Group Routing without Group Routing Tables: An Exercise in Protocol Design

Jorge A. Cobb
University of Houston
Houston, TX 77204-34785
{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 local unicast routing tables of each process. Thus, group routing is accomplished by composing two protocols: an underlying 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 group routing protocol is developed in three steps. First, a simple protocol is obtained, and is proven correct. Then, the protocol is refined twice. Each refined protocol improves upon its predecessor by satisfying all of the predecessor's properties plus some additional stronger properties. 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 destination process group. This dissemination of data messages to a process group has many applications, such as audio and video conferencing [WH92], replicated database updating and querying, and resource discovery [KS92].

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 node in the tree, and hence each group member, receives the data message.

To build a group tree, processes need to learn about the topology of the network and find the best paths between each other. These tasks, however, are typical of unicast routing protocols, which are an integral part of most networks. Thus, we assume that a unicast routing protocol exists in the network, and each process has its local unicast routing table. We take advantage of the unicast routing table in each process and use it as a guide in the construction of an efficient group tree.

Many unicast routing algorithms exist in the literature, e.g., [AGH90, AS92, GS94, KG89, SC87]. 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 (e.g., links being taken in or out of service). To maintain the efficiency of the group tree, when the unicast routing tables change, the tree is restructured to reflect these changes.

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, using each step as a stepping stone to the next. First, we present a basic version of the protocol, and prove some correctness properties for this version. Then, we present two refined versions of the basic protocol. Each refined version improves upon the previous version by satisfying all of the properties of the previous version, and also satisfying additional stronger properties.

The basic group routing protocol builds a tree that adapts itself to the unicast routing tables in a manner that may temporarily disrupt the integrity of the group tree. After the second refinement, 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. 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 [DM78]. In [DC90] [De94], 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 [Bal95] [BFC93], 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 the network topology changes.

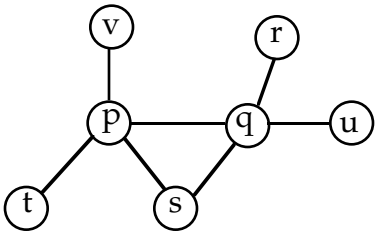


Figure 1: Network of processes

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 to the group routing protocol are mentioned. Concluding remarks are given in Section 8. To simplify the exposition, all proofs are deferred to the appendix.

2. A Protocol Family

Below, 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, as shown in Figure 1. In this graph, a node represents a process, and an edge between processes p and q represents two communication channels, one channel from process p to process q and another channel from process q to process p. We say that processes p and q are neighbors iff they are joined by an edge in the network graph. Each process is assigned a unique identifier, which we assume to be of type integer.

The channel from a process p to a process q is denoted by ch.p.q. A message sent from process p to process q is stored in channel ch.p.q until the message is received by q. When process p sends a message to q, the message is added at the tail of the message sequence of channel ch.p.q. When process q receives a message from p, q receives the message at the head of the message sequence in ch.p.q.

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** j, where msg is a message type and j 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 state-

ments of the form **send** msg **to** j. Similar notations for defining network protocols are discussed in [Gou93] [Gou95].

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** j, and there is a message of type msg at the head of channel ch.j.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** j, and this action is in process p, then, before the action's command is executed, a message of type msg is removed from the head of channel ch.j.p. Any 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 $\rightarrow$ x := j

is actually a shorthand notation for the following three actions, one for each possible value of parameter j.

rcv msg **from** 0 $\rightarrow$ x := 0

[] **rcv** msg **from** 1 $\rightarrow$ x := 1

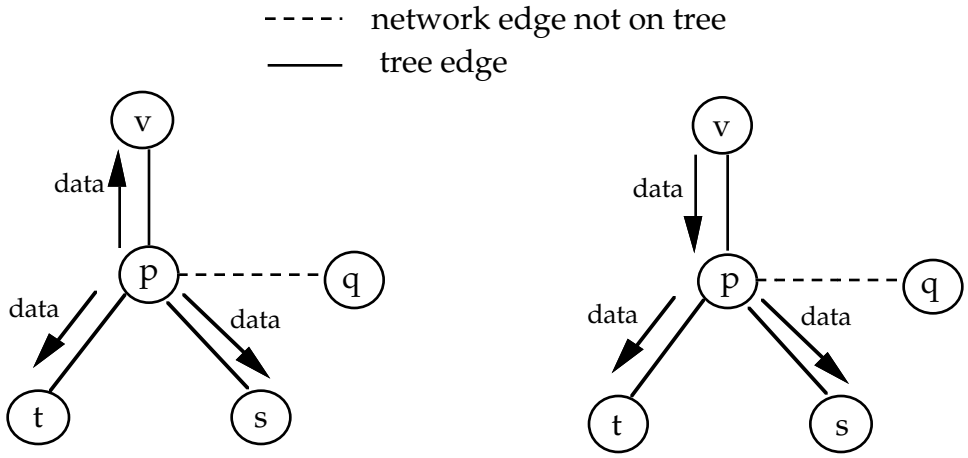
[] **rcv** msg **from** 2 $\rightarrow$ x := 2

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.

Assume there are n members in the group. When a process creates a data message, the process could construct n - 1 copies of this message, and unicast each message to a distinct member of the group, using the regular unicast routing tables of the network. This, however, is wasteful, since it generates more messages than necessary, plus it requires each member to have knowledge of all other members of the group.

We adopt instead the approach of constructing 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 node 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.



a) p is originator of data message b) p is not originator of data message

Figure 2

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 2a. When a node 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, as shown in Figure 2b.

To determine which edges in the network belong to the group tree, we take advantage of the existing spanning trees provided by the unicast routing tables in the network. Recall that the unicast routing table at node p determines, for each possible destination node 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 construct a group tree, we designate one node in the network as the root node of the group tree, and the parent of each node p in the group tree is the next-hop neighbor in the unicast path from p to the designated root node. In this way, the group tree is a subset of the unicast spanning tree with the same root.

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

ent does not receive a request from a child within some timeout period, it removes the process from its set of children.

The unicast spanning tree, although usually stable, may change due to varying network conditions, such as communications channels taken into and out of service, processes being added or removed from the network, variations in channel utilization, etc.. These changes in the network may temporarily cause problems in unicast routing, such as fluctuations in the routing tables at a process, and also unicast routing loops. We assume that these problems are temporary, and that the unicast spanning tree will again become stable.

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, and hence, some data messages may not be delivered to all group members. In Sections 5 and 6 below, we refine our solution to prevent disruption of message delivery due to changes in the unicast routing tables.

We next present in detail the code for each process. The constants and variables in each process are as follows.

Each process has a global constant, root, which is the identifier of the group member chosen as the root of the group tree. Each process also has two local constants, also known as inputs. Input nbr in process 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.

Each process 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 a process p determines that it should not belong to the group tree, the process assigns its own identifier p to pr.

When process p calls the function ROUTE(p, q), it gets in return the next-hop neighbor in the unicast path from p to q. Note that this function call 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 p is the root of the group tree, its variable pr is always equal to p.

Each process p in the network can now be defined as follows.

process p

const

root	:	**integer**	{ root of group tree }

inp

nbr	:	**set of integer**,	{ set of neighboring processes }
mbr	:	**boolean**	{ **true** iff p is a group member }

var

chl	:	**set of integer**,	{ children of p in group tree }
pr	:	**integer**,	{ parent of p in group tree}
wr	:	**set of integer**	{ waiting for replies from these neighbors }

```
par
   j      :   nbr                          { j ranges over each element of nbr }
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. If the message is received from a neighbor that is neither a child nor parent, the message is discarded.

Otherwise, the message is sent to each neighbor that is either a child or parent, except for the neighbor from which the message was received. 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 it should be, p assigns to its parent variable pr the next-hop neighbor to the root, and it sends a request to this parent, 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 sent 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 [Gou95].

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 of 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 this tree.

4. Protocol Properties

In this section, we present two types of properties needed to described the intended behavior of our group routing protocols, namely, closure and convergence [Gou93]. After defining the terms closure and convergence, we show which specific closure and convergence properties are satisfied by the protocol of Section 3.

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.

Many of our state predicates make use of universal quantifications of the form:

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

This quantification 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.

From the above definitions, if S is a closure in **N**, then any computation starting from an S-sate is guaranteed to continue indefinitely and every state encountered in the computation will be an S-state. Thus, predicate S defines a non-empty and closed domain of indefinite execution for protocol **N**. If S is a weak-closure, then any computation starting from an S-state also continues indefinitely and encounters only S-states, except for a finite prefix of the computation that contains some non-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.

$$S0 \equiv \langle \forall\ p, q : : (q \notin p.wr \wedge rqst\#ch.p.q + rply\#ch.q.p = 0) \vee (q \in p.wr \wedge rqst\#ch.p.q + rply\#ch.q.p = 1) \rangle$$

$$S1 \equiv \langle \forall\ p : p.chl = \emptyset \wedge \neg p.mbr : pr = p \rangle$$

$$C0 \equiv S0 \wedge S1$$

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 could be p.pr = p, p.wr = $\emptyset$ for all p, and no request or reply messages in any channel. Predicate C0 satisfies the following property.

Property 0:

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. The set of edges in the group tree[1] is denoted by GT. Edges in GT are directed, that is, edge (p, q) differs from edge (q, p). For each pair of neighboring processes, p and q, we define:

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) $(p, q) \in GE \Leftrightarrow (p, q) \in GT - BE$

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 whose root equals the root of GT. Let path(UT, p) be the edges in UT of the path from p to the root, and sub(UT, p) be the set of nodes of the subtree of UT rooted at p.

Property 1:

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 2:

For any process p,

$C0 \wedge \langle \forall\ r : r \in sub(UT, p) : \neg r.mbr \rangle$

converges to

$C0 \wedge \langle \forall\ r : r \in sub(UT, p) : \neg r.mbr \rangle \wedge \langle \forall\ r, s : r \in sub(UT, p) : (r, s) \notin GT \rangle$

The first property states that if a node 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 nodes in a subtree of UT are not members of the process group, then the entire subtree will be removed from the

[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.

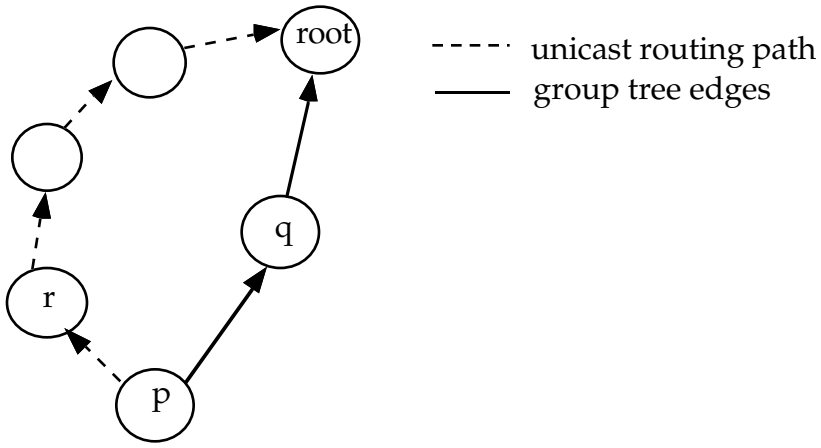


Figure 3: changing parents

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

In this section, we refine the basic protocol of Section 3 by restricting when a process can change from one parent to another. The purpose of this restriction is to ensure that a node that has joined the group tree remains connected to the tree while changes in the unicast routing tables are occurring. Like the basic protocol, the refinement should satisfy the properties presented in the previous section, or it should satisfy properties that are very similar to these.

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 3. It is possible that, after p assigns r to p.pr, process 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, process p will be temporarily disconnected from the group tree.

To prevent being disconnected from the group tree, process p should not change its parent from q to r until r is connected to the group tree. We say that 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.

Each process p in the network can be defined as follows.

```
process p
const
   root  :  integer                    { root of group tree }
inp
   nbr   :  set of integer,             { set of neighboring processes }
   mbr   :  boolean                     { true iff p is a group member }
var
   chl   :  set of integer,             { children of p in group tree }
   pr    :  integer,                    { current parent of p in group tree}
   tpr   :  integer,                    { tentative parent of p in group tree}
   wr    :  set of integer,             { waiting for replies from these neighbors }
   b     :  boolean                     { auxiliary variable}
par
   j     :  nbr                         { j ranges over each element of nbr }
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     →
                      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
end
```

This protocol has six actions. The first two actions are the same as in the basic protocol.

In the third action, 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. The request is sent only if the neighbor has replied the last request sent to it. Similarly, a request is sent to the current parent, also provided no reply is outstanding for this parent.

In the fourth action, 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 fifth action, 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, 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 3:

For all processes p,

$$p.pr \neq p \wedge p.mbr \wedge p \neq root \text{ is a closure}$$

This property is satisfied without making any assumptions about the behavior of the unicast routing tables. Thus, for Property 3, we may assume the unicast routing tables 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, it must be the case that q also has a current parent or q is the root. We express this in predicate C1 below. We require this predicate to hold at the initial state of the system. It is somewhat stronger that C0, because it involves the new variables introduced in the refinement, such as p.tpr and the bit in the rply message, and it also involves the aforementioned requirement on connectivity.

$$S2 \equiv \langle \forall\ p, q : p.pr = q \wedge p \neq q : (p, q) \in BE \wedge (q.pr \neq q\ \vee q = root) \rangle$$

$$S3 \equiv \langle \forall\ p, q : rply(\textbf{true}) \in ch.p.q : q \in p.chl \wedge (p.pr \neq p \vee p = root) \rangle$$

$$S4 \equiv \langle \forall\ p : p.chl = \varnothing \wedge \neg p.mbr : p.tpr = p \rangle$$

$$C1 \equiv C0 \wedge S2 \wedge S3 \wedge S4$$

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

The refinement in this section satisfies Properties 0, 1 and 2 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 eventually converges to the smallest

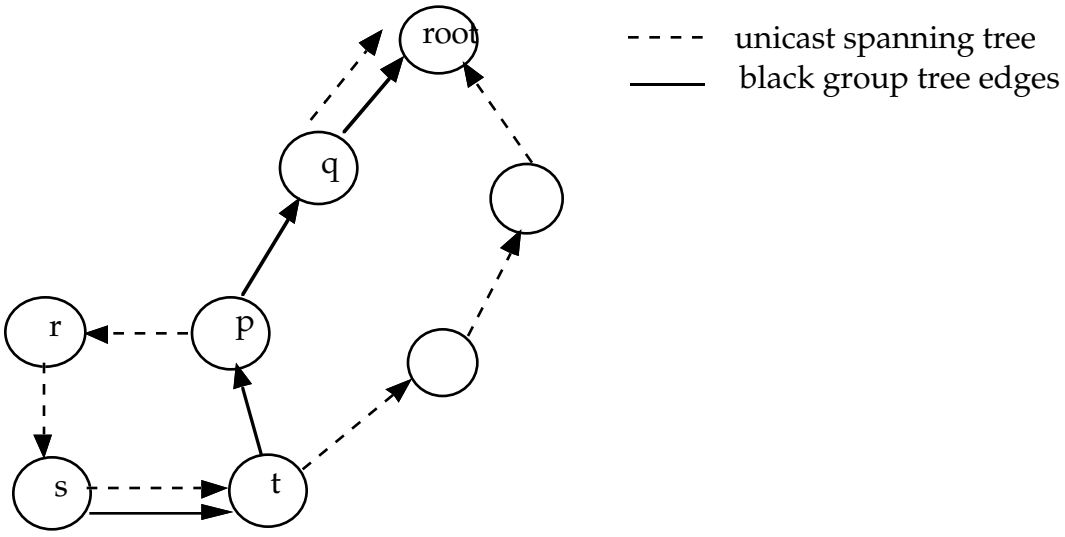


Figure 4: Temporary loops in group tree

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 4. 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, r 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, because s is connected to the tree, process 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 this loop is possible even if the unicast routing tables themselves 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 will not solve the problem. The problem must be solved by further refining the protocol, which is the subject of the next section.

6. Second Refinement: Maintaining Loop-freedom

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 node maintains a timestamp variable ts. The root increments its timestamp periodically. A non-root node may not increment its timestamp on its own. Instead, it receives the value of its parent timestamp in each reply message from its parent. If the received timestamp is larger than the node's own timestamp, the node adopts the value received as its new timestamp.

When the routing tables indicate that the process 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, and waits to receive a reply from the tentative parent with a timestamp larger than its own. When this occurs, and the reply indicates that the tentative parent is connected to the group tree, the process chooses the tentative parent as its current parent.

The reason no loops are created is the following. All processes in the group subtree rooted at p always 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, this indicates to p that the tentative parent is not part of the subtree of process p, and choosing this neighbor as the new current parent cannot introduce a loop.

The changes required for the refinement are as follows. A new integer variable, ts, stores the timestamp of the process. The first three actions and the timeout action of the protocol of the previous section remain unchanged. The actions to receive a request and to receive a reply, plus a new action to increase the timestamp, are given below.

```
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
```

In the first action above, process p increments its timestamp provided it is the root of the tree. When process p receives a request, it returns to its child the current value of its timestamp in the reply message. When process p receives a reply message, it checks the timestamp and the sender of the message. If the sender is the tentative parent, the sender is connected to the tree, and the timestamp is larger than p's timestamp, then p chooses this neighbor as its current parent, and sets its timestamp to the received value.

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.

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

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

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

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

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

The refinement in this section satisfies Properties 0, 1 and 2 of Section 4, with the exception that each occurrence of C0 in these properties is replaced by C2. Hence, C2 is a closure, and the group tree eventually converges to the smallest subgraph of the unicast spanning tree that maintains all the group members connected. Furthermore, the refinement in this section also satisfies Property 3 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 not greater than that 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. This property is satisfied without making any assumption about the unicast routing tables, i.e., they are free to change throughout the computation.

Let pr_path(p) be the set of nodes in the network path obtained by following the pr variables beginning with p.pr.

Property 4:

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

Property 4 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 3, once a process has a current parent, it continues to have a current parent throughout the computation. Finally, from Property 1, 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 become stable. While the unicast routing tables are changing, any process that already 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 refinements that are possible 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. That is, whether a process is a member of the group or not remains constant throughout the computation. The protocols in this paper can be easily enhanced to allow the 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. If the message arrives first to the root, the root forwards the message down the group tree. If the message arrives first to a non-root node p before it reaches the root node, then node p forwards the message to all its children and its parent, as if it were a regular data message originated by p.

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 [Gou95a].

The main adjustment of the protocol to become fault-tolerant is to ensure that the timestamp of the root eventually becomes larger that the timestamp of all other processes in the network. This will eventually happen, since the root increases its timestamp periodically. However, if the root's timestamp is much smaller than that of other processes, it might take a significant amount of time for the root's timestamp to become larger. There are two possible approaches to remedy this.

One approach is for the processes to use additional messages to propagate towards the root the largest timestamp they have learned from their neighbors. In this manner, the root process will learn from its neighbors the value of the largest timestamp in the network. Then, the next timestamp the root generates should be larger than this value.

Another approach is for the root to derive its timestamp from a real-time clock, and have all processes synchronize their clocks using algorithms like those in [Mil91]. Each process periodically checks its timestamp against its real-time clock. If its timestamp is larger than its clock plus the upper bound on the clock's skew, then the process resets its timestamp to the value of the clock. Since no process can have a timestamp significantly larger than the real-time clock, and all clocks are synchronized, the root will quickly have a timestamp larger than that of any other process.

Another refinement to increase fault-tolerance is to have multiple choices for a root node. These choices are ordered. Assume the unicast routing tables in a process indicate that the highest ordered choice cannot be reached. This could occur because the chosen node 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. Summary and Concluding Remarks

We defined a protocol for routing data messages to every member of a process group. This is accomplished by building a tree of processes, where each group member is included in the tree, and forwarding each data message along the entire tree. We assume that a unicast routing protocol exists in the network, which provides each process with its local unicast routing table. The group routing protocol takes advantage of the unicast routing tables as a guide in the construction of an efficient group tree. To maintain the efficiency of the group tree, when the unicast routing tables change, the tree is restructured to reflect these changes. Furthermore, the changes to the tree occur in a controlled manner, preserving the integrity of the tree at all times.

The design of the group routing protocol is based on protocol composition. That is, it is assumed that an underlying unicast routing protocol exists. However, no specific details are assumed about this protocol, other than the basic requirement of converging to a stable and sensible assignment of values to the routing tables.

The design of the group routing protocol is also based on protocol refinement. First, we present a basic version of the protocol, and prove some correctness properties for this version. Then, we present two refined versions of the basic protocol. Each refined version improves upon the previous version by satisfying all of the properties of the previous version, and also satisfying additional stronger properties.

The technique of propagating timestamps has been used previously in unicast routing protocols [AGH90]. 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 [Per94]. 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 [De94]. 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

[AGH90] Arora A., Gouda M., Herman T., ``Composite Routing Protocols'', *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing,* 1990.

[AS92] Alaettinoglu C, Shankar U., ``Stepwise Design of Distance-Vector Algorithms'', *12th Symposiumm on Protocol Specification, Testing and Verification,* 1992.

[Bal95] Ballardie T., ``Core Based Tree Multicast'', Internet RFC, work in progress.

[BFC93] Ballardie T., Francis P., Crowcroft J, ``Core Based Trees: An Architecture for Scalable Inter-Domain Multicast Routing'', *ACM SIGCOMM Conference*, 1993.

[DC90] Deering S., Cheriton D., ``Multicast Routing in Datagram Networks and Extended LANs'', *ACM Transactions on Computer Systems,* Vol 8., No 2., May 1990.

[De94] Deering S. et. al., ``An Architecture for Wide-Area Multicast Routing'', *ACM SIGCOMM Conference,* 1994.

[DM78] Dalal, Y. K., Metcalfe, R. M., ``Reverse Path Forwarding of Broadcast Packets'', *Communications of the ACM,* Vol. 21, No. 12, Dec. 1978.

[Gou93] Gouda M., ``Protocol Verification Made Simple'', *Computer Networks and ISDN Systems*, Vol. 25, 1993, pp. 969-980.

[Gou95] Gouda M., *The Elements of Network Protocols*, textbook in preparation.

[Gou95a] Gouda M., ``The Triumph and Tribulation of System Stabilization'', *International Workshop on Distributed Algorithms,* 1995.

[GS94] Gouda M., Schneider M., ``Maximum Flow Routing'', *Joint Conference on Information Sciences,* 1994.

[KG89] Cheng C., Riley R., Kumar S, Garcia-Luna-Aceves J., ``A Loop-free Bellman-Ford Routing Protocol without Bouncing Effect'', *ACM SIGCOMM Conference*, 1989.

[KS92] Kahle B., Schwartz M., Emtage A., Neuman B., ``A Comparison of Internet Resource Discovery Approaches'', *Computing Systems,* Vol. 5 No. 4., Fall 1992.

[Mil91] Mills D., ``Internet Time Synchronization: The Network Time Protocol'', *IEEE Transactions on Communications*, Vol. 39, No. 10, Oct 1991, p. 1482.

[Per94] Perkins, C. et. al., ``Ad Hoc Networking in Mobile Computing'', *ACM SIGCOMM Conference*, 1994.

[PWD88] Partridge C., Waitzman D., Deering S., ``Distance Vector Multicast Routing Protocol'', Internet Request for Comments, RFC 1075.

[RF89] Rajagopalan B., Faiman M., ``A New Responsive Distributed Shortest Path Routing Algorith'', *ACM SIGCOMM Conference*, 1989.

[SC87] Shin K. G., Chen M., ``Performance Analysis of Distributed Routing Strategies Free of Ping-Pong-Type Looping'', *IEEE Transactions on Computers,* 1987.

[WH92] Wilbur S., Handley M., ``Multimedia Coferencing: from Prototype to National Pilot'', *INET' 92 International Networking Conference.*

Appendix

Proofs of Properties

Notation: For a quantification of the form

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

R is known as the *range* of the quantification, and T is know as the *body* of the quantification.

1. Basic Protocol

1.1. Property 0

We are required to show that if C0 holds before the execution of any action in the protocol, then it holds after the action's execution.

The first two actions do not alter any variables mentioned in C0. Thus, we concentrate in the remaining four actions.

The third action sends a request to neighbor pr and adds pr to wr, but only if it is not currently a member of wr. Thus, the action cannot invalidate S0. The action cannot invalidate S1, since it only changes pr provided $chl \neq \emptyset \vee mbr$.

The fourth action affects only S0. If a request is received from j then a reply is sent to j. Hence, if the second disjunct of S0 is true before execution, it remains true after execution of the action. Similarly, the fifth action affects only S0. If it executes, the second disjunct of S0 is true before execution, and the first disjunct is true after execution.

The sixth action affects only S1. If the timeout removes a child from the set chl, then pr is assigned p, provided chl = $\emptyset \wedge \neg$mbr holds. Thus S1 remains true after executing the action.

Since all actions preserve the truthfulness of C0, then C0 is a closure.

1.2. Property 1

Since we assume the unicast routing tables will eventually achieve a stable value, we begin our computations from a state in which these tables have already stabilized.

First note that if the unicast routing tables are stable, and p.pr = ROUTE(p, root), then p.pr will no longer change value. Furthermore, if the edge (p, p.pr) is black, it continues to be black, because p.pr does not change value, and the timeout of neighbor p.pr is not enabled.

Let the sequence of processes in path(UT, p) be p, r1, ... , rn, root. Edge (p, r1) becomes black as follows. If there is a request in ch.p.r1 or a reply in ch.r1.p, then, by fairness during execution, r1 eventually receives the request, sends back a reply, and p receives the reply, removing r1 from p.wr. Again, by fairness during execution, and p.mbr = true, the third action in p is eventually executed, assigning r1 to p.pr, and a request is sent to r1. Note that, as argued above, p.pr will no longer change value. When r1 receives the request, it adds p to its child set, making the edge black. Again, as argued above, this edge will continuously remain black.

A simple inductive argument along the path p, r1, ... , rn, using the above as a base, shows that all the edges in the path become black and remain black.

1.3. Property 2

We again begin our computations from a state in which the unicast routing tables have achieved a stable value.

Consider any network edge (s, t), where (s, t) $\notin$ UT. If this edge is in GT, then it is eventually removed from GT as follows.

If there is a request from s to t, then by fairness t receives it and returns a reply to s. If there is a reply from t to s, s eventually receives it, and removes t from s.wr. If the third action of s executes, then s.pr = ROUTE(s, root), and since (s, t) $\notin$ UT, s.pr $\neq$ t. Hence, s.pr never again equals t, and never sends a new request to t. If the third action of s cannot execute, then, by closure C0, s.pr = p. If s.pr ever changes again, its new value will be ROUTE(s, root), which is different from t. Thus, eventually t times-out and removes s from its child set. Since s never again sends a request to t, edge (s, t) will never rejoin the tree.

Thus, there is a point in the computation where GT is a subset of UT. Consider now the subtree sub(UT, p), and an edge (r, s) where r is a leaf in the subtree. Since r is a leaf, it has no children, and since it is in sub(UT, p), r.mbr is false. Hence, the third action in r never executes, and by closure C0, r.pr = r remains true forever. Furthermore, any request from r to s is eventually received, and any reply from s to r is also eventually received. Also, since r.pr = r holds indefinitely, no request is

ever sent again by r. Thus, s times-out and removes r from its child set, permanently removing edge (r, s) from GT.

A simple inductive argument on the height of the tree using the above as a base shows that all edges in sub(UT, p) are removed permanently from GT.

2. First Refinement

Recall that Properties 0, 1 and 2 in the first refinement are identical to the respective properties of the basic protocol, except that C0 is replaced by C1.

2.1. Property 0

It can be proven that C0 is a closure of this protocol in a very similar way to the proof of Property 0 in the basic protocol, which we do not repeat here. Thus, we assume C0 holds at every state of the computation, and concentrate only on S2 through S4.

For each of the three predicates, S2, S3, and S4, we assume they are true before execution of each action, and show they are true after executing the action. Each of these predicates can be falsified if the range changes to true while leaving the body false, or if the body changes to false while leaving the range true.

The range of S2 can be made true in the fifth action by changing p.pr. If p.tpr is assigned to p.pr it implies that the bit received in the reply is true, which by S3 (instantiating p with p.tpr and q with p) implies that neighbor p.tpr considers p to be its child (i.e. edge (p, p.tpr) is black after the assignment) and the neighbor is the root or the neighbor has a current parent. This preserves S2. On the other hand, the body of S2 can be falsified by removing p from q.chl or assigning q to q.pr. Neither of these is possible while p.pr = q, because the timeout in q is not enabled. Thus, S2 is preserved by action execution.

The range of S3 is made true by sending a reply to a neighbor. The code in the fourth action ensures that if the bit in the message is true, then the body is true. The body can be made false by removing q from p.chl or setting p.pr to p. This only happens in the timeout action, which is not enabled if a reply to q has not been received. Thus, S3 is preserved by action execution.

Finally, the range of S4 can be made true by removing an element of p.chl in the timeout action. The timeout though will set p.tpr to p, thus preserving S4. The body of S4 is made false by assigning a value other than p to p.tpr. This can only happen in action 3, whose guard ensures the range is false. Thus, S4 is preserved by action execution.

Since S2 through S4 are preserved through action execution, and C0 is a closure, then C1 is a closure.

2.2. Property 1

Due to Property 0, we assume predicate C1 holds in all states of the computation.

Since for this property we assume that the routing tables achieve a stable state, we begin our computation after the tables become stable.

Note that the first refinement treats variable tpr in a very similar way to the way variable pr is treated in the first protocol, with the exception that the timeout is stronger since it refers to both variables. Thus, a very similar argument to that of Property 1 of the basic protocol shows that, eventually, any edge (r, s) in path(UT, p) has $r \in s.chl$ and r.tpr = s. Furthermore, the argument shows that this remain continuously true.

We next need to show that after the above holds, for any edge (r, s) in path(UT, p), r.pr is assigned s. This is done by induction on the path p, r1, ... , rn, root, with the edge (rn, root) as the base. For the base case, the next request from rn to root adds rn to root.chl. The next reply that the root sends to rn contains the bit true, because the root always returns replies with true bits. When this is received by rn, rn assigns root to rn.pr, because rn.tpr = root. Since rn.tpr no longer changes values, rn.pr will continuously have the value root. The inductive case is similar. Thus, all edges in path(UT, p) become black and remain black continuously.

2.3. Property 2

Due to Property 0, we assume predicate C1 holds in all states of the computation.

Since for this property we assume that the routing tables achieve a stable state, we begin our computation after the tables become stable.

What we need to show is that eventually, for all nodes r in sub(UT, p), each of r.tpr and r.pr are equal to either ROUTE(r, root) or r. Furthermore, this should continue to be true for the remainder of the computation. If this is shown, then an inductive argument on the subtree sub(UT, p) similar to the one used for Property 2 of the basic protocol can be applied.

Assume first that $r.chl = \emptyset$ and $\wedge \neg r.mbr$. Because C1 holds, r.tpr = r and r.pr = r. Furthermore, if eventually $r.chl = \emptyset$ and $\wedge \neg r.mbr$ does not hold, when action three is executed, r.tpr is assigned ROUTE(r, root). Since the routing table is constant, r.tpr can only have this value or r from this point on. Similarly, since r.pr receives its value from r.tpr, from this point on, r.pr can only be assigned ROUTE(r, root) or be assigned r.

Assume now that $r.chl \neq \emptyset \vee r.mbr$. When action three is executed, r.tpr is assigned ROUTE(r, root), and since the routing tables are constant, r.tpr remains constant as long as $r.chl \neq \emptyset \vee r.mbr$. Regarding r.pr, using an argument similar to that of Property 1, if $r.chl \neq \emptyset \vee r.mbr$ remained constant, then all the edges from r to the root become black and remain black, i.e., eventually r.pr = ROUTE(r, root) and does not change value. On the other hand if $r.chl \neq \emptyset \vee r.mbr$ fails to hold along the computation, then we have the case of the previous paragraph.

2.4. Property 3

The predicate has only one variable, p.pr, and only two actions in the code of process p affect this variable: the fifth action and the sixth action. In the fifth action, p.pr is set to the identifier of a neighbor, and thus p.pr ≠ p is maintained. In the sixth action, since p.mbr is true, p.pr is not updated. Thus, no action can falsify the predicate, and it is thus a closure.

3. Second Refinement

Recall that Properties 0, 1 and 2 in the first refinement are identical to the respective properties of the basic protocol, except that C0 is replaced by C2.

3.1. Property 0

Predicate C2 is very similar to C1 except that it has the additional restrictions on the values of the timestamps. Thus we concentrate only on satisfying the constraints involving the timestamps in each of S5, S6, and S7. The remaining part of the proof is very similar to Property 0 in the first refinement.

For S5, the range could become true in action 5, by assigning p.tpr to p.pr. This happens if you receive a message from p.tpr, which the guard of the **if** ensures has a timestamp grater than p.ts. From S6, the value of the message's timestamp is no grater than the sender's timestamp, and thus the timestamp of neighbor p.tpr is at least p.pr, as required in the body. The body of S5 could become false if p increases its timestamp beyond that of its p.pr neighbor. The timestamp increases only when a reply is received, and the statement of the action, in combination with S6, ensures that after execution, p.ts is no grater than the timestamp of neighbor p.pr. Thus, S5 is preserved by action execution.

For S6, the range becomes true when a reply is sent by p after receiving a request. Since the timestamp in the message is that of p, the body is true. The body could become false by decreasing p.ts, but no action can decrease p.ts, so this is moot. Hence, S6 is preserved by action execution.

For S7, note that no process, including the root, ever decreases its timestamp, so S7 cannot be falsified by decreasing root.ts. Furthermore, any change to p.ts comes from the timestamp of a reply message, which from S6, is at most the timestamp of the sender, which from S7 is at most root.ts. Thus, p.ts is never assigned a value larger than root.ts.

Hence, C2 is a closure.

3.2. Property 1

Due to Property 0, we assume predicate C1 holds in all states of the computation.

Since for this property we assume that the routing tables achieve a stable state, we begin our computation after the tables become stable.

The proof is similar to that of Property 1 of the first refinement. That is, it can be shown in a similar way that, eventually, any edge (r, s) in path(UT, p) has r ∈

s.chl and r.tpr = s. Furthermore, the argument shows that this remain continuously true.

We next need to show that after the above holds, for any edge (r, s) in path(UT, p), r.pr is assigned s. This is done by induction on the path p, r1, ... , rn, root, with the edge (rn, root) as the base. For the base case, the next request from rn to root adds rn to root.chl. Note that rn is never removed from root.chl because the timeout of the root is not enabled since rn.tpr = root. Next, either root.ts > rn.ts, or this will hold when root increases its timestamp. Thus, the next reply that the root sends to rn contains the bit true and a timestamp greater than rn.ts. When this is received by rn, rn assigns root to rn.pr. Since rn.tpr no longer changes values, rn.pr will continuously have the value root.

The inductive case assumes that all nodes ri, ..., root have a timestamp larger than nodes p, r1, ..., r(i-1). Showing that edge (r(i-1), ri) becomes and remains black is similar to the base case.

Thus, all edges in path(UT, p) become black and remain black continuously.

3.3. Property 2

The proof is similar to that of Property 2 in the first refinement and is not repeated here.

3.4. Property 3

The proof is similar to that of Property 3 in the first refinement and is not repeated here.

3.5. Property 4

From Property 0 we know that C2 is a closure, so we assume C2 holds in all states of the computation.

Let us denote by P the quantification in Property 4. This quantification could become false if the range becomes true and the body becomes false. The range becomes true when a reply is received and p.pr is assigned p.tpr. From C2, neighbor p.tpr has a current parent, and thus, assuming P holds before the execution of the action, root $\in$ pr_path(p.tpr). Furthermore, also from C2, neighbor p.tpr cannot be a descendant of p in the group tree, since its timestamp is larger than p.ts. Hence, assigning p.tpr to p.pr maintains root $\in$ pr_path(p).

Predicate P could also becomes false by making the body false while leaving the range true. The body becomes false by removing root from pr_path(p). This can only be done if some node r in pr_path(p) changes its r.pr variable. The argument of the previous paragraph shows that if after the change, r.pr $\neq$ r, then root $\in$ pr_path(r), so the root cannot be removed from pr_path(p) in this case. On the other hand, if r assigns r to r.pr, it can only be done in a timeout. Note that r.chl cannot be empty because of the following. Let s be the previous node to r in pr_path(p). Hence, s.pr = r, and from C2, s $\in$ r.chl. Furthermore, r cannot remove s from r.chl because the timeout for s is not enabled (because s.pr = r). Thus, r.chl

can never be empty, and no execution of any of the timeout actions in r can set r.pr to r.

Thus, Property 4 holds.