Dynamic Multicast Trees

Jorge A. Cobb

Department of Computer Science (EC 31)
The University of Texas at Dallas
Richardson, TX 75083-0688
jcobb@utdallas.edu

Abstract

We present a family of protocols to build a multicast tree in a network of processes. No processing or storage overhead is required for processes not included in the tree. The overhead of processes in the tree consists solely of the periodic exchange of a pair of messages with their parent in the tree. To choose the processes that constitute the tree, we take advantage of the existing unicast routing tables. In addition, our protocol family distinguishes itself from other protocols in three ways. First, the protocols are proven correct. Second, the integrity of the multicast tree is preserved as the tree adapts to changes in the unicast routing table. Third, the protocols are self-stabilizing, i.e., they tolerate all transient faults.

1. Introduction

We present a family of protocols to build a multicast tree in a network of processes. The network consists of a set of processes interconnected by point-to-point communication channels. Multicast routing has many applications, such as audio and video conferencing [16], replicated database updating and querying, and resource discovery [13].

In multicast routing, a tree is constructed which spans all the nodes of a multicast group. Each node in the tree corresponds to a process in the network, and each edge in the tree corresponds to a communication link between two processes. For simplicity, we present a protocol for a single process group. The extension to multiple groups is straightforward.

To build a multicast tree, we take advantage of the existing unicast routing tables of each process, and use them as a guide in the construction of an efficient tree. The unicast routing tables define a forest of spanning trees, one tree for each process in the network. The multicast tree is constructed as a subgraph of one of these spanning trees.

The maintenance and construction of the multicast tree requires minimal overhead from the network. Processes involved in the multicast tree simply periodically exchange a pair of messages with their parent in the multicast tree, and store the process id's of their parent and children. Processes not involved in the group tree incur no message or storage overhead.

Many unicast routing algorithms exist in the literature, e.g., [1], [2], [11], [12], [15]. These algorithms have many differences. However, common to all is the ability to change the routing tables in response to varying network conditions, such as fluctuations in traffic loads. To make our protocols suitable for operating in conjunction with any unicast routing protocol, we only make the following simple assumption about unicast routing. The unicast routing tables may temporarily fluctuate, but they eventually converge to a value that, for each pair of processes p and q, defines a path from p to q.

To maintain the routing efficiency of the multicast tree, when the unicast routing tables change, the tree is restructured to reflect these changes. For example, consider a multicast distribution of a video image. If the topology of the network changes, and a network path with greater bandwidth is created between the source of the video and its destinations, then the multicast tree should converge and use this new path.

In addition, our protocols have the desirable property of maintaining the integrity of the multicast tree while the unicast routing tables change. That is, no temporary loops are introduced, the tree is always connected, and no member of the multicast group is temporary removed from the tree. In the above example, the multicast tree would adapt itself to the best network path without interrupting the flow of video to the members of the multicast group.

Building multicast trees based on the unicast routing tables is an existing technique. In [3] [4], a tree is initially built from the unicast routing tables. However, the tree is static, and does not adapt to changes in these tables. Obtaining a broadcast tree from the unicast routing tables was first presented in [8]. In [6] [7], the broadcast tree is trimmed into a multicast tree that excludes those processes not needed to reach the members of the group. However, as the unicast routing tables change, the tree may lose its integrity and become disconnected, until the unicast routing tables converge to a stable value.

In [5], we presented a multicast tree protocol that adapts the tree to changes in the unicast routing tables, without compromising the integrity of the multicast tree. In this paper, we present a protocol, which, as the protocol in [5], maintains the integrity of the multicast tree. However, it has the following powerful additional features.

First, the root of the tree is not fixed. Previous protocols require the root of the tree to be a constant. However, the location of the root is crucial for the efficiency of the protocol, and the best choice for the root node may vary over time. In our protocol, the root of the tree is allowed to change, and during the change the integrity of the tree is maintained. Second, our protocol is self-stabilizing [9], i.e., if started from any arbitrary initial state, it converges to a normal operating state. This makes the protocol very robust against transient failures, such as link failures and recoveries, and the reception of corrupted messages that passed their CRC check. These failures may cause the system to be thrown into an arbitrary state. Nonetheless, the protocol will converge to a good operating state.

We present our family of multicast routing protocols in three steps. First, we present a basic version of the protocol, which maintains a multicast tree and preserves its integrity. Then, we enhance the protocol by allowing the root of the three to be dynamically chosen. Finally, we present the self-stabilizing version of the protocol.

Due to space restrictions, the correctness proofs of these protocols may be found in [17].

2. A Loopless Multicast Protocol

In this paper, we present a family of multicast tree protocols. Each protocol consists of a set of processes which exchange messages via communication channels. The processes and their channels form a network that may be represented as an undirected graph. In this graph, each node[1] represents a process, and an edge between processes p and q represents two first-in-first-out communication channels, one from p to q and another from q to p.

In this section, we present a multicast tree protocol, similar to the one we presented in [5]. It is based on a fixed root, it adapts to changes in the unicast routing tables, and preserves the integrity of the multicast tree.

We present the protocol in three steps. We first present the technique to build a multicast tree from the existing unicast routing tables. We then show how to ensure each node always has a parent in the tree. Finally, we show how to eliminate temporary loops in the multicast "tree", therefore ensuring we have a well-defined tree at all times.

3.1 Using The Unicast Routing Tables

Consider the network in Figure 1, where edges represent bi-directional channels between processes. Consider process s. The arrows correspond to the next-hop neighbor to reach process s according to the unicast

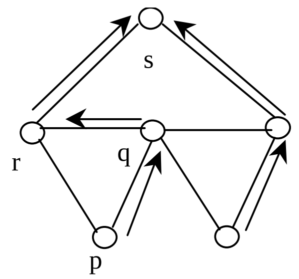

Figure 1

routing tables of each process. That is, the next-hop neighbor of p to reach s is q, the next-hop neighbor of q to reach s is r, and the next-hop neighbor of r to reach s is s itself. Note that the edges denoted by the arrows form a spanning tree.

We take advantage of the spanning tree defined above to build a multicast tree. To do so, we designate a fixed process in the network as the root (e.g., s = root above). The parent of each process p in the multicast tree is the next-hop neighbor in the unicast path from p to the designated root process. Thus, the multicast tree is a subset of the unicast spanning tree whose root is also the designated root process.

Note that the multicast tree must contain all members of the process group, plus any additional processes required to complete the tree. Process p chooses to join the tree if p is a group member, or if it has a neighbor which has chosen p as its parent. If neither of these is true, p removes itself from the multicast tree.

To build and maintain the multicast tree, each process p requires the following variables: p.par, p.chil, and p.next. Variable p.par stores the process identifier of p's parent in the tree. Variable p.chil is a set of process identifiers, and it contains the identifiers of the children of process p in the tree. Variable p.next is the unicast routing table of p, that is, p.next[d] gives the next-hop neighbor of p to reach destination d.

Obtaining the values of p.par and p.chil is performed as follows. If process p chooses to join the tree, p sends a *parent* message to neighbor p.next[root], i.e., to its next-hop neighbor to the root of the tree, and assigns this neighbor to p.par. Let q = p.next[root]. When q receives a *parent* message from p, it adds p to q.chil, and then returns a *child* message to p.

Each process sends a *parent* message periodically to its parent. If a process does not receive a *parent* message from a child within some timeout period, it removes the child from its child set. If a process has no children and is not a member of the multicast group, then it removes itself from the multicast tree by setting its parent variable to **nil**.

From the above, all network edges (p, p.par) will form a multicast tree, and also, $q \in$ p.chil if and only if q.par = p.

When the unicast routing tables change, problems in

[1] Since the network is viewed as a graph, we use the terms *node* and *process* interchangeably.

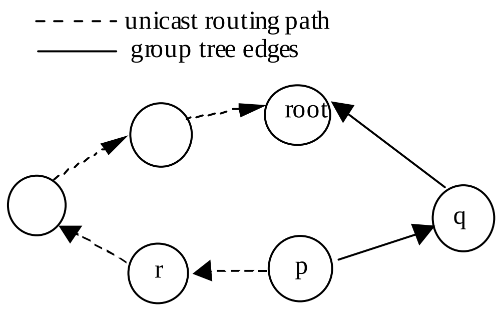

Figure 2: Changing Parents

unicast routing may arise, such as temporary routing loops. We assume these problems are temporary, and the unicast routing tables will converge to a consistent value.

If the unicast spanning tree changes due to changes in the unicast routing tables, the multicast tree changes accordingly, and becomes a subgraph of the new spanning tree. However, while these changes occur, the multicast tree may become disconnected, disrupting the flow of data messages. We next address how to overcome this problem.

3.2 Multicast Tree Integrity

In this section, we enhance our basic protocol by restricting when a process changes its parent. The purpose is to ensure that a process that has joined the multicast tree remains connected to the tree while changes in the unicast routing tables occur.

To show how a process becomes disconnected, consider Figure 2. Assume p.par = q, p.next[root] = r, and all edges in the unicast path from p to the root are not in the multicast tree. Once p chooses r as its parent, q may time out and remove p from its child set, before all the edges in the unicast path from p to the root have been added to the multicast tree. This temporarily removes p from the tree.

To prevent this, p should not change its parent from q to r until r is connected to the multicast tree. We say that a process r is connected to the multicast tree if r.par ≠ **nil**.

Recall that process r chooses to join the multicast tree if either it is in multicast group, or if its child set is not empty. To ensure r's child set is not empty, p sends a *parent* message to r as if r were its parent. Then, r adds p to its child set, and returns a *child* message to p. Each *child* message will include a Boolean bit indicating if the sender is connected to the multicast tree. If p receives a *child* message from r with the connected bit equal to true, then p may safely choose r as its parent.

2.3 Loopless Multicast tree

It is easy to show that for the above protocol, if a process p has p.par ≠ **nil**, then the path obtained by following the parent variables starting from p ends in the root node, or p is involved in a temporary loop. That is,

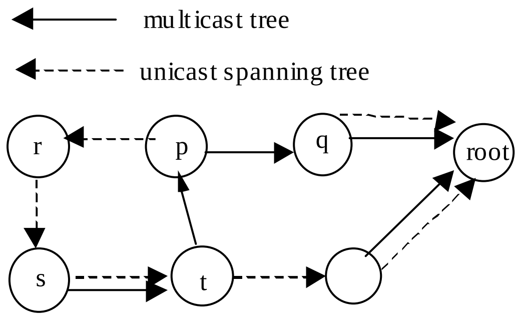

Figure 3: Temporary Loops in Multicast Tree

the parent variables could form a loop, in which case p is temporarily unreachable from the root of the tree.

To see this, consider the system state depicted in Figure 3. In this state, p sends a *parent* message to r, and r sends a *parent* message to s. Then, s returns a *child* message to r, and r sets r.par = s, and r returns a *child* message to p. Process p sets p.par = r, completing the loop. Note that the loop is possible even if the unicast routing tables are loop-less, as shown in the figure. Thus, restricting the multicast tree protocol to work only in conjunction with a loop-less unicast routing protocol is not sufficient. The problem must be solved by further refining the protocol.

The refinement consists of introducing a diffusing computation for loop avoidance. Each process p maintains an integer timestamp, p.ts. The root process increments its timestamp periodically. A non-root process may not increment its timestamp on its own. Rather, each process includes its timestamp in each *child* message. If a process receives a timestamp from its parent that is larger than its own, it sets its timestamp to the parent's timestamp.

Assume the routing tables indicate that p should choose a different parent. Call this new parent the *tentative parent* of p. In this case, p ignores the timestamps received from its current parent. When p receives a *child* message from the tentative parent, with a timestamp greater than p's timestamp, p changes parents by setting p.par to the tentative parent, and sets p.ts to the received timestamp.

The reason no loops are created is simple. All processes in the multicast subtree rooted at p have a timestamp at most p's timestamp. When the tentative parent provides to p a timestamp greater than p's timestamp, it indicates to p that it is not part of the subtree of p. Thus, choosing this neighbor as a new parent cannot form a loop.

2.4 Protocol Specification

We next specify the loopless multicast routing protocol. We begin with a brief description of our notation. Each process consists of a set of actions, separated by the symbol []. Each action is of the form

guard → *command*

When the guard is true, the command is enabled for execution. Commands from different actions are executed one at a time and in any order, provided the command is enabled when chosen for execution, and no enabled command is continuously ignored for execution.

The channel from process p to process q is denoted ch.p.q. The number of messages of type msg_type in this channel is denoted msg_type#ch.p.q. We assume channels may lose and reorder messages, but not duplicate them. A similar notation may be found in [10].

Process p below represents any network process. Its variables are par, chil, and ts, which we explained earlier. It has three inputs: mbr, which indicates if p is currently a member of the multicast group, ngh, which indicates the neighbors of p, and route, which is p's unicast routing table. Also, p has an identifier, join, which is a shorthand for the expression $mbr \vee chil \neq$ **empty**.

If p is the root process, then always par.p = p, that is, the parent of the root process is itself (it cannot be **nil** since the root must always be connected to the tree).

The specification of process p is as follows.

```
process p
const
  root   :  process_id
inp
  mbr    :  boolean,
  ngh    :  set of process_id - {p},
  route  :  array [process_id] of element of ngh
var
  par    :  element of ngh ∪ {p, nil}
  chil   :  subset of ngh,
  ts     :  integer
  j      :  ngh
always
  join   :  (mbr ∨ chil ≠ empty)
begin
   /*  refresh parent */
  timeout (parent#ch.p.par + child#ch.par.p) = 0 ∧
           par ≠ nil ∧ par ≠ p   →  send parent to par
[] /* request new parent */
  timeout par ≠ p  ∧ join ∧
  (parent#ch.p.route[root] + child#ch.route[root].p) = 0
        →        send parent to route[root]
[] /* receive parent message */
  rcv parent from any j   →  chil := chil ∪ {j};
                             send child(par ≠ nil, ts) to j
[] /* receive child message */
  rcv child(b, t) from any j →
      if ts < t ∧ b ∧ j = route[root] ∧ join  then
            ts := t;  par := j
      fi
[] /* disconnect from lost child */
  timeout   (some j ∈ chil ∧ child#ch.p.j = 0 ∧ j.par ≠ p)
      →       chil := chil - {j}
[] /* leave the tree */
  ¬join ∧ par ≠ p       →         par := nil
[] /*  root creates the next timestamp */
  par = p      →   ts := ts+1
end
```

The first action consists of a timeout. This action is enabled if there are no *parent* messages from p to its parent, there are no *child* messages from p's parent to p, if p is connected to the tree, and if p is not the root. If so, p sends a new *parent* message to its parent, p.par.

Even though this timeout guard refers to the contents of the channels (which are not visible to a process in a message passing system), this guard can be implemented using conventional timers, as explained in [10].

The second action is also a timeout action. It sends a *parent* message to the next-hop neighbor along the unicast path to the root, provided p chooses to join the tree.

The third action receives a *parent* message from any neighbor. Thus, p adds the neighbor to its child set, and returns a *child* message with p's timestamp and a bit indicating if p is connected to the tree.

The fourth action receives a *child* message from any neighbor. Process p adopts this neighbor as its new parent if it's the next-hop to the root and has a greater timestamp.

The fifth action is also a timeout. If p waits long enough without receiving a *parent* message from a neighbor, and the neighbor is a child of p, it means the neighbor no longer considers p as its parent. Then, p removes the neighbor from its child set. Again, an action of this form can be implemented with conventional timers [10].

In the sixth action, process p leaves the multicast tree by setting p.par to **nil**, provided it is not supposed to join the multicast tree (i.e., $\neg$join), and, obviously, if it is not the root, since the root should always be part of the tree.

Finally, in the last action, if process p is the root process, it may increase its timestamp at any time.

4. Making The Root Dynamic

Above, we presented a multicast tree protocol that adapts to changes in the unicast routing tables, while preserving the integrity of the multicast tree. In this section, we enhance this protocol to allow the root node to be changed, in order to obtain a more efficient multicast tree. That is, if the current choice of root node becomes non-optimal, due to changes in the network topology or traffic load, then a new root node may be chosen, and the tree is adjusted accordingly. During this adjustment, the integrity of the multicast tree is preserved.

A root node may be chosen in many different ways. For example, it may be chosen to minimize the total delay between any node in the multicast group to the root node plus the delay from the root node to any other node in the

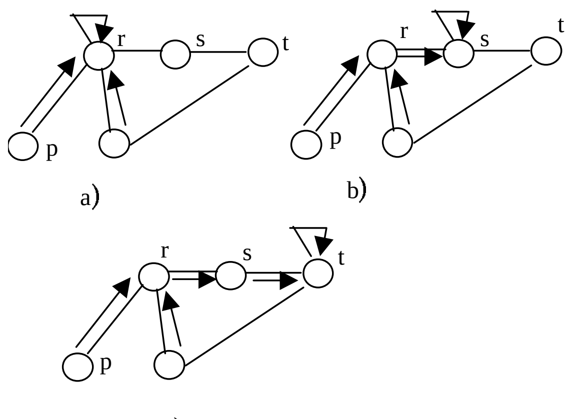

Figure 4: Changing the root node of the tree

multicast group. This would minimize the delay of sending a packet between any two nodes on the tree.

However, our objective is not to obtain the best way to choose a root node, but rather to adapt the multicast tree algorithms to change root nodes. Thus, we simply assume there exists a scheme to elect a new root for the multicast tree. Also, we assume there is a mechanism through which the current root node learns the identity of the next root node. E.g., there could be a system of servers, such as the Internet's Domain Name System, which the current root could use to learn the identity of the next root node.

Since the root node may change, we must define which process is currently the root. Process p is the root of the multicast tree if p.par = p. Also, each process has an input, called *best* (best root), which indicates which node in the network would currently be the best root. The only node making use of this input is the current root of the tree.

To change the root of the multicast tree without compromising the integrity of the tree, it is best to make gradual changes to the tree. Therefore, we choose to move the root of the tree one hop at a time. For example, consider Figure 4. Assume r is the current root (4-a), and t should be the next root. In this case, we move the root by one hop in the direction of t. That is, if s = r.route[t], then s should become the next root, and r sends a *root* message to s. When s receives the message, it adds r to its child set, and sets s.par = s, which in effect turns s into the root. Process s will return a *child* message to r, which causes r to set r.par = s (4-b). Similarly, since t is the best root for the tree, s sends a *root* message to t, causing t to add s to its child set, and set t.par = t. Then t sends a *child* message to s, and s sets s.par = t (4-c).

Included in the *root* message is the timestamp of the root. This informs the new root of the largest timestamp in the tree, and thus, its next timestamp must be greater than this value. Note that it is possible that the next root of the tree is a descendant of the original root. For example, in Figure 4(a), assume the next root should be p rather than t. In this case, when p receives the *root* message from r, it sets p.par = p, and r sets r.par = p. Thus, a loop is not introduced in the tree. Therefore, even if the new root is a descendant of the old root, the technique is sound.

Notice that since the root changes, then *the unicast spanning tree that serves as the foundation for the multicast tree also changes.* To ensure that the multicast tree converges to become a subset of the correct unicast spanning tree, each *child* message in the tree, besides carrying a timestamp, also carries the process identifier of the root node which created this timestamp. In this way, all the nodes in the tree will learn who is the current root of the tree, and choose their parents accordingly.

There is one final issue. Assume a node not on the multicast tree decides to join the multicast group, and hence, also the tree. This node may not be aware of which node is currently the root of the tree, and thus, it does not know the unicast path to the root. To remedy this, we assume there is at least one default node, which is always on the multicast tree, and the identity of this node is well known. Thus, a node joining the tree will first establish a path to the default node. Once the node has joined the tree through the default node, it learns the identity of the current root, and establishes an efficient path to the root.

We are now ready to present the specification of the dynamic root protocol. Process p has a new input, best, which gives the best choice for the root of the multicast tree, and a new variable, new, which is the new root, i.e., the next hop towards the best root. Also, process p has a constant, def, with the default process, and the expression join is enhanced to also be true if p is the default node.

The specification is as follows.

```
process p
const
  def        :   integer
inp
  mbr   :   boolean
  ngh   :   set of process_id - {p},
  route :   array [process_id] of element of ngh,
  best  :   process_id
var
  root  :   process_id,  /* current root */
  new   :   ngh ∪ {p},   /* new root */
  par   :   element of ngh ∪ {p, nil},/* parent */
  chil  :   subset of ngh,
  ts    :   integer
  j     :   ngh
always
  join  :   (mbr ∨ chil ≠ empty ∨ def = p)
begin
  /* refresh parent */
  timeout (parent#ch.p.par + child#ch.par.p) = 0 ∧
            par ≠ nil ∧ par ≠ p    →   send parent to par
[] /* request new parent */
```

```
    timeout par ≠ p ∧ join ∧ par ≠ nil ∧
    (parent#ch.p.route[root] + child#ch.route[root].p) = 0
     →                send parent to route[root]
[] /* join the tree */
   timeout join ∧ par = nil ∧ p ≠ def ∧
(parent#ch.p.route[def] + child#ch.route[def].p) = 0
 →                    send parent to route[def]
[] /* receive parent message */
   rcv parent from any j →
                      chil := chil ∪ {j};
                      send child(par ≠ nil, ts, root) to j
[] /* receive child message */
   rcv child(b, t, r) from any j →
         if ts < t ∧ b ∧ j = route[root] ∧ join  then
                      ts := t; par := j;  root := r
         fi
[] /* disconnect from lost child */
   timeout (some j ∈ chil ∧ child#ch.p.j = 0 ∧ j.par ≠ p) →
                      chil := chil - {j}
[] /* leave the tree */
   ¬join ∧ par ≠ p         →    par := nil; ts := 0
[] /*  root creates the next timestamp */
   par = p ∧ new = p    →    ts := ts + 1;  new := route[best]
[] /* root asks for a new root */
   timeout par = p ∧ (∀ i, i ∈ ngh, root#ch.p.i = 0) ∧
     new ≠ p          →           send root(ts) to new
[] /*  being asked to become the new root */
   rcv root(t) from j →
         if t ≥ ts  then par := p; root := p; new := p; ts := t+1
         fi;
         child := child  ∪ {j};
         send child(par ≠ nil, ts, root) to j
end
```

The specification has ten actions. The first action sends a new *parent* message to the current parent of p, and is the same action as in the previous version of the protocol. The next action sends a *parent* message to the new parent, and is the same as before, except that the guard is a little stronger. The message is not sent if p.par = **nil**, since in this case p does not know which node is currently the root.

The third action sends a parent message towards the default node, which is when p wants to join the tree, but does not yet have a parent. The fourth and fifth actions receive a *parent* and *child* message. They differ from before in that the root value is passed in the *child* message.

The sixth action removes a child from the child set, and the seventh action removes process p from the tree. Both are unchanged. The eighth action increases the root's timestamp, and is as before, except that the timestamp is not increased if a new root has been chosen. Also, if the timestamp is increased, a new root is computed.

In the ninth action, if p.new ≠ p, then, neighbor p.new should be the new root, and process p sends a root message to p.new. In the tenth action, p receives a root message from a neighbor. The root message is accepted if it has a timestamp at least p.ts. If it is accepted, p turns itself into the root of the tree. Whether accepted or not, the neighbor is added to the child set of p.

5. Tolerating Transient Faults

In the previous section we presented a protocol which dynamically changes the root of the tree without compromising the tree's integrity. Our final enhancement to the protocol is to add fault-tolerance, in particular, the resulting protocol will be self-stabilizing. That is, if the state of the system (its variables and channel contents), is arbitrarily changed, the system converges to a good operating state. In order to be self-stabilizing, we assume that the unicast routing algorithm is also self-stabilizing. Many unicast routing algorithms, such as distance-vector and link-state routing, are known to be self-stabilizing.

We next address the problems to be overcome if an arbitrary state is to converge to a good operating state.

5.1 Timeout Implementation

A process implements a timeout by keeping track of the last time it sent a message, such as the *parent* message, and waiting enough time to receive a response message, i.e., the *child* message. However, due to a fault, the process may loose track of when it sent the message, or a fault may introduce additional messages into the channel.

This can be overcome as follows. Consider *parent* and *child* messages. Normally, a process does not send a new *parent* message until the previous *parent* and *child* messages to and from this neighbor have been received or lost. Assume, however, that spurious *parent* and *child* messages exist in the channel. If the process sends *parent* messages at a rate lower than the rate at which they can be consumed by the neighbor, and if the process consumes *child* messages at a rate faster than it generates *parent* messages, then eventually the timeout implementation will be correct. That is, a new *parent* message will be sent only of there are no old *parent* or *child* messages.

5.2 Timestamps Out of Order

In a good state, node timestamps should be decreasing from the root to the leaves of the tree. However, due to a fault, this may not be true at the initial state.

In order to re-establish this relationship, each *parent* message will include the timestamp of the child. When the parent receives the message, it compares this timestamp with its own timestamp. If the child's timestamp is greater, then it sets its own timestamp to its child timestamp.

By using this technique, eventually the relationship between the timestamps of any pair of parent and child nodes will be restored automatically. Furthermore, it will always be the case that, as long as a node is connected to the tree, its timestamp is non-decreasing.

5.3 Existing Loops

Once timestamps are in order, no new loops are created, since no process will choose one of its descendants as its new parent. However, due to faults, the initial state of the system may contain chains of nodes whose parent variables form a loop. These initial loops must be broken.

To break these loops, we assume an upper bound D on the network diameter. Each node computes an estimate of its height in the tree as the maximum of its children's height plus one. To inform its parent of its height, each node includes its height in each *parent* message. A loop is detected when the node's height becomes greater than D.

To implement this, each node maintains an array with the heights of all its children, and updates this array as it receives *parent* messages. If the maximum of the children's heights plus one reaches a value greater than D, then a loop exists.[2] The node breaks the loop by simply turning itself into a root node, i.e., it sets its parent variable to itself. There is not much else the node can do, since it is likely that the node's information about who is the current root node is probably incorrect, due to the loop.

Obviously, breaking loops will generate multiple trees. This is a problem that must be dealt with anyway, since faults in the initial state could have created multiple initial trees. Notice, however, that under a fault-free execution, we always have a single tree. Furthermore, if all the roots of these trees have the same value for the best root, then all these roots will converge to the same best root, and a single tree will be obtained.

5.4 Relationship Between Local Variables

The local variables of a process must have the proper relationship. For example, if a node has no parent, i.e., if its par variable is **nil**, then its timestamp should be zero. To re-instate the right relationship between local variables, we will add a few corrective actions that will check for the improper relationship and correct it.

5.5 Protocol Specification

Finally, we are ready to present the specification of the fault-tolerant protocol. It contains an additional array, height, which stores the height of each child.

```
process p
const
  def    :  process_id
inp
  mbr    :  boolean,
  ngh    :  set of process_id - {p},
  route  :  array [process_id] of element of ngh,
  best   :  process_id
var
  root   :  process_id,
  new    :  ngh ∪ {p},
  par    :  element of ngh ∪ {p, nil},
  chil   :  subset of ngh,
  height :  array [ngh] of 0 .. D,
  ts     :
  j      :  ngh
always
  join   :  (mbr ∨ chil ≠ empty ∨ def = p)
begin
/* local relationships */
   par = p     →   root := p
[] j ∉ chil    →   height[j] := 0
[] par = nil   →   ts := 0
[] /* become root if default node or a loop is found */
  (par ≠ nil ∧ par ≠ p ∧ max(height) > D) ∨
  (p = def ∧ par = nil) →
                   par := p; root := p; new := p
[] /* refresh parent */
  timeout (parent#ch.p.par + child#ch.par.p) = 0 ∧
  par ≠ nil ∧ par ≠ p   →
                   send parent(ts, max(height)+1) to par
[] /* request new parent */
  timeout par ≠ nil ∧ par ≠ p ∧ root ≠ p ∧ join ∧
  (parent#ch.p.route[root] + child#ch.route[root].p) = 0
  →                send parent(0, 0) to route[root]
[] /* join the tree */
  timeout join ∧ par = nil ∧ p ≠ def ∧
  (parent#ch.p.route[def] +child#ch.route[def].p) = 0
  →                send parent(0, 0) to route[def]
[] /* receive parent message */
  rcv parent(t, h) from any j →
          chil := chil ∪ {j};
          if par ≠ nil ∧ t > ts then ts := t; height[j] := h
          else if par ≠ nil ∧ t = ts then height[j] := h
          else if par = nil ∨ t < ts then height[j] := 0
          fi;
          send child(par ≠ nil, ts, root) to j
[] /* receive child message */
  rcv child(b, t, r) from any j →
```

[2] We were tempted to use the depth of a node, rather than its height, to break loops, but we obtained an example in which the depth calculated by a node is greater than D, eventhough no fault occurred and no loop existed.

```
            if (j = route[r] ∨ par = p ∨ (j = par ∧ r ≠ root))
               ∧ ts < t ∧ b ∧ then
                        ts := t; par := j; root := r;
                        for each i in ngh
                             height[i] := 0
            else if ts = t ∧ b ∧ j = par then
                        root:= r
            else if j = par ∧ ¬b then
                        par := p; root := p; new := p
            fi
[] /* disconnect from lost child */
   timeout (some j ∈ chil ∧ child#ch.p.j = 0) ∧
   (∀ t,h,  t > 0,  parent(t,h)#ch.j.p = 0) ∧ j.par ≠ p   →
            chil := chil - {j};
            height[j] := 0
[] /* leave the tree */
   ¬join ∧ par ≠ p →   par := nil; ts := 0
[] /* root creates the next timestamp */
   par = p ∧ new = p    →   ts := ts+1; new := route[best]
[] /* root asks for a new root */
   timeout par = p ∧ new ≠ p ∧
   (∀ i, i ∈ ngh, root#ch.p.i = 0) ∧
   (parent#ch.p.new + child#ch.new.p = 0)    →
               send root(ts) to new
[] /* being asked to become the new root */
   rcv root(t) from any j →
        if t ≥ ts then par :=p; root := p; new := p; ts := t+1
        fi;
        child := child  ∪ {j};
        send child(par ≠ nil, ts, root) to j
end
```

The protocol contains few additional actions. The first three correct some relationships between local variables. The next action makes p a root node if a loop is found or if p is the default node and it has no parent.

The parent message has a timestamp of zero if the child is not yet pointing to the new parent, and a timestamp greater than zero if it is pointing to the new parent. The timeout to remove a neighbor from the child set is strengthened to make sure no incoming parent messages with nonzero timestamps exist from the neighbor.

6. Concluding Remarks

The propagating-timestamps technique has been used in unicast routing protocols [1]. The timestamp's purpose in these protocols is to quickly break routing loops that form in networks whose topology changes quickly, such as mobile networks [14]. In our protocol, we use the technique somewhat differently. The timestamps are used to ensure that the multicast tree remains loopless.

The use of an unbounded timestamp is a weakness, since no value in reality is unbounded. However, there are several techniques that may be used to bound the timestamp. For example, a clock synchronization protocol can be run in the network, and the root of the tree ensures that its new timestamp is never greater than the real-time clock. In this way, if a node has a timestamp greater than the real-time clock, it reduces its timestamp to the clock value, and does not accept a message timestamp whose value is greater than the real-time clock.

References

[1] Arora A., Gouda M., Herman T., ``Composite Routing Protocols'', *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, 1990.

[2] Alaettinoglu C, Shankar U., ``Stepwise Design of Distance-Vector Algorithms'', *12th Symposium on Protocol Specification, Testing and Verification*, 1992.

[3] Ballardie T., ``Core Based Tree Multicast'', Internet RFC, work in progress.

[4] Ballardie T., Francis P., Crowcroft J, ``Core Based Trees: An Architecture for Scalable Inter-Domain Multicast Routing'', *ACM SIGCOMM Conference*, 1993.

[5] Cobb J, Gouda M, "The Request-Reply Family of Group Routing Protocols", *IEEE Transactions on Computers*, Vol. 46 No. 6, June 1997.

[6] Deering S., Cheriton D., ``Multicast Routing in Datagram Networks and Extended LANs'', *ACM Transactions on Computer Systems*, Vol 8., No 2., May 1990.

[7] Deering S. et. al., ``An Architecture for Wide-Area Multicast Routing'', *ACM SIGCOMM Conference*, 1994.

[8] Dalal, Y. K., Metcalfe, R. M., ``Reverse Path Forwarding of Broadcast Packets'', *Communications of the ACM*, Vol. 21, No. 12, Dec. 1978.

[9] Gouda M., ``The Triumph and Tribulation of System Stabilization'', *International Workshop on Distributed Algorithms*, 1995.

[10] Gouda M., *The Elements of Network Protocols*, Wyley Publishers, 1998.

[11] Gouda M., Schneider M., ``Maximum Flow Routing'', *Joint Conference on Information Sciences*, 1994.

[12] Cheng C., Riley R., Kumar S, Garcia-Luna-Aceves J., ``A Loop-free Bellman-Ford Routing Protocol without Bouncing Effect'', *ACM SIGCOMM Conference*, 1989.

[13] Kahle B., Schwartz M., Emtage A., Neuman B., ``A Comparison of Internet Resource Discovery Approaches'', *Computing Systems*, Vol. 5 No. 4., Fall 1992.

[14] Perkins, C. et. al., ``Ad Hoc Networking in Mobile Computing'', *ACM SIGCOMM Conference*, 1994.

[15] Shin K. G., Chen M., ``Performance Analysis of Distributed Routing Strategies Free of Ping-Pong-Type Looping'', *IEEE Transactions on Computers*, 1987.

[16] Wilbur S., Handley M., ``Multimedia Conferencing: from Prototype to National Pilot'', *INET' 92 International Networking Conference*.

[17] Correctness proos for the protocolsl in this paper may be found in http://www.utdallas.edu/~jcobb