The Request-Reply Family of Group Routing Protocols

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 family of group routing protocols for a network of processes. The task of these protocols 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. No processing or storage overhead is required for processes not included in the tree. The overhead of processes in the tree consists solely of the periodic exchange of request/reply messages with their parent. To choose the processes that constitute the tree, we take advantage of the existing unicast routing protocol in the network. In addition, our family of group routing protocols distinguishes itself from other group routing protocols in three ways. First, the protocols are proven correct. Second, the protocols preserve the integrity of the group tree as it adapts to changes in the unicast routing tables, even in the presence of temporary unicast routing loops. Third, data messages are propagated along the entire group tree, even while the tree adapts to changes in the unicast routing tables.

1. Introduction

In this paper, we present a family of group routing protocols for a network of processes. The network consists of a set of processes interconnected by point-to-point communication channels. 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 [18], 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 discussed at the end of the paper.

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 existing 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.

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

Many unicast routing algorithms exist in the literature, e.g., [1, 2, 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 make our group routing protocols suitable for operating in conjunction with any unicast routing protocol, the following is the only assumption we make about unicast routing. The unicast routing tables may temporarily fluctuate, but they eventually converge to a value that, for each pair of processes p and q, defines a path from p to q.

To maintain the routing efficiency of the group tree, when the unicast routing tables change, the tree is restructured to reflect these changes. For example, consider a group tree for the multicast distribution of a video image. If the topology of the

network changes, and a network path with greater bandwidth is created between the source of the video and its destinations, then the group tree changes and includes the new path. This allows the source of the video to send an image of greater resolution and quality due to the increase in available bandwidth.

In addition, our protocols have the nice property that they maintain the integrity of the group tree while the unicast routing tables are changing. That is, they do not introduce temporary loops, they always maintain the tree connected, and they never remove a group member from the tree. In the above example, the group tree would adapt itself to the best network path without interrupting the flow of video to the members of the multicast group.

The related work in the area of group routing is as follows. The notion of 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 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.

Our family of group routing protocols distinguishes itself from the above protocols in three ways. First, our protocols are proven correct. Second, our protocols always preserve the integrity of the group tree as it adapts to changes in the unicast routing tables, even in the presence of temporary unicast routing loops. Third, data messages are propagated throughout the entire group tree (i.e., no message loss) even while the group tree is adapting to changes in the unicast routing tables.

We present our family of group routing protocols in four steps. First, we present a basic version of the protocol, and prove its correctness. Then, we refine the protocol three times. Each refined version improves upon the previous version by satisfying all of the correctness properties of the previous version, plus some additional stronger properties.

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 Sections 4 through 7, several refinements to the basic

protocol along with the correctness properties of each refinement are presented. In Section 8, possible further refinements are discussed. Concluding remarks are given in Section 9. Proofs for the properties of each protocol are given in the appendix.

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, each 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. We assume that channels may lose but not duplicate messages.

Each process is assigned a unique identifier, which we assume to be of type integer. The reserved integer **nil** may not be used as a process id. 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 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 $\rightarrow$ x := j

is a shorthand notation for the following three actions.

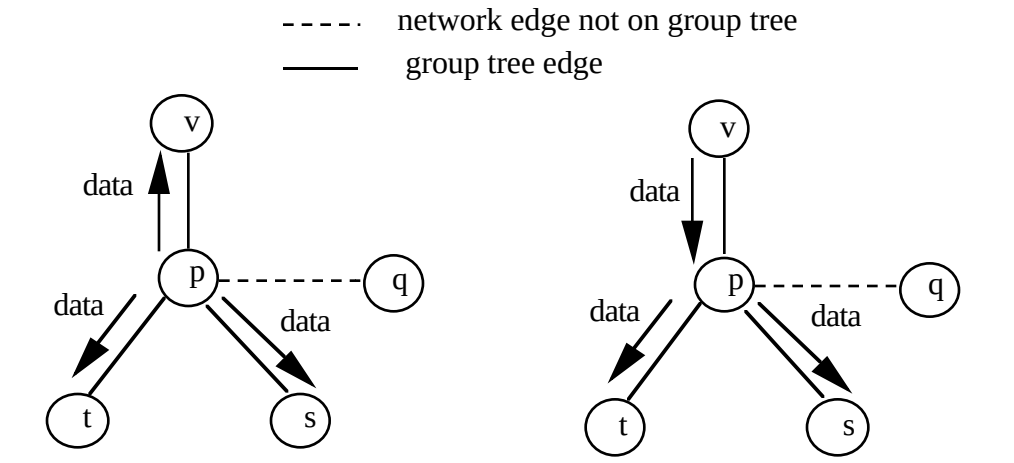
rcv msg **from** 0 $\rightarrow$ x := 0
[] **rcv** msg **from** 1 $\rightarrow$ x := 1
[] **rcv** msg **from** 2 $\rightarrow$ x := 2

3. 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 addressed to the process group, and each of these data messages 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 data 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 addressed to the group is forwarded along the entire group tree. In this way, each process in the tree, and hence each group member, receives each of these data messages. 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



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

Figure 1

message to all of its neighbors in the tree except the one from which the message was received.

It is also possible for processes outside the group to generate data messages addressed to the group. If a process outside of the group tree receives or generates a data message addressed to the group, it forwards the message as if it were addressed to the root of the group tree. In this way, the message will arrive to a process in the group tree (either the root or another tree process encountered in the unicast path towards the root), and this process will propagate the message along the tree as if it were the originator of the message.

To determine which edges in the network belong to the group tree, we take advantage of the spanning trees provided 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 6 and 7, 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 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 **nil** 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 **nil**. Hence, if root = p, then p.pr is always equal to **nil**.

Finally, process p stores in variable wr the process id of the neighbor from which it expects a reply. Variable wr equals **nil** if no reply is expected.

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    :  integer           (* waiting for reply from this neighbor *)

par
  j     :  nbr               (*  j ranges over all neighbors *)

begin

  true →
        if chl ≠ ∅ ∨ mbr          →   for each d in (chl ∪ {pr}) - {nil} do
                                          send data to d
                                      rof
        []¬(chl ≠ ∅ ∨ mbr)        →   if ROUTE(p, root) = nil  →  skip
                                      [] ROUTE(p, root) ≠ nil  →
                                              send data to ROUTE(p, root)
                                      fi
        fi
```

```
▯ rcv data from j     →
        if chl ≠ ∅ ∨ mbr      →    for each d in (chl ∪ {pr}) - {j, nil} do
                                       send data to d
                                   rof;
                                   if  mbr     →    deliver data
                                   [] ¬mbr     →    skip
                                   fi
        []¬(chl ≠ ∅ ∨ mbr)    →    if ROUTE(p, root) = nil  →  skip
                                   [] ROUTE(p, root) ≠ nil  →
                                          send data to ROUTE(p, root)
                                   fi
        fi

▯ (chl ≠ ∅ ∨ mbr) ∧ wr = nil     →
        pr := ROUTE(p, root);
        wr := pr;
        if pr ≠ nil     →  send rqst to pr
        [] pr = nil     →  skip
        fi

▯ rcv rqst from j     →
        chl := chl ∪ {j};
        send rply to j

▯ rcv rply from j     →
        wr := nil

▯ timeout  j ∈ chl ∧ j.pr ≠ p     →
        chl := chl - {j};
        if  chl = ∅ ∧ ¬mbr    →  pr := nil
        [] chl ≠ ∅ ∨ mbr      →  skip
        fi

▯ timeout wr ≠ nil ∧ (rqst#ch.p.pr + rply#ch.pr.p) = 0     →
        wr := nil
end
```

Process p has seven actions. In the first action, the process creates a data message and forwards the message towards the root if the process is not on the group tree, and along the tree if it belongs to the group tree. In the second action, process p receives a data message from one of its neighbors, and forwards the message to

either the root or to its 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 sixth 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 this 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 **nil** to pr.

The final action is a timeout action to retransmit a request when either the request or the reply was lost by a channel.

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 basic 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 basic protocol. 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.

$$C0 \equiv S0 \wedge S1$$

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

$$S1 \equiv \langle \forall\ p : p.chl = \varnothing \wedge \neg p.mbr : p.pr = \textbf{nil} \rangle$$

Predicate C0 states that variable wr in each process indicates the neighbor from which a reply is expected (or message loss occurred). 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 = **nil**, p.wr = **nil**, and no request or reply messages in any channel.

The basic protocol satisfies the following property.

Basic Safety Property:

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 property, we define the following boolean expressions.

a) in(p, q) ≡ p.pr = q ∧ p ∈ q.chl

b) out(p, q) ≡ p.pr = **nil** ∧ p ∉ q.chl

c) partial(p, q) ≡ (p.pr = q ∧ p ∉ q.chl) ∨ (p.pr ≠ q ∧ p ∈ q.chl)

Thus, edge (p, q) is in the group tree (i.e., in(p, q) = **true**) iff p considers q to be its parent and q considers p to be its child. Edge (p, q) is not in the group tree (i.e., out(p, q)) if neither p considers q to be its parent nor q considers p to be its child. We say that (p, q) is partially in the tree if neither in(p, q) nor out(p, q) are true. If during the execution of the protocol an edge is only partially in the group tree, then this is only 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, or when the membership in the process group changes.

For the following property, 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 utree(p, q) = **true** iff edge (p, q) is in the unicast spanning tree. Also, let utdesc(r, p) = **true** iff process r is a descendant of process p in the unicast spanning tree.

Tree Convergence Property:

a) C0

converges to

$C0 \wedge \langle \forall\ p, q : : \neg utree(p, q) \Rightarrow out(p, q) \rangle$

b) C0

converges to

$C0 \wedge \langle \forall\ p, q : : utree(p, q) \wedge \langle \exists\ r : utdesc(r, p) : r.mbr \rangle \Rightarrow\ in(p, q)\ \rangle$

c) C0

converges to

$C0 \wedge \langle \forall\ p, q : : utree(p, q) \wedge \neg\langle \exists\ r : utdesc(r, p) : r.mbr \rangle \Rightarrow out(p, q)\ \rangle$

♦

Part (a) states that all edges not in the unicast spanning tree are permanently removed from the group tree. Part (b) states that each edge in the unicast tree that has a group member as a descendant in the unicast spanning tree is added permanently to the group tree. Part (c) states that each edge in the unicast spanning tree with no group members as descendants in the unicast spanning tree is permanently removed from the group tree. These properties combined imply that the group tree converges to the smallest subset of the unicast spanning tree required to connect all the group members together, as desired.

5. Cumulative Waiting Protocol

We next present a simple refinement to the basic protocol, as a stepping stone to the refinement in the next section.

In the basic protocol, a process sends a request to its parent and expects a reply. In the event that the unicast routing tables change, the process does not send a request to the new parent until a reply is received from the old one (or the reply is lost). In the next section, we are required to send requests to more than one parent. Thus, we change the type of variable wr to a set of process id's, and store in wr the process id's of those neighbors to whom a request has been sent but no reply has been received.

The actions requiring changes are as follows.

```
(chl ≠ ∅ ∨ mbr) ∧ ROUTE(p, root) ∉ wr   →
        pr := ROUTE(p, root);
        if pr ≠ nil     → send rqst to pr;
                          wr := wr ∪ {pr}
        [] pr = nil → skip
        fi

[] rcv rply from j      →
        wr := wr - {j}

[] timeout  j ∈ wr ∧ (rqst#ch.p.j + rply#ch.j.p) = 0     →
        wr := wr - {j}
```

6. Dual-Parent Protocol

We continue refining the basic protocol by restricting when a process changes its parent. The purpose of this restriction is to ensure that a process that has already joined the group tree remains connected to the tree while changes in the unicast routing tables are occurring.

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 one of the following two conditions hold: a) r is the root, b) edge (r, r.pr) is in the group tree (i.e., in(r, r.pr)) and r.pr is also connected to the tree.

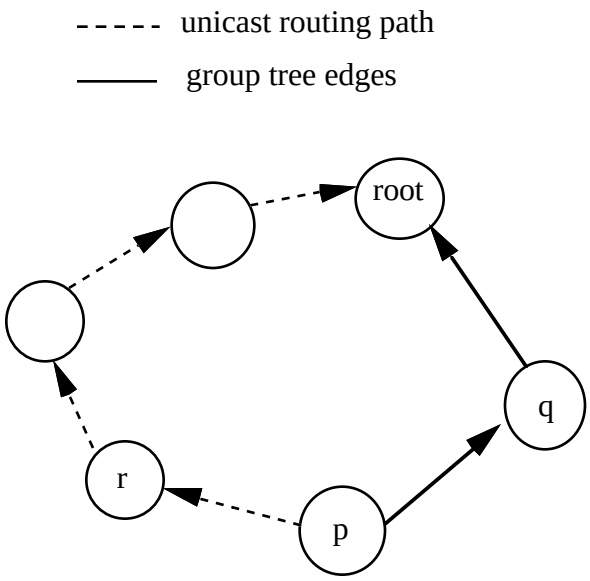


Figure 2: changing parents

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 remains 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 = **nil**. 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 code of the dual-parent protocol.

```
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 *)
  tpr   :   integer,          (* tentative parent of p *)
  wr    :   set of integer    (* expecting reply from these neighbors *)

par
  j     :   nbr               (* j ranges over all neighbors *)

begin

  true →
        if chl ≠ ∅ ∨ mbr  →      for each d in (chl ∪ {pr}) - {nil} do
                                     send data to d
                                 rof
        []¬(chl ≠ ∅ ∨ mbr)  →    if ROUTE(p, root) = nil     → skip
                                 [] ROUTE(p, root) ≠ nil     →
                                        send data to ROUTE(p, root)
                                 fi
        fi

[] rcv data from j  →
        if chl ≠ ∅ ∨ mbr  →      for each d in (chl ∪ {pr}) - {j, nil} do
                                     send data to d
                                 rof;
                                 if  mbr    →    deliver data
                                 [] ¬mbr    →    skip
                                 fi
        []¬(chl ≠ ∅ ∨ mbr)  →    if ROUTE(p, root) = nil → skip
                                 [] ROUTE(p, root) ≠ nil →
                                        send data to ROUTE(p, root)
                                 fi
        fi

[] (chl ≠ ∅ ∨ mbr) ∧ ROUTE(p, root) ∉ wr ∧ ROUTE(p, root) ≠ nil  →
        tpr := ROUTE(p, root);
        send rqst to tpr;
        wr := wr ∪ {tpr}

[] (chl ≠ ∅ ∨ mbr) ∧ pr ∉ wr ∧ pr ≠ nil      →
        send rqst to pr;
        wr := wr ∪ {pr}
```

▯ **rcv** rqst **from** j $\rightarrow$
chl := chl $\cup$ {j};
b := (pr $\neq$ **nil** $\vee$ p = root);
send rply(b) **to** j

▯ **rcv** rply(b) **from** j $\rightarrow$
wr := wr - {j}
if j = tpr $\wedge$ b $\rightarrow$ pr := tpr
▯ $\neg$(j = tpr $\wedge$ b) $\rightarrow$ **skip**
fi

▯ **timeout** j $\in$ chl $\wedge$ j.pr $\neq$ p $\wedge$ j.tpr $\neq$ p $\wedge$ rply#ch.p.j = 0 $\rightarrow$
chl := chl - {j};
if chl = $\emptyset$ $\wedge$ $\neg$mbr $\rightarrow$ pr := **nil**; tpr := **nil**
▯ chl $\neq$ $\emptyset$ $\vee$ mbr $\rightarrow$ **skip**
fi

▯ **timeout** j $\in$ wr $\wedge$ (rqst#ch.p.j + rply#ch.j.p) = 0 $\rightarrow$
wr := wr - {j}

end

In the third and fourth actions 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 fifth 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 $\neq$ **nil**, or if p is the root. In the sixth 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 eighth 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 **nil** 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 **nil**. 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.

The last action is needed to cope with the loss of a request or a reply msg.

We next present the properties of the dual-parent 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.

Continuous Parent Property:

For all processes p,

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

♦

This property 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.

Given that variable wr is now a set, and a new parent tpr has been introduced, we modify predicate C0 (see Section 4) for the dual-parent protocol as follows.

$C0' \equiv S0' \wedge S1'$

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

$S1' \equiv \langle \forall\ p : p.chl = \emptyset \wedge \neg p.mbr : p.pr = \textbf{nil} \wedge p.tpr = \textbf{nil} \rangle$

The protocol in this section also satisfies both properties of the basic protocol, with the exception of replacing C0 by C0′. Hence, the group tree converges to the smallest subgraph of the unicast spanning tree that maintains all the group members connected. Furthermore, we have the additional property below.

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 than C0´, because it involves the bit in the rply message and also the above requirement on connectivity.

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

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

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

Connectivity Property:

C1 is a closure.

♦

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

The above property states that if a process p has a current parent, then the edge from p to its parent is in the group tree, and the current parent of p also has a current parent or is the root. Hence, either the path obtained by following the parent variables starting from p ends in the root, or this path contains a loop and leads back to p. 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. Then, since s is connected to the tree, r chooses s as its current parent, adding the edge (r, s) to the group tree. Subsequently, edge (p, r) is also added to the group tree, 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.

7. Propagated Timestamp Protocol

We next present the last protocol in our family of group routing protocols. The purpose of this protocol 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 protocols of the previous sections.

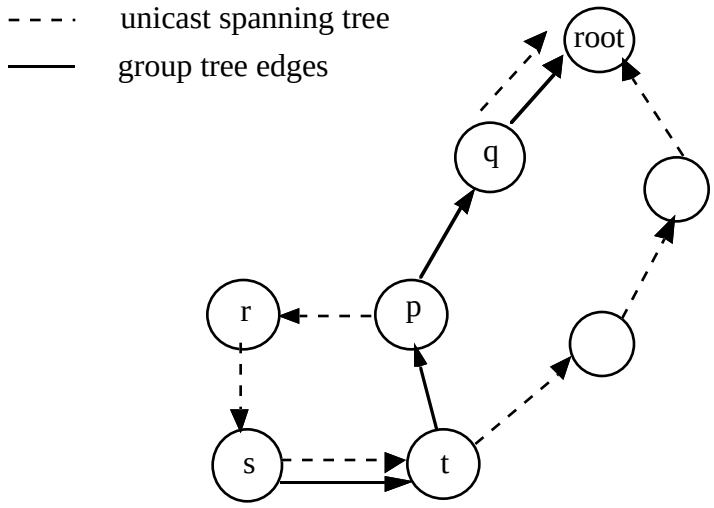


Figure 3: Temporary loops in group tree

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

$p = root \wedge chl \neq \emptyset \;\rightarrow$
    ts := ts + 1

[] **rcv** rqst **from** j $\rightarrow$
    chl := chl $\cup$ {j};
    b := (pr $\neq$ **nil** $\vee$ p = root);
    send rply(b, ts) **to** j

[] **rcv** rply(b, t) **from** j $\rightarrow$
    wr := wr - {j}
    if j = tpr $\wedge$ b $\wedge$ t > ts $\rightarrow$ pr, ts := tpr, t
    [] $\neg$(j = tpr $\wedge$ b $\wedge$ t > ts) $\rightarrow$ **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.

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

$S4 \equiv \langle \forall\ p, q : p.pr = q :\ in(p, q) \wedge (q.pr \neq \textbf{nil} \vee q = root) \wedge q.ts \geq p.ts \rangle$

$S5 \equiv \langle \forall\ p, q, t : rply(\textbf{true}, t) \in ch.p.q :$
$\qquad q \in p.chl \wedge (p.pr \neq \textbf{nil} \vee p = root) \wedge p.ts \geq t \rangle$

$S6 \equiv \langle \forall\ p : : p.ts \leq root.ts \rangle$

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

The protocol in this section satisfies all the properties of the basic protocol (with C0′ replacing C0) and also all the properties of the dual-parent protocol. Thus, the group tree converges to the smallest subgraph of the unicast spanning tree that

maintains all the group members connected. Also, a group member that has a current parent will continue to have a current parent indefinitely.

We also have the additional property.

Decreasing Timestamp Property:

C2 is a closure

♦

The above property implies that the timestamp of each process is always 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.

Loopless Property:

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

This property 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, each edge in this path is in the group tree. Also, from the continuous parent property, once a process in the process group has a current parent, it continues to have a current parent throughout the computation. Finally, from the tree convergence property, 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.

8. Further Refinements

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

8.1 Dynamic Group Membership

Throughout the paper, we have assumed that the group membership remains constant, that is, no process is added to or removed from the group. To include changes in the group membership, we only need to add one more action to each process. Namely, an action that will set tpr and pr to **nil** in the event that the process decides to remove itself from the group when its child set is empty. This will ensure that the basic safety property is not violated.

Regarding the tree convergence property, parts (a) and (b) also hold in the case of changes to the process group, i.e., edges not in the multicast tree are removed from the group tree, and each member of the group (that does not remove itself from the group) will have a path in the group tree to the root. Part (c), i.e., removing unnecessary unicast tree edges from the group tree, will hold once the group membership becomes stable.

All other properties remain true even while the membership in the group is changing.

8.2 Multiple Process Groups

We have considered only the case of a single process group. For multiple process groups, a separate group tree would be constructed for each process group. Thus, if we have a total of n process groups, then each process could have a parent and a set of children for each group tree in which it belongs. The necessary changes for the multiple group case are straightforward. They consist of turning each variable into an array with n elements, one element for each group tree, and labeling each message with an additional field indicating the group of the message.

8.3 Fault Tolerance

It is possible to modify the propagated timestamp protocol to be stabilizing in the following sense [10]. If a fault occurs that leaves the system in an arbitrary state, (i.e., a state with an arbitrary assignment of values to its variables), then the system automatically converges to a normal operating state.

Specifically, consider the tree convergence property. It states that if a computation begins from a state in which C0´ holds, then eventually the group tree converges to the desired subset of the unicast spanning tree. For the protocol to be stabilizing, the group tree should converge to the desired subset of the unicast spanning tree

even though initially C0′ does not hold. That is, predicate C0′ in the property should be replaced by the predicate **true**.

To show that this stabilizing property holds, we only need to show how an arbitrary state eventually converges to a C0′ state. This, by transitivity with the already proven property [10], gives us the desired result.

For an arbitrary state to reach a C0′ state (recall C0′ = S0′ ∧ S1′), we need two changes. First, an action is needed that checks if chl is an empty set and mbr equals **false**. If that is the case, variables tpr and pr are set to **nil**. This restores S1′ automatically. To restore S0' automatically, we can ensure that the channels between a parent and its child are empty periodically (the channels could have bogus rqst and rply messages). This can be done by having rqst messages be created at rate slower than the rate at which messages are removed from the channel by the receive actions. Thus, eventually, the channels become empty, and S0′ is restored.

As a side effect of the stabilizing version of the tree convergence property mentioned above, an arbitrary state also eventually reaches a state in which predicate C2 holds. Thus, from any arbitrary state, the protocol converges to a normal operating state, and it is thus stabilizing.

8.4 Bounded Timestamps

One disadvantage of the propagated-timestamp protocol, is that its timestamps are unbounded. We would like to consider the case of a bounded timestamp in the range 0 . . b-1, for some integer constant b. For simplicity, we assume that b is even.

Since timestamps are bounded, we say that t < u iff u equals one of the following: (t+1) mod b, (t+2) mod b, . . . , (t+(b/2)) mod b.

The normal operation of the protocol requires the timestamp of the root to be at least the timestamp of any other process. Thus, the root should not generate timestamp t as long as there is a process in the group tree with a timestamp u, such that t < u.

If the time to propagate a timestamp throughout the entire group tree is about D seconds, then the root should generate timestamp t at least D seconds after it has used timestamp (t+(b/2)+1) mod b.

For example, if D = 120 seconds and b = 256 (8 bit timestamp), then an upper bound on the rate at which the root process may generate new timestamps is about one timestamp per second.

Although practical, the above handling of bounded timestamps is dependent on the propagation time of timestamps along the group tree. If timestamps propagate slower than estimated, then it could disrupt the normal operation of the protocol. For completeness, we next briefly describe a diffusing computation that allows the use of bounded timestamps without a timing constraint on their propagation.

The approach consists of not increasing the timestamp of the root until the root has learned that the previous timestamp has propagated throughout the entire tree. In this way, at most two different timestamps (e.g., t and t+1) can occur concurrently in the tree.

To accomplish this, each process includes in its rqst message its timestamp and a bit indicating whether or not all its descendants have a timestamp equal to its own. When a node receives from each child a rqst message with the bit set to true and with a timestamp equal to its own, it sets the bit to true in future messages sent to its parent. Thus, when the root has learned from all its children that all their descendants have a timestamp equal to its own, the root process is free to increase its timestamp.

Thus, for the above protocol, a timestamp with four values (two bits) suffices.

8.5 Dual Trees

When process p changes parents, it is still possible for p to receive data messages from its previous parent (who still has p in its child set). If this is the case, p will treat the message as if it had originated from a non-group member. That is, as if the data message was in route to the root process, and p was the first tree node encountered by the message. Thus, p will forward the data message to all its neighbors in the tree, including its new parent, therefore duplicating the message in the tree.

To prevent this, a bit in the header could indicate whether a data message has reached the group tree. Once the data message reaches the tree, the message is forwarded with this bit set to true.

In the above case, p has two options. Process p could forward the message only to its children (since the message has or will arrive to its new parent) or it could discard the message and expect to receive the message again from its new parent.

In the former case, message duplication may occur, since the same data message may still be on its way down from the root to the new parent of p, and thus p will receive the message again and forward it again to its children. In the latter case, message loss may occur. If the new parent received and forwarded the data message a few moments before p become its child, then p would never receive or forward the message to its children.

The above problems are relatively minor, since the number of messages that can be lost or duplicated while a process is changing parents is small. However, for completeness, we next discuss a technique to solve this problem.

Rather than maintaining a single tree, each node has a parent and a set of children in two trees: the new tree and the old tree. Each tree is identified by a timestamp. Thus, for example, timestamp t could be the old tree and timestamp t+1 the new tree. Each data message contains the timestamp of the tree over which it is being forwarded. Thus, if a received data message has a timestamp t, it is forwarded over the old tree, and if it has a timestamp t+1, it is forwarded over the new tree.

The root does not generate timestamp t+2 until it is sure that all nodes have received timestamp t+1, as done for the bounded timestamps protocol mentioned earlier.

Assume the old tree has timestamp t, and the new tree has timestamp t+1. When a node receives timestamp t+2 from a parent, the parent and children of the new tree become the parent and children of the old tree. Furthermore, the node sets the timestamp of the old tree to t+1, and the timestamp of the new tree to t+2. The parent in the new tree is the parent from which timestamp t+2 is received. When the process receives a request from a child with timestamp t+2, it adds this child to the set of children in the new tree.

Assume next that all nodes in the tree have received timestamp t+1, and timestamp t+2 is being propagated down the tree. Furthermore, data messages with timestamp t+1 are being forwarded along tree t+1. When the root learns that all nodes have received timestamp t+2, it forwards an empty data message with timestamp t+2 down the new tree. This message indicates to all nodes that tree t+2 is active, and

also that all new data messages should be forwarded over tree t+2 using timestamp t+2.

Note that old data messages with timestamp t+1 could still be in the process of being forwarded along tree t+1. Thus, the root waits long enough for these old messages to clear from the network (i.e., enough for the time-to-live of the messages to expire [17]). Then, the root is free to generate a new timestamp, t+3, causing all nodes to discard their information about tree t+1.

No message duplication occurs because each data message includes the timestamp of the tree over which it must be forwarded. Also, no message loss occurs because a node does not discard the parent and child information of a tree with timestamp t until all data messages with timestamp t have been cleared from the network.

9. 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 Symposium on Protocol Specification, Testing and Verification*, 1992.

[3] Ballardie T., ``Core Based Tree Multicast", Internet RFC, work in progress.

[4] Ballardie T., Francis P., Crowcroft J, ``Core Based Trees: An Architecture for Scalable Inter-Domain Multicast Routing", *ACM SIGCOMM Conference*, 1993.

[5] Cobb J, Gouda M, ``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] Sloan L., ``Mechanisms that Enforce Bounds on Packet Lifetimes", *ACM Transcations on Computers* , Vol. 1, No. 4, 1983, p. 311-330.

[18] Wilbur S., Handley M., ``Multimedia Conferencing: 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. Basic Safety Property

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 five actions.

The third action sends a request to neighbor pr and sets wr to pr, but only if wr is **nil**. Thus, the channels between p and pr were empty before execution, and the action does not violate 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 **nil**, provided $chl = \emptyset \wedge \neg mbr$ holds. Thus S1 remains true after executing the action.

The last action affects only S0, by setting p.wr to **nil**, but only if the channels between p.wr and p are empty, hence, S0 is preserved.

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

1.2. Tree Convergence Property

1.2.1 Part (a)

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

Since C0 is a closure, we assume it holds throughout the computation.

We are required to show that for any edge (p, q) not in the unicast tree, the edge is eventually removed permanently from the group tree.

If p.pr ≠ q at the initial state of the computation, then it remains so forever, since the actions of process p assign either **nil** or ROUTE(p, root) to p.pr, neither of which equals q .

Thus, we must show that if initially p.pr = q, then eventually action three is executed or action six is executed with only one child left in set chl.

Due to C0, if p.pr ≠ **nil**, then (chl ≠ ∅ ∨ mbr). Since mbr is constant, this can only change if chl becomes empty, i.e., action six is executed, setting p.pr = **nil**. Thus, we assume that chl remains non-empty, and we must show that action three is executed.

The guard of action three also requires p.wr = **nil**. If this is not the case, then by C0, there can be at most one message in the channels from p to q. If no message exists, then action seven executes due to fairness, and sets p.wr to **nil**. If there is a rqst then process q receives it and returns a reply. If there is a reply, the reply is received by p and p.wr set to **nil**. Thus, the guard of action three becomes true and remains true (since only action three can set p.wr to non-nil). By fairness, this action is executed, and p.pr ≠ q afterwards.

We next must show p is eventually permanently removed from the set of children of q. We begin when the computation reaches p.pr ≠ q permanently.

If there is a request or a reply in the channels between p and q, these messages (by fairness) are received and removed from the channels. Since p.pr ≠ q, p never sends another request to q, and thus action six of process q executed, removing p from q.chl. Since no more requests are sent to q by p, p is never again added to q.chl.

Thus, out(p, q) holds eventually and continues to hold forever.

1.2.2 Part (b)

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

Let (p, q) be an edge in the unicast spanning tree, and r be a process in the unicast subtree rooted at p that is a member of the process group. Let P be the path from r to the root (which includes (p, q)). We show by induction that all the edges in this path join the group tree.

As a base, consider the edge (r, s), where s is the edge from r to its parent in the unicast spanning tree. Since r is a member of the group, the first conjunct of its third action is enabled. If r.wr ≠ **nil**, then, there is at most one rqst or rply in the channels between r and s. With an argument similar to that in the proof of part (a) above, eventually r.wr becomes **nil**, and remains **nil** until the third action is executed.

When the third action is executed, r.pr is assigned s and a rqst message is sent to s. Furthermore, since the routing tables are stable, the only possible value assigned to r.pr in action three is s. In action six, r.pr cannot be assigned **nil**, since r.mbr is true. Hence, r.pr = s remains true continuously.

By fairness, the request is received by s and s adds r to its child set, i.e., the edge is in the group tree.

Furthermore, s cannot remove r from its child set, since r.pr = s indefinitely (action six of s cannot execute with j = r), and hence, r is always in the child set of s, which in turn implies that edge (r, s) is continuously on the group tree.

A similar argument shows that for any edge (x, y) in P, if x continuously has either a non-empty child set or is a member of the group, then edge (x, y) joins the group tree indefinitely. Thus, by induction over the edges of P, edge (p, q) joins the group tree indefinitely.

1.2.3 Part (c)

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

We first show that if a process p that is not a member of the group has no children indefinitely, then eventually each edge (p, q), for any neighbor q, is removed from the group tree permanently.

Since p is not a member of the group and has no children, from the basic safety property, p.pr = **nil** always holds. The guard of the third action of p is always disabled, and hence no new rqst is sent by p. Hence, any rqst or rply message in the channels between p and q is received and removed from the channels. Thus, process q will timeout (with j = p) and remove p from its set of children permanently (since no new rqst will be sent by p). Thus, edge (p, q) is removed permanently from the three.

To prove part (c), we first observe that all edges not in the multicast spanning tree are permanently removed from the tree (part (a)). For an edge (p, q) in the multicast tree, let S be the subtree at p of the unicast tree, and let no process in S be a member of the process group.

Notice that the leaves of S have no children, and hence, from the argument above, the edge between leaf and parent in S is removed permanently from the group tree. By induction on the levels of S, all edges in S are permanently removed from the group tree. Thus, p is left childless permanently, and hence (p, q) is removed permanently from the group tree.

2. Dual-Parent Protocol

Recall that the basic safety and tree convergence properties below are identical to the respective properties of the basic protocol, except that C0 is replaced by C0´.

2.1. Basic Safety Property

It can be proven that C0´ is a closure of this protocol in a very similar way to the proof of the basic safety property in the basic protocol, which we do not repeat here.

2.2. Tree Convergence Property

2.2.1 Part (a)

Due to the basic safety property, we assume predicate C0´ 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 would like to show is that eventually for all processes p, each of p.tpr and p.pr is equal to either ROUTE(p, root) or **nil**. Furthermore, this should

continue to be true for the remainder of the computation. If this is shown, then any edge (p, q) not in the unicast spanning tree is removed permanently from the group tree, because q will time out and remove p from its set of children.

Assume first that p.chl = ∅ and ∧ ¬p.mbr. Because C0´ holds, p.pr = **nil** and p.tpr = **nil**. Furthermore, if eventually p.chl = ∅ and ∧ ¬p.mbr does not hold, when action three is executed, p.tpr is assigned ROUTE(p, root). Since the routing table is constant, p.tpr can only have this value or **nil** from this point on. Similarly, since p.pr receives its value from p.tpr, from this point on, p.pr can only be assigned ROUTE(p, root) or be assigned **nil**.

Assume now that p.chl ≠ ∅ ∨ p.mbr. When action three is executed, p.tpr is assigned ROUTE(p, root), and since the routing tables are constant, p.tpr remains constant as long as p.chl ≠ ∅ ∨ p.mbr. Regarding p.pr, using an argument similar to that of part (b) below, if p.chl ≠ ∅ ∨ p.mbr remained constant, then all the edges from p to the root join the group tree permanently, i.e., eventually p.pr = ROUTE(p, root) and does not change value. On the other hand if p.chl ≠ ∅ ∨ p.mbr fails to hold along the computation, then we have the case of the previous paragraph.

2.2.2 Part (b)

Due to the basic safety property, we assume predicate C0´ 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.

Let (p, q) be an edge where utree(p, q), and let r be a descendant of p in the unicast spanning tree such that r.mbr is true. Let P be the path in the unicast spanning tree from r to the root.

Note that the dual-parent protocol treats variable tpr in a very similar way to the way variable pr is treated in the basic protocol, with the exception that the timeout is stronger since it refers both pr and tpr. Thus, a very similar argument to that of part (b) of the basic protocol shows that, eventually, any edge (x, y) in P has x ∈ y.chl and x.tpr = y. Furthermore, the argument shows that this remain continuously true.

We next need to show that after the above holds, for any edge (x, y) in P, x.pr is assigned y. This is done by induction on the path P, where P = r1, r2, ... , 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 set to 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 and rn.chl is never empty again, rn.pr will continuously have the value root. The inductive case is similar. Thus, all edges in P join the group tree and remain in it continuously.

2.2.3 Part (c)

Due to the basic safety property, we assume predicate C0´ 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.

Due to part (a), we assume that all edges not in the unicast spanning tree have been removed from the group tree permanently.

As argued in the proof of part (a) of the basic protocol, we have that if a process p that is not a member of the group has no children indefinitely, then eventually each edge (p, q), for any neighbor q, is removed from the group tree permanently.

For an edge (p, q) in the multicast tree, let S be the subtree at p of the unicast tree, and let no process in S be a member of the process group.

Notice that the leaves of S have no children, and hence, the edge between leaf and parent in S is removed permanently from the group tree. By induction on the levels of S, all edges in S are permanently removed from the group tree. Thus, p is left childless permanently, and hence (p, q) is removed permanently from the group tree.

2.3. Continuous Parent Property

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

2.4. Connectivity Property

It can be proven that C0´ is a closure of this protocol in a very similar way to the proof of the basic safety property 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 and S3.

For each of S2 and S3, 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 sixth action by assigning q to p.pr. If p.tpr = q, and p.tpr is assigned to p.pr, it implies that the bit received in the reply is true, which by S3 implies that q has p in its child set, and q is the root or q.pr ≠ **nil**. This preserves S2. On the other hand, the body of S2 can be falsified by removing p from q.chl or assigning **nil** 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 fifth 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 **nil**. This only happens in the timeout action, which is not enabled if the reply to q has not been received. Thus, S3 is preserved by action execution.

Since S2 and S3 are preserved through action execution, and C0′ is a closure, then C1 is a closure.

3. Propagated Timestamp Protocol

Recall that the basic safety property and the tree convergence property in this protocol are identical to the respective properties of the basic protocol, except that C0 is replaced by C0′.

3.1. Basic Safety Property

It can be proven that C0′ is a closure of this protocol in a very similar way to the proof of the basic safety property in the dual-parent protocol, which we do not repeat here.

3.2. Tree Convergence Property

3.2.1 Part (a)

The proof of this property is identical to the proof of the dual-parent protocol, with the exception that the reference to the proof of part (b) in the dual-parent protocol refers now to the proof of part (b) in the propagated timestamp protocol.

3.2.2 Part (b)

(This proof is a variant of the proof for the dual-parent protocol.)

Due to the basic safety property, we assume predicate C0´ 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.

Let (p, q) be an edge where utree(p, q), and there a descendant r of p in the unicast spanning tree such that r.mbr is true. Let P be the path in the unicast spanning tree from r to the root.

Note that the protocol treats variable tpr in a very similar way to the way variable pr is treated in the basic protocol, with the exception that the timeout is stronger since it refers both pr and tpr. Thus, a very similar argument to that of part (b) of the basic protocol shows that, eventually, any edge (x, y) in P has $x \in y.chl$ and x.tpr = y. Furthermore, the argument shows that this remain continuously true.

We next need to show that after the above holds, for any edge (x, y) in P, x.pr is assigned y, and the timestamp of x increases periodically. This is done by induction on the path r, r1, r2, ... , rn, root, with the edge (rn, root) as the base. Since rn.tpr = root, rn keeps sending requests periodically to the root, and the root returns a reply for each request. Notice that the root increases its timestamp periodically (since its child set is non-empty). Also, each reply sent has a timestamp copied from the root's timestamp. Thus eventually, the timestamp of the reply is greater than rn.ts. When this reply is received, rn assigns root to rn.pr and chooses the timestamp of the message as its own. Notice that rn.pr = root remains true, since rn.tpr = root also remains true continuously and rn.chl is always non-empty. Furthermore, the timestamp of rn increases periodically, since the reply message from the root continues to have increasing timestamps.

The inductive case is similar to the above. Thus, all edges in P join the group tree and remain in it continuously.

3.2.3 Part (c)

The proof is similar to that of part (c) in the dual-parent protocol and is not repeated here.

3.3. Continuous Parent Property

The proof is similar to that in the dual-parent protocol and is not repeated here.

3.4. Decreasing Timestamps Property

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 S4, S5, and S6. The remaining part of the proof is very similar to the connectivity property in the dual-parent protocol.

For S4, the range could become true in action six, 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 S5, 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 of S4. The body of S4 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 S5, ensures that after execution, p.ts is no grater than the timestamp of neighbor p.pr. Thus, S4 is preserved by action execution.

For S5, 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 impossible. Hence, S5 is preserved by action execution.

For S6, note that no process, including the root, ever decreases its timestamp, so S6 cannot be falsified by decreasing root.ts. Furthermore, any change to p.ts (where $p \neq root$) comes from the timestamp of a reply message, which from S5, is at most the timestamp of the sender, which from S6 is at most root.ts. Thus, p.ts is never assigned a value larger than root.ts.

Hence, C2 is a closure.

3.5. Loopless Property

From the decreasing timestamps property, we know that C2 is a closure, so we assume C2 holds in all states of the computation.

Let us denote by Q the quantification in the loopless property. 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 Q 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 Q could also become 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 $\neq$ **nil**, 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 **nil** 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 **nil**.

Thus, the loopless property holds.