Group Routing without Group Routing Tables

Jorge A. Cobb
University of Houston
Houston, TX 77204-3475
{cobb@cs.uh.edu}

Mohamed G. Gouda
The University of Texas at Austin
Austin, TX 78712-1188
{gouda@cs.utexas.edu}

Abstract

We present a group routing protocol for a network of processes. The task of the protocol is to route data messages to each member of a process group. To this end, a tree of processes is constructed in the network, ensuring each group member is included in the tree. To build this tree, the group routing protocol relies upon the unicast routing tables of each process. Thus, group routing is a composition of a unicast routing protocol, whose detailed behavior is unknown but its basic properties are given, and a protocol that builds a group tree based upon the unicast routing tables. The design of the group routing protocol is presented in three steps. First, a basic group routing protocol is presented and proven correct. Then, the protocol is refined twice, strengthening its properties with each refinement. The final protocol has the property of adapting the group tree to changes in the unicast routing tables without compromising the integrity of the group tree, even in the presence of unicast routing loops.

1. Introduction

In this paper, we present a group routing protocol for a network of processes. In group routing, the processes in the network are organized into groups. When the destination of a data message is a process group, the data message is forwarded along the network until it is received by every member of the process group. Group routing has many applications, such as audio and video conferencing [17], replicated database updating and querying, and resource discovery [14].

For simplicity, we present a group routing protocol for a single process group. The extension to multiple groups is straightforward.

To forward data messages to all group members, a group tree is constructed. 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. The tree contains each member of the process group, plus any additional processes necessary to connect the tree together. When a data message is addressed to the process group, the message is forwarded along the entire tree. In this way, each process in the tree, and hence each group member, receives the data message.

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

Many unicast routing algorithms exist in the literature, e.g., [1, 2, 12, 13, 16]. These algorithms have many differences, such as using different metrics in choosing the best path between two processes. However, common to all of these is the ability to change the routing tables in response to varying network conditions, such as fluctuations in traffic, or changes in the network topology.

To maintain the efficiency of the group tree, when the unicast routing tables change, the tree is restructured to reflect these changes. In addition, the protocol has the nice property that it maintains the integrity of the group tree while the unicast routing tables are changing. That is, it does not introduce temporary loops, it always maintains the tree connected, and it never removes a group member from the tree.

Obtaining a broadcast tree from the unicast routing tables was introduced in [8]. In [6] [7], the broadcast tree is trimmed into a group tree that excludes those processes not needed to reach the members of the multicast group. Unfortunately, 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 [3] [4], a group tree is initially built from the unicast routing tables. However, the tree does not adapt itself to changes in these tables, and thus may lose its efficiency as network conditions change.

The design of our group routing protocol is based on the paradigm of protocol composition. The protocol is correct when composed with any unicast routing protocol that satisfies the following basic requirement. The routing tables may fluctuate, but they eventually converge to a value that, for each pair of processes p and q, defines a path from p to q. In this way, the group routing protocol performs as desired even when the details of the particular unicast routing protocol in use are unavailable.

We design our group routing protocol in three steps. First, we present a basic version of the protocol, and prove its correctness. Then, we refine the protocol twice. Each refined version improves upon the previous version by satisfying all of the correctness properties of the previ-

ous version, plus some additional stronger properties. The end result is a group routing protocol that adapts itself to the unicast routing tables, and in addition maintains the integrity of the group tree at all times.

Due to space restrictions, all proofs are deferred to [5].

The structure of the paper is as follows. In Section 2, the notation to specify each group routing protocol is introduced. The basic group routing protocol is introduced in Section 3. In Section 4, the correctness properties of the basic protocol are presented. The first refinement of the basic protocol, along with its correctness properties, is presented in Section 5. The second refinement is presented in Section 6. In Section 7, possible further refinements are mentioned. Concluding remarks are given in Section 8.

2. Protocol Notation

In this paper, we present a family of group routing 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, a node represents a process, and an edge between processes p and q represents two first-in-first-out communication channels, one channel from process p to process q and another channel from process q to process p. The channel from process p to process q is denoted by ch.p.q.

Each process is assigned a unique identifier, which we assume to be of type integer. We say that processes p and q are neighbors iff they are joined by an edge in the network graph.

Each process is defined by a set of global and local constants, a set of local variables, and a set of actions. If multiple processes have the same name for a local variable, say v, then we denote variable v in process p by p.v.

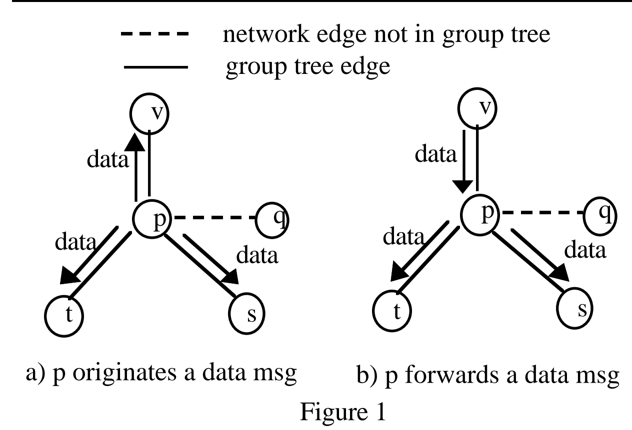
Actions are separated from each other with the symbol [], using the following syntax:

begin *action* [] *action* [] . . . [] *action* **end**

Each action is of the form *guard* → *command*. A guard is either a boolean expression involving the local variables of its process, or a receive statement of the form **rcv** msg **from** q, where msg is a message type and q is the identifier of a neighboring process. A command is constructed from sequencing (;), conditional (**if fi**), and iterative (**for rof**) constructs that group together **skip**, assignment, and send statements of the form **send** msg **to** q. Similar notations for defining network protocols are discussed in [9] [11].

An action in process p is said to be *enabled* if its guard is either a boolean expression that evaluates to true, or a receive statement of the form **rcv** msg **from** q, and there is a message of type msg at the head of channel ch.q.p.

An execution step of a protocol consists of choosing any enabled action from any process, and executing the action's command. If the guard of the chosen action is a receive statement **rcv** msg **from** q, and this action is in process p, then, before the action's command is executed,



a) p originates a data msg b) p forwards a data msg

Figure 1

a message of type msg is removed from the head of channel ch.q.p. Protocol execution is fair, that is, each action that remains continuously enabled is eventually executed.

Multiple actions that differ by a single value can be abbreviated into a single action by introducing parameters. For example, let j be a parameter whose type is the range 0 . . 2. The action

rcv msg **from** j → x := j

is a shorthand notation for the following three actions.

rcv msg **from** 0 → x := 0
[] **rcv** msg **from** 1 → x := 1
[] **rcv** msg **from** 2 → x := 2

3. The Basic Protocol

In this section, we define a protocol for routing data messages to every member of a process group. Any member of the group can generate data messages, and each data message is forwarded to each member of the group. Since group members do not necessarily have direct channels between each other, data messages are forwarded from one process in the network to another until they reach each group member.

To route messages between group members, we construct a group tree of processes. The edges of the tree are a subset of the edges in the process network, and the set of processes in the tree contains all members of the process group. Each data message is forwarded along the entire group tree. In this way, each process in the tree, and hence each group member, receives each data message. Notice that there may be processes that are nodes in the group tree but are not members of the process group. These additional nodes are needed to ensure the group tree is connected.

To forward a data message along the group tree, the originator of the message forwards the message to all of its neighbors in the tree, as shown in Figure 1. When a process receives a data message from a neighbor in the tree, it forwards the message to all of its neighbors in the tree except the one from which the message was received.

To determine which edges in the network belong to the group tree, we take advantage of the spanning trees pro-

vided by the unicast routing tables in the network. The unicast routing table at each process p determines, for each possible destination process r, which neighbor is the next-hop in the unicast path from p to r. Hence, a spanning tree rooted at r is obtained by choosing all network edges (p, q), where q is the next-hop in the unicast path from p to r.

To build a group tree, we designate one process in the network as the root of the tree. The parent of each process p in the tree is the next-hop neighbor in the unicast path from p to the designated root process. Thus, the group tree is a subset of the unicast spanning tree whose root is also the designated root process.

Note that the group tree must contain all members of the process group, plus any additional processes required to complete the tree. A process p determines that it belongs in the group tree as follows. If p is a group member, then p belongs in the group tree. If p is not a group member, but it has a neighbor that belongs in the group tree, and the neighbor's parent in the group tree is p, then p also belongs in the group tree.

The general strategy is the following. If a process determines that it belongs in the group tree, it sends a request message to its parent in the tree. When the parent receives the request, it adds the process to its set of children and returns a reply to the child. Each process in the tree sends a request periodically to its parent. To ensure that at most one request is outstanding at any time, a process will not send a new request to its parent until a reply is received for the previous request. If a parent does not receive a request from a child within some timeout period, it removes the process from its set of children.

The unicast routing protocol adapts the unicast routing tables in each process to changes in network conditions, such as traffic loads. While the unicast routing tables are changing, problems in unicast routing may arise, such as routing loops. We assume that these problems are temporary, and that the unicast routing tables will converge to a consistent value.

If the unicast spanning tree changes, the group tree changes accordingly, and becomes a subgraph of the new spanning tree. However, while these changes are occurring, the group tree may become disconnected, disrupting the delivery of data messages. In Sections 5 and 6, we refine our solution to ensure that the integrity of the group tree is never violated.

We next present the code for each process p in the network. Its constants and variables are as follows.

Process p has a global constant, root, which is the identifier of the group member chosen as the root of the group tree. It also has two local constants, also known as inputs. Input nbr p is a set containing the identifiers of the neighbors of process p. Input mbr is a boolean indicating whether p is a member of the process group or not.

Process p maintains in variable pr the identifier of its parent in the group tree, and maintains in variable chl the set of neighbors which are its children in the group tree. If process p determines that it does not belong in the group tree, it assigns its own identifier p to pr.

The function ROUTE(p, q) returns the next-hop neighbor in the unicast path from p to q. Note that this function may not always return the same value, since the unicast routing path may be undergoing some changes. Also, we assume that ROUTE(p, p) always returns p. Hence, if root = p, then pr is always equal to p.

Each process p in the network is defined as follows.

```
process p
const
  root  :  integer              (* root of group tree *)
inp
  nbr   :  set of integer,      (* neighboring processes *)
  mbr   :  boolean              (* is p a group member *)
var
  chl   :  set of integer,      (* children of p *)
  pr    :  integer,             (* parent of p *)
  wr    :  set of integer       (* set of pending replies *)
par
  j     :  nbr           (* j ranges over all neighbors *)
begin
   mbr  →  for each d in (chl ∪ {pr}) - {p} do
               send data to d
           rof
[] rcv data from j   →
           if j ∈ chl ∪ {pr} →
               for each d ∈ (chl ∪ {pr}) - {j, p}
                   send data to d
               rof;
               if   mbr     →  deliver data
               []  ¬mbr     →  skip
               fi
           [] j ∉ chl ∪ {pr} →  skip
           fi
[] chl ≠ ∅ ∨ mbr    →
           pr := ROUTE(p, root);
           if pr ≠ p ∧ pr ∉ wr   →  send rqst to pr;
                                    wr := wr ∪ {pr}
           [] pr = p ∨ pr ∈ wr   →  skip
           fi
[] rcv rqst from j   →  chl := chl ∪ {j};
                        send rply to j
[] rcv rply from j   →  wr := wr - j
[] timeout j ∈ chl ∧ j.pr ≠ p    →
           chl := chl - {j};
           if chl = ∅ ∧ ¬mbr   →  pr := p
           [] chl ≠ ∅ ∨ mbr    →  skip
           fi
end
```

Process p has six actions. In the first action, the process creates a data message and sends it to its parent and

children in the group tree. In the second action, process p receives a data message from one of its neighbors in the group tree, and forwards the message to all other neighbors in the tree. Also, if p is a member of the group, the data message is delivered to the application.

In the third action, process p checks whether it should be part of the group tree. If so, it chooses the next-hop neighbor to the root as its parent, and it sends a request to this neighbor, provided it is not waiting for a reply from an earlier request. In the fourth action, the process receives a request from a neighbor. Thus, the neighbor is added to the set of children, and a reply is returned to the child. In the fifth action, the reply is received from the parent.

The final action is a timeout action that models the expiration of a timer. We simplify the modeling of this action by using a global predicate as the action's guard, rather than modeling a real-time clock explicitly. In this global predicate, j.pr stands for the value of variable pr of neighbor j. Although timeout actions are modeled by a predicate, they can be implemented in practice using a real-time clock [11].

In the timeout action, if the process has a child j, and it has not received a request from this neighbor in a certain amount of time (i.e., $j.pr \neq p$), it removes the child from set chl. Furthermore, if the process determines that it should no longer take part in the group tree, it assigns its own identifier to pr.

Note that once the unicast routing tables and the group tree have achieved their final values, the periodic exchange of request and reply messages in the protocol occurs only between neighbors in the group tree. Thus, processes that are not in the group tree do not incur any processing overhead in maintaining the tree.

4. Protocol Properties

In this section, we characterize the behavior of the protocol of Section 3 using closure and convergence properties [9]. We first define the terms closure and convergence, and then present the specific closure and convergence properties of the protocol.

A *computation* of a network protocol **N** is a sequence (state.0, action.0; state.1, action.1; state.2, action.2; . . .) where each state.i is a state of **N**, each action.i is an action of some process in **N**, and state.(i+1) is obtained from state.i by executing action.i. Computations are fair, i.e., every continuously enabled action is eventually executed. Computations are also maximal, i.e., if state.j is the last state in a computation, then no action is enabled in state.j.

A *state predicate* of a network protocol **N** is a function that yields a boolean value (true or false) at each state of **N**. A state of **N** is an *S-state* iff the value of state predicate S is true at that state.

In our state predicates we make use of universal quantifications of the form:

$$\langle \forall\ x : R(x) : T(x) \rangle$$

This predicate is true iff every possible value of x that satisfies the boolean function R(x) also satisfies the boolean function T(x). We assume that the values of x are restricted to process identifiers in the network. If R(x) is omitted, x ranges over all process identifiers.

Let S be a state predicate of **N**. Predicate S is a *closure* in **N** iff at least one state of **N** is an S-state, and every computation that starts in an S-state is infinite and all its states are S-states. Predicate S is a *weak-closure* in **N** iff at least one state of **N** is an S-state, and every computation that starts in an S-state has an infinite suffix consisting solely of S-states.

Let S be a closure in **N**, and S' be a closure or a weak-closure in **N**. We say that S *converges* to S' iff every computation whose initial state is an S-state contains an S'-state.

From the above definition, if S converges to S' in **N**, and if the system is in an S-state, then eventually the computation should reach an S'-state. Furthermore, if S' is a closure, the computation continues to encounter only S'-states indefinitely. If S' is a weak-closure, the computation may encounter a finite number of non-S'-states, but this is followed by an infinite number of S'-states.

We next present the properties of the protocol of Section 3. To begin, we require the system to have a sensible initial state, which we characterize with predicate C0 below. The notation rqst#ch.p.q denotes the number of messages of type rqst currently in channel ch.p.q.

$$C0 \equiv S0 \wedge S1$$
$$S0 \equiv \langle \forall\ p,q :: (q \notin p.wr \wedge rqst\#ch.p.q + rply\#ch.q.p = 0) \vee (q \in p.wr \wedge rqst\#ch.p.q + rply\#ch.q.p = 1) \rangle$$
$$S1 \equiv \langle \forall\ p : p.chl = \varnothing \wedge \neg p.mbr : p.pr = p \rangle$$

Predicate C0 states that variable wr in each process accurately reflects whether a reply is expected for each neighbor. Also, it states that variable p.pr does not point to a neighbor if p does not belong in the group tree.

A simple initial state of the protocol that satisfies C0 is, for all p, $p.pr = p$, $p.wr = \varnothing$, and no request or reply messages in any channel.

Property 1

C0 is a closure

Thus, if C0 holds in the initial state of a computation, then it holds in all states of the computation.

Before presenting the next two properties, we define the following. Let GT be the set of edges in the group tree[1]. Edges in GT are directed, i.e., edge (p, q) differs from edge (q, p). Let p and q be any pair if neighboring processes.

a) $(p, q) \in GT \Leftrightarrow p.pr = q \vee p \in q.chl$
b) $(p, q) \in BE \Leftrightarrow p.pr = q \wedge p \in q.chl$
c) $(p, q) \in GE \Leftrightarrow (p, q) \in GT - BE$

[1] The term tree is a misnomer, since the graph of the group tree may temporarily contain loops or be disconnected. However, the graph will converge to a tree.

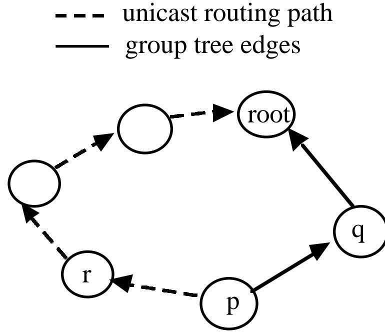


Figure 2: changing parents

Thus, edge (p, q) is in GT if either p considers q to be its parent or q considers p to be its child. The edges in GT are divided into a set of black edges, BE, and a set of gray edges, GE. An edge (p, q) is black if p and q agree, i.e., p considers q to be its parent, and q also considers p to be its child. An edge (p, q) is gray if p and q do not agree. This disagreement is temporary, and it occurs only during a period of transition in which the group tree is adapting to new changes in the unicast routing tables.

For the following two properties, we make the assumption that the unicast routing tables may fluctuate temporarily, but eventually remain fixed and define a spanning tree for each destination.

Let UT be the edges of the unicast spanning tree with the same root as the group tree. Let path(UT, p) be the edges in UT of the path from p to the root, and sub(UT, p) be the subtree of UT rooted at p.

Property 2
For any process p,
$C0 \wedge p.mbr$ converges to
$C0 \wedge p.mbr \wedge \langle \forall\, r,s : (r,s) \in path(UT, p) : (r,s) \in BE \rangle$

Property 3
For any process p,
$C0 \wedge \langle \forall\, r : r \in sub(UT, p) : \neg r.mbr \rangle$
converges to
$\langle \forall\, r, s : r \in sub(UT, p) : (r, s) \notin GT \rangle \wedge$
$C0 \wedge \langle \forall\, r : r \in sub(UT, p) : \neg r.mbr \rangle$

The first property states that if a process is a group member, then all the edges in its unicast path to the root will become black, i.e., they will be part of the group tree. The second property states that if all the processes in a subtree of UT are not members of the process group, then the entire subtree will be removed from the group tree. These two properties combined imply that GT will become the smallest subgraph of UT that connects all the members of the process group, as desired.

5. First Refinement: Maintaining Connectivity

We next refine the basic protocol of Section 3 by restricting when a process changes its parent. The purpose of this restriction is to ensure that a process that has joined the group tree remains connected to the tree while changes in the unicast routing tables are occurring. This refined protocol must preserve all the correctness properties presented in the previous section.

To show how a process becomes disconnected from the tree, consider the following. Assume p.pr = q, ROUTE(p, root) = r, and all edges in the unicast path from p to the root do not belong to the group tree, as shown in Figure 2. It is possible that, after p chooses r as its parent, q times out and removes p from its set of children before all the edges in the unicast path from p to the root have been added to the group tree. If this occurs, p will be temporarily disconnected from the group tree.

To prevent this from occurring, p should not change its parent from q to r until r is connected to the group tree. We say that a process r is connected to the group tree if r is the root, or if the edge between r and its parent is black and the parent of r is also connected to the group tree.

Recall that process r determines that it should join the group tree if either it is a member of the process group or if its child set is non-empty. To ensure r's child set is non-empty, p sends a request to r as if r were its parent. Process r adds p to its child set, and returns a reply to p. The reply includes a bit indicating if r is connected to the group tree. Process p continues to send requests to r until it receives a reply with this bit set to true. Then, p chooses r as its parent, i.e., it assigns r to p.pr, and thus becomes connected to the tree.

To perform the above, process p maintains two parent variables: the current parent, pr, which is connected to the group tree, and the tentative parent, tpr, which may not be connected to the tree. If p has no parent that is connected to the tree, then p.pr = p. When a reply is received from the tentative parent indicating that it is connected to the group tree, p turns its tentative parent into its current parent by assigning tpr to pr.

We next present the actions of the refined process. The first two actions are the same as in Section 3. The remaining four actions are as follows.

```
[] chl ≠ ∅ ∨ mbr      →
        tpr := ROUTE(p, root)
        if tpr ≠ p ∧ tpr ∉ wr →   send rqst to tpr;
                                  wr := wr ∪ {tpr}
        [] tpr = p ∨ tpr ∈ wr →   skip
        fi;
        if pr ≠ p ∧ pr ∉ wr   →   send rqst to pr;
                                  wr := wr ∪ {pr}
        [] pr = p ∨ pr ∈ wr   →   skip
        fi
[] rcv rqst from j    →
        chl := chl ∪ {j};
        b := (pr ≠ p ∨ p = root);
        send rply(b) to j
```

```
[] rcv rply(b) from j →
            wr := wr - {j}
            if  j = tpr ∧ b      →   pr := tpr
            [] ¬(j = tpr ∧ b)   →   skip
            fi
[] timeout j ∈ chl ∧ j.pr ≠ p ∧ j.tpr ≠ p ∧ rply#ch.p.j=0
   →        chl := chl - {j};
            if  chl = ∅ ∧ ¬mbr   →   pr := p; tpr := p
            []  chl ≠ ∅ ∨ mbr     →   skip
            fi
```

In the first action above, process p checks whether it should be part of the group tree. If it should be, the next-hop neighbor to the root is chosen as the tentative parent, and a request is sent to this neighbor. Similarly, a request is sent to the current parent, to prevent it from removing p from its child set.

In the second action above, a request is received from a neighbor. The neighbor is added to the child set as in the basic protocol. A reply is sent to the child indicating whether p is connected to the tree or not. Process p is connected to the tree if p has a parent that is also connected to the tree, i.e., if pr ≠ p, or if p is the root. In the third action above, a reply is received from a neighbor. If the reply is from the tentative parent, and the tentative parent is connected to the tree, then process p makes the tentative parent its current parent.

In the last action, a neighbor is removed from the child set after a timeout. If a neighboring process j does not consider p to be either its current or tentative parent (i.e., it has not sent a request to p for some time) and the last reply from p to j has been received by j, then j is removed from the child set. Furthermore, both pr and tpr are set to p if p no longer needs to take part in the group tree.

The reason we require the timeout period to be long enough to ensure j has received the last reply is as follows. Assume the reply indicates that p is connected to the tree. However, after removing j from the child set, p no longer needs to be on the group tree, and sets tpr and pr to p. If later j decides to rejoin the tree using p as a tentative parent, and j receives the old reply from p, it will erroneously conclude that p is connected to the tree, and prematurely choose p as its current parent.

We next present the properties of the protocol presented in this section. We begin by noting that if a process has a current parent, and the process is a member of the process group, then the process will continue to have a current parent indefinitely.

Property 4
For all processes p,
$p.pr \neq p \wedge p.mbr \wedge p \neq root$ is a closure

Property 4 is satisfied without making any assumptions about the behavior of the unicast routing tables, i.e., they are free to change at any point along the computation.

Let q be the current parent of p. To prevent p from being disconnected from the group tree, q must also have a current parent or be the root. We express this in predicate C1 below. This predicate must hold at the initial state of the system. It is stronger that C0, because it involves the new variables introduced in the refinement, i.e., p.tpr and the bit in the rply message, and it also involves the above requirement on connectivity.

$C1 \equiv C0 \wedge S2 \wedge S3 \wedge S4$
$S2 \equiv \langle \forall\ p, q : p.pr = q \wedge p \neq q : (p, q) \in BE \wedge (q.pr \neq q \ \vee q = root)\rangle$
$S3 \equiv \langle \forall\ p, q : rply(\mathbf{true}) \in ch.p.q : q \in p.chl \wedge (p.pr \neq p \vee p = root)\rangle$
$S4 \equiv \langle \forall\ p : p.chl = \emptyset \wedge \neg p.mbr : p.tpr = p\rangle$

A simple initial state of the protocol that satisfies C1 is, for all p, p.pr = p, p.tpr = p, p.wr = ∅, and no request or reply messages in any channel.

The refinement in this section also satisfies Properties 1, 2 and 3 of Section 4, with the exception that each occurrence of C0 in these properties is replaced by C1. Hence, C1 is a closure, and the group tree converges to the smallest subgraph of the unicast spanning tree that maintains all the group members connected.

Predicate C1 states that if a process p has a current parent, then the edge from p to its parent is black, and the current parent of p also has a current parent or is the root. Hence, either the group tree has a path of black edges from p to the root, or the path of black edges starting from p leads to a loop. The obvious shortcoming is that if a loop exists, then 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 chooses r as its tentative parent, and sends a request to r. Process r receives the request, and adds p to its child set. Then, r chooses s as its tentative parent, it sends a request to s, and s adds r to its child set. Thus, all the edges in the path from p to s become gray. Then, since s is connected to the tree, r chooses s as its current parent, making the edge (r, s) black. Subsequently, edge (p, r) also becomes black, forming a loop.

Note that the loop is possible even if the unicast routing tables are loop-less, as shown in the figure. Therefore, restricting the group routing 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, which is the topic of the next section.

6. Second Refinement: Preventing Routing Loops

We next present the second and final refinement of the group routing protocol. The purpose of this refinement is to avoid loops in the group tree when changes occur in the unicast routing tables. This loop-freedom must be achieved while still maintaining all the properties presented for the basic protocol and for the first refinement of the previous section.

The refinement consists of introducing a diffusing computation as a method for avoiding loops. Each process

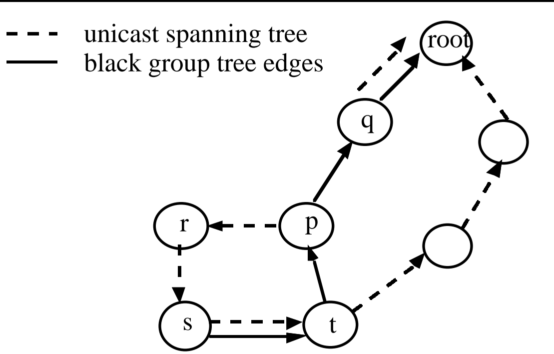


Figure 3: Temporary loops in group tree

maintains an integer timestamp variable, called ts. The root process increments its timestamp periodically. A non-root process may not increment its timestamp on its own. Instead, each parent includes its timestamp in each reply sent to a child. If a process receives a timestamp from its current parent that is larger than its own, then it sets its timestamp to the timestamp of its current parent.

When the routing tables indicate to a process that it should choose a different parent, i.e., when the tentative parent is not the current parent, the process ignores the timestamps received from the current parent. When the process receives a reply from the tentative parent with a timestamp greater than its own, and the tentative parent is connected to the tree, the process chooses the tentative parent as its current parent, and sets its timestamp to the tentative parent's timestamp.

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

The only actions requiring a change for the refinement are the action to receive a request and the action to receive a reply. In addition, a new action to increase the timestamp of the root is needed. These actions are as follows.

```
  p = root      →  ts := ts + 1
[] rcv rqst from j       →
        chl := chl ∪ {j};
        b := (pr ≠ p ∨ p = root);
        send rply(b, ts) to j
[] rcv rply(b, t) from j    →
        wr := wr - {j}
        if    j = tpr  ∧ b ∧ t > ts  →  pr, ts := tpr, t
        [] ¬(j = tpr ∧ b ∧ t > ts)  →  skip
        fi
```

Note that process p only accepts timestamps from the tentative parent tpr and not from its current parent pr. However, if p is not in the process of changing parents, then pr = tpr, and p will accept new timestamps from its current parent. Thus, p always has one parent from which it accepts new timestamps, whether it is in the process of changing parents or not.

We next present the properties of the protocol described in this section. Since we have introduced a new variable ts in each process, we need to strengthen the initial state of the system to reflect an appropriate value for these variables. The new initial state predicate C2 is defined next.

$C2 \equiv C0 \wedge S4 \wedge S5 \wedge S6 \wedge S7$

$S5 \equiv \langle \forall\ p, q : p.pr = q \wedge p \neq q :$
$\quad (p, q) \in BE \wedge (q.pr \neq q \vee q = root) \wedge q.ts \geq p.ts \rangle$

$S6 \equiv \langle \forall\ p, q, t : rply(\mathbf{true}, t) \in ch.p.q :$
$\quad q \in p.chl \wedge (p.pr \neq p \vee p = root) \wedge p.ts \geq t \rangle$

$S7 \equiv \langle \forall\ p : : p.ts \leq root.ts \rangle$

An initial state of the protocol that satisfies C2 is, for all p, p.pr = p, p.tpr = p, p.ts = 0, and p.wr = ∅, and no request or reply messages in any channel.

The refinement in this section satisfies Properties 1, 2 and 3 of Section 4, with the exception that each occurrence of C0 in these properties is replaced by C2. Thus, C2 is a closure, and the group tree converges to the smallest subgraph of the unicast spanning tree that maintains all the group members connected. Also, the refinement in this section satisfies Property 4 of the previous section, that is, a group member that has a current parent will continue to have a current parent indefinitely.

Predicate C2 indicates that the timestamp of each process is at most the timestamp of its current parent. This alone does not guarantee loop freedom, since all the process in a loop could have identical timestamps. Loop freedom is guaranteed by the additional property below, which is true even if the unicast routing tables are never stable.

Let pr_path(p) be the set of processes in the path obtained by following the pr variables starting with p.pr.

Property 5

$C2 \wedge \langle \forall\ p : p.pr \neq p : root \in pr_path(p) \rangle$ is a closure

Property 5 states that if a process p chooses a neighbor as its current parent, then there is a path from this neighbor to the root obtained by following the pr variables beginning with p.pr. From C2, this path consists entirely of black edges, that is, each parent and its child in the path are in agreement with each other. Also, from Property 4, once a process has a current parent, it continues to have a current parent throughout the computation. Finally, from Property 3, every member of the group eventually has a current parent, provided the unicast routing tables become stable.

In summary, every group member is guaranteed to have a current parent leading to the root once the unicast routing tables are stable. While the unicast routing tables are changing, any process that has a current parent continues to have a current parent and also has a path to the root. Thus, the group tree adapts to the new unicast spanning tree, and in the process it continues to be loopless and maintains all the group members connected, as desired.

7. Further Refinements

There are several other possible refinements for the group routing protocol. We mention a few of these briefly in this section.

The basic protocol and its refinements assume that the membership of a process in the process group is constant. The protocols in this paper can be easily enhanced to allow a process to join or leave the process group at will.

Another possible refinement is to allow a process to send data messages to the process group, even though the process is not a member of the group. In this case, the process would not receive any data messages addressed to the group, but it would be able to send data messages to all group members. To accomplish this, the message sent by a non-member is routed through the network as if it were a unicast message to the root of the group tree. When the message arrives to a process that is in the group tree, the process forwards the message to all its neighbors in the tree, as if it had originated the message.

It is also possible to modify the group routing protocol to become more fault-tolerant. In particular, the protocol could begin from an arbitrary initial state, and converge to the desired state where the group tree is the required subset of the unicast spanning tree. To achieve fault-tolerance, the unicast routing protocol must be fault-tolerant, and also the communication protocol to exchange messages between neighbors must be fault-tolerant. Fault-tolerant protocols for these tasks are referenced in [10].

Another refinement to increase fault-tolerance is to have multiple choices for a root process. These choices should be linearly ordered. Assume the unicast routing tables in a process indicate that the highest ordered choice cannot be reached. This could occur because the chosen process is down or the faulty links have partitioned the network. If this is the case, the process chooses as a root the first process in the order of choices which is reachable according to its routing tables, and chooses its new parent accordingly.

8. Concluding Remarks

The technique of propagating timestamps has been used previously in unicast routing protocols [1]. The purpose of the timestamp in these protocols is to quickly break routing loops that form in networks whose topology quickly changes, such as mobile networks [15]. In our protocol, we use the technique somewhat differently. The timestamps are used to ensure that the group tree always remains loopless.

There has been some debate on whether a single "core" group tree should be used to multicast data messages to a process group, or multiple group trees should be used, one per source of data messages [7]. Regardless of which of these two approaches is taken, the techniques presented in this paper may be used to ensure that each tree is responsive to the changes in the unicast routing tables without compromising the integrity of the tree.

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 Symposiumm 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, ``Group Routing without Group Routing Tables: An exercise in Protocol Design'', to appear in *Computer Communications,* 1996. Also, to appear in a technical report, University of Houston, 1996 (available from the authors).

[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., ``Protocol Verification Made Simple'', *Computer Networks and ISDN Systems*, Vol. 25, 1993, pp. 969-980.

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

[11] Gouda M., *The Elements of Network Protocols*, textbook in preparation.

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

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

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

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

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

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