudo-code needed for this protocol is presented and code for the generic functioning of the core based trees protocol is omitted. The core simply responds to a REJOIN\_REQUEST with a REJOIN\_ACK, and routers that are not part of the multicast tree simply forward messages, so no pseudo-code is shown for these routers. The root of the disconnected subtree is presented first:

```

subtree root()
  Increment msgID;
  Set retry-counter to 0;
1: Send REJOIN_REQUEST toward the core;
  Wait for (REJOIN_REQUEST or REJOIN_ACK)
    if (REJOIN_ACK) //successfully reconnected!
      Terminate REJOIN process;
    else if (REJOIN_REQUEST)
      Extract first on-tree router address from message;
      Send LOOP_FORMED message to first on-tree router;
      Goto 2:
    endif
  endwait
  if (timed-out)
    Increment retry-counter;
    if (retry-counter > max)
      Send FLUSH_TREE message to subtree;
      Terminate REJOIN process;

```

```

        else
            Goto 1:
        endif
    endif
endif
2: Wait for (REVERSE_EDGES or REVERSE_NACK)
    if (REVERSE_EDGES)
        Mark incoming channel;
        Send REVERSE_ACK on this channel;
        Reverse Direction of incoming channel;
        Terminate REJOIN process;
    else if (REVERSE_NACK)
        Send FLUSH_TREE message to subtree;
        Terminate REJOIN process;
    endif
endwait
endprocess

```

The processing by other on-tree routers is partitioned for clarity. The first on-tree router is considered separately from the other on-tree routers.

```

first-on-tree-router()
Upon receipt of a REJOIN_REQUEST:
if (duplicate request and REJOIN_ACK already received)
    Resend REJOIN_ACK;
else
    Put ID in REJOIN_REQUEST message;
    Forward message up the multicast tree;
    Wait for (REJOIN_ACK or LOOP_FORMED)
        if (REJOIN_ACK) // Reached the core, no loop!
            Send REJOIN_ACK down the multicast tree;
            Terminate REJOIN processing;
        else if (LOOP_FORMED)
            Mark appropriate edge;
            Set retry-counter to 0;
            Goto 1:
        endif
    endwait
1: Send REVERSE_EDGES up the tree;
    Wait for (REVERSE_ACK or REVERSE_NACK)
        if (REVERSE_ACK)
            Reverse edge;
            Execute subtree root code;
        else if (REVERSE_NACK)
            Terminate REJOIN processing;
        endif

```

```

endwait
if (timed-out)
    Increment retry-counter;
    if (retry-counter > max)
        // Current root will flush the subtree after a timeout
        Terminate REJOIN process;
    else
        Goto 1:
    endif
endif
endif
endprocess

other-on-tree-router()
Upon receipt of a REJOIN_REQUEST:
if (duplicate request and REJOIN_ACK already received)
    Resend REJOIN_ACK;
else
    Forward message up the multicast tree;
    Wait for (REJOIN_ACK or REVERSE_EDGES)
    if (REJOIN_ACK)
        Send REJOIN_ACK down the multicast tree;
        Terminate REJOIN processing;
    else if (REVERSE_EDGES)
        Mark appropriate edge;
        Send REVERSE_EDGES up the tree;
    endif
endwait
1: Wait for (REVERSE_ACK or REVERSE_NACK or REVERSE_EDGES)
    if (REVERSE_ACK)
        Forward REVERSE_ACK along the appropriate tree edge;
        Reverse edge;
    else if (REVERSE_NACK)
        Terminate REJOIN processing;
    else if (REVERSE_EDGES)
        if (already sent a REVERSE_ACK)
            Resend REVERSE_ACK along the appropriate tree edge;
        else
            Send REVERSE_EDGES up the tree;
            Goto 1:
        endif
    endif
endwait
endif
endprocess

```

## References

- [1] A. Ballardie. Core Based Trees (CBT) Multicast Routing Architecture. RFC 2201, Internet Engineering Task Force, September 1997.
- [2] A. Ballardie. Core Based Trees (CBT version 2) Multicast Routing Protocol Specification. RFC 2189, Internet Engineering Task Force, September 1997.
- [3] A. Ballardie, B. Cain, and Z. Zhang. Core Based Trees (CBT version 3) Multicast Routing Protocol Specification. Internet Draft draft-ietf-idmr-cbt-spec-v3-01, Internet Engineering Task Force, August 1998.
- [4] A. Ballardie, P. Francis, and J. Crowcroft. Core based trees (CBT). In *ACM SIGCOMM*, pages 85–95, 1993.
- [5] S. Deering, C. Partridge, and D. Waitzman. Distance Vector Multicast Routing Protocol. RFC 1075, Internet Engineering Task Force, November 1988.
- [6] S. K. S. Gupta and P. K. Srimani. An Adaptive Protocol for Reliable Multicast in Mobile Multi-Hop Radio Networks. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 111–122, 1999.
- [7] C. Labovitz, R. Malan, and F. Jahanian. Internet Routing Instability. *IEEE/ACM Transactions on Networking*, 6(5):515–528, October 1998.
- [8] J. Lin and R.-S. Chang. A Comparison of the Internet Multicast Routing Protocols. *Computer Communications*, 22(2):144–155, January 1999.
- [9] T. Pusateri. Distance Vector Multicast Routing Protocol. Internet Draft draft-ietf-dvmrp-v3-08, Internet Engineering Task Force, February 1999.
- [10] C. Shields and J. J. Garcia-Luna-Aceves. The Ordered Core Based Tree Protocol. In *INFOCOM '97*, 1997.