

A Stabilizing Solution to The Stable-Paths Problem

Jorge A. Cobb¹, Mohamed G. Gouda², and Ravi Musunuri¹

¹ Department of Computer Science,
The University of Texas at Dallas,
Richardson, TX 75083-0688,
`{cobb,musunuri}@utdallas.edu`

² Department of Computer Science,
The University of Texas at Austin,
1 University Station C0500,
Austin, TX 78712-0233,
`gouda@cs.utexas.edu`

Abstract. The stable-paths problem is an abstraction of the basic functionality of the Internet’s BGP routing protocol. This abstraction has received considerable attention, due to the instabilities observed in BGP. In this abstraction, each process informs its neighboring processes of its current path to the destination. From the paths received from its neighbors, each process chooses the best path according to some locally chosen routing policy. However, since routing policies are chosen locally, conflicts may occur between processes, resulting in unstable behavior. Current solutions either require expensive path histories, or prevent processes from locally choosing their routing policy. In this paper, we present a solution with small overhead, and furthermore, each process has the freedom to choose any routing policy. However, to avoid instabilities, each process is restricted to choose a path that is consistent with the current paths of its descendants on the routing tree. This is enforced through diffusing computations. Furthermore, our solution is stabilizing, and thus, recovers automatically from transient faults.

1 Introduction

The Border Gateway Protocol (BGP) [14] has become the de-facto standard for exchanging routing information between Internet Service Providers (ISPs). Generally, ISPs choose a path to each destination based on a locally chosen routing policy. Furthermore, ISPs are unwilling to share their local policies with other ISPs. Due to this lack of coordination between ISPs, Varadhan et al. showed that BGP exhibits unstable behavior, i.e., the chosen path to each destination may continuously oscillate [16]. Furthermore, simple techniques like route-flap-damping [17] do not eliminate these oscillations, but simply increase the oscillation period. Furthermore, route-flap-damping may also significantly increase the convergence times of stable routes [13].

To study the problem formally, Griffin et al. proposed an abstract model of BGP routing behavior [9]. In this abstraction, the network consists of a set of processes interconnected by channels. Each process informs its neighboring processes of its current path to the destination. From the paths received from its neighbors, each process chooses the best path according to some local routing policy. Conflicts between local routing policies result in unstable behavior.

Three solutions to this problem have been proposed. The first approach analyzes the local routing policies of all processes, and decides whether the routing policy is “safe”, that is, whether the network is guaranteed to converge to a stable configuration. However, this approach has two disadvantages. First, ISPs are often unwilling to share their local policies. Second, Griffin and Wilfong [9] proved that deciding the safety of a routing policy is an NP-complete problem.

The second approach maintains path histories during run-time to determine if the network is oscillating and thus failing to converge. This approach was proposed by Griffin et al. in their safe path-vector protocol [10]. Oscillations are detected through loops in the path histories, and selected paths are removed to ensure convergence. However, the removed paths may be of later use as the network topology changes. Furthermore, the existence of a loop in the path history is necessary but not sufficient for divergence. Finally, maintaining path histories significantly increases the memory and message overhead.

The third approach, proposed by Gao and Rexford [3] [4] restricts the routing policy to a hierarchical structure. Therefore, no routing conflicts occur, and the network is guaranteed to reach a stable state. However, this severely restricts the choice of routing policies, and the desired property of locally choosing a routing policy is lost.

The solution we present in this paper introduces a small overhead, and furthermore, each process has the freedom to choose its individual routing policy. However, to avoid instabilities, each process is restricted to choose a path that is consistent with the current paths of its descendants on the routing tree. This is enforced through diffusing computations. Furthermore, our solution is stabilizing [15][12], and thus, it recovers automatically from transient faults.

2 Notation

We begin with an overview of our process notation. A similar notation may be found in [6]. A network consists of a set of processes interconnected via communication channels. Each channel transmits a sequence of messages between two processes. The channel from a process p to a neighboring process q is denoted $ch(p, q)$. A single message type, *path*, is used in our protocols. The number of *path* messages in channel $ch(p, q)$ is denoted $\#ch(p, q)$.

Channels have the following properties. For every pair of processes p and q , if there is a channel from p to q , then there is also a channel from q to p . For simplicity, even though channels may fail, we assume the network remains connected. In addition, channels may lose and reorder messages. Finally, each channel has a known bound on its message lifetime.

A process consists of a set of inputs, a set of variables, a parameter, and a set of actions. The inputs declared in a process can be read, but not written, by the actions of that process. The variables declared in a process can be read and written by the actions of that process. The parameter is discussed further below.

Every action is of the form: $\langle \text{guard} \rangle \rightarrow \langle \text{command} \rangle$. The $\langle \text{guard} \rangle$ can be of three types: local, receiving, and timeout. A local guard is a boolean expression over the inputs, variables, and parameter declared in the process. An action with a local guard is said to be enabled if its guard evaluates to true. A receiving guard at process p is of the form

$$\mathbf{rcv} \text{ } path \text{ from } q$$

where q is a neighbor of p . An action with this guard is enabled iff there is a message of type $path$ in channel $ch(q, p)$. Furthermore, if this action is chosen for execution, then this message is removed from channel $ch(q, p)$. Finally, a timeout guard at process p is of the form

$$\mathbf{timeout} \#ch(p, q) = 0 \wedge \#ch(q, p) = 0$$

where q is a neighbor of p . An action with this guard is enabled iff there are no $path$ messages in either channel $ch(p, q)$ or channel $ch(q, p)$ ¹.

The $\langle \text{command} \rangle$ in an action is a sequence of conditional statements or sends statements. Conditional statements are of the following form.

$$\langle \text{variable} \rangle := \langle \text{expression} \rangle \quad \mathbf{if} \langle \text{boolean} \rangle$$

If $\langle \text{boolean} \rangle$ is true before the conditional statement is executed, then $\langle \text{variable} \rangle$ is assigned the current value of $\langle \text{expression} \rangle$. If $\langle \text{boolean} \rangle$ is false, then $\langle \text{variable} \rangle$ remains unchanged. If the phrase $\mathbf{if} \langle \text{boolean} \rangle$ is not present, then the value of $\langle \text{expression} \rangle$ is assigned to $\langle \text{variable} \rangle$ unconditionally. A send statement in process p is of the form $\mathbf{send} \text{ } path \text{ to } q$, where q is a neighbor of p .

The parameter declared in a process is used to write a set of actions as a single action, with one action for each possible value of the parameter. For example, if we have the following parameter definition,

$$\mathbf{par} \ g : \mathbf{element} \ \mathbf{of} \ \{r, s\}$$

then the following action

$$\mathbf{rcv} \text{ } path \text{ from } g \rightarrow \mathbf{send} \text{ } path \text{ to } g$$

is a shorthand notation for the following two actions.

$$\frac{\mathbf{rcv} \text{ } path \text{ from } r \rightarrow \mathbf{send} \text{ } path \text{ to } r}{\mathbf{rcv} \text{ } path \text{ from } s \rightarrow \mathbf{send} \text{ } path \text{ to } s}$$

¹ This form of timeout can be easily implemented using timers, as shown in [6].

An execution step of a protocol consists of choosing an enabled action out of all the actions of all processes, and executing the command of this action. An execution of a protocol consists of a sequence of execution steps, which either never ends, or ends in a state where all actions are disabled. We assume all executions of a protocol are weakly fair, that is, an action whose guard is continuously true must be eventually executed.

3 Ordered Paths

As a prelude to the stable-paths problem, we formally define the routing policy of a process to be an ordering on all paths originating at the process. In addition, we present a greedy protocol where each process attempts to improve the order of its path without regard for other processes. We begin with some notation.

Definition 1. *A path is a sequence of processes. Paths have the following properties:*

- $|P|$ denotes the number of processes in the path.
- λ denotes the empty path.
- P_i denotes the i th process in path P . P_1 and $P_{|P|}$ are the first and last processes in P , respectively.
- $p:P$ denotes the concatenation of process p with path P .
- Process root is a distinguished process. A path is rooted iff it is simple and its last process is root.
- A path is a network path iff, for every pair of consecutive processes p and q along the path, p and q are neighbors.

Each process p attempts to find a network path from itself to *root*. However, paths must be consistent between neighbors: if P and Q are the paths chosen by p and q , respectively, and if $q = P_2$, then $p:Q = P$.

We assume that rooted paths are ordered, and that processes are greedy. That is, each process chooses the path with highest order among the paths offered by its neighbors. We next formalize the ordering of paths.

Definition 2. *A path ordering is a tuple $\langle V, \preceq \rangle$, where:*

- V is a set of processes, and \preceq is a transitive relation on paths from V ,
- for every pair of rooted paths P and Q ,

$$(P_1 = Q_1) \Rightarrow (P \preceq Q \vee Q \preceq P)$$

$$(P \prec Q) \equiv (P \preceq Q \wedge Q \not\preceq P)$$

- for every path P , $P \preceq \text{root}$.
- for every rooted path P , $\lambda \prec P$.

Intuitively, if $p = P_1 = P'_1$, then $P < P'$ indicates that p prefers path P' over path P , i.e., P' has a higher order than P .

We next present the *greedy protocol*. In this protocol, each process chooses the highest ordered path from those offered by its neighbors. To do so, each pair of neighboring processes exchange a single *path* message between them. This message contains the current rooted path chosen by the process.

The specification consists of two processes: process *root* and a non-root process p . These processes store in variable P their current rooted path. Process *root* simply stores *root* in its variable since the highest ordered rooted path from *root* to itself is just itself. The greedy protocol is specified as follows:

```

process  $p$ 
inp   $N$  : set of neighbors
var   $P$  : path,           // path of  $p$  //
       $G$  : path           // path from neighbor  $g$  //
par   $g$  : element of  $N$  // any neighbor //
begin
  rcv  $path(G)$  from  $g$   $\rightarrow$ 
     $P := p:G$            if  $P < p:G \vee g = P_2$ ;
     $P := \lambda$          if  $\neg sound(P)$ ;
    send  $path(P)$  to  $g$ 
   $\square$ 
  timeout  $\#ch(p, g) = 0 \wedge \#ch(g, p) = 0 \rightarrow$ 
    send  $path(P)$  to  $g$  if  $p < g$ 
end

process root
inp   $N$  : set of neighbors
var   $P$  : path           // path of root //
       $G$  : path           // path from neighbor  $g$  //
par   $g$  : element of  $N$  // any neighbor //
begin
  rcv  $path(G)$  from  $g$   $\rightarrow$ 
     $P := root$ ; send  $path(P)$  to  $g$ 
   $\square$ 
  timeout  $\#ch(root, g) = 0 \wedge \#ch(g, root) = 0 \rightarrow$ 
    send  $path(P)$  to  $g$  if  $root < g$ 
end

```

Both processes contain two actions: a timeout action and a receive action. The timeout action retransmits the *path* message if it is lost, and is similar in both processes. The receive action is different between both processes.

In process p , upon receiving a *path* message from g , P is updated by concatenating p to the path given by g , i.e., process p chooses g as the next process along its path to *root*. This update is performed only under the following two conditions. First, if g is already the next process along the path of p , i.e., $g = P_2$.

In this case, any change in the path of g must be reflected in the path of p . Second, if the path through g has a higher order than P . In this case, to improve the order of its path, p chooses the path through g . After updating its path, p performs a sanity check to see if its path P is sound, where $sound(P)$ is defined as follows:

$$sound(P) \equiv (P_1 = p \wedge P_2 \in N \wedge rooted(P)).$$

If the path is not sound, then the empty path is chosen. After the sanity check, p returns the *path* message to g containing the updated path P .

The receive action in process *root* simply returns the *path* message to g with *root* as the path, because this path has the highest order in the network.

4 Stable-Paths Problem

Conflicts between the routing policies of neighbors may cause the greedy protocol to never achieve a stable configuration. An example² of these conflicts is shown in Figure 1. Arrows indicate the current path taken by each process, and the ordering of paths at each process is given in a table at the bottom of the figure. Paths not included in the table are assumed to have the lowest order.

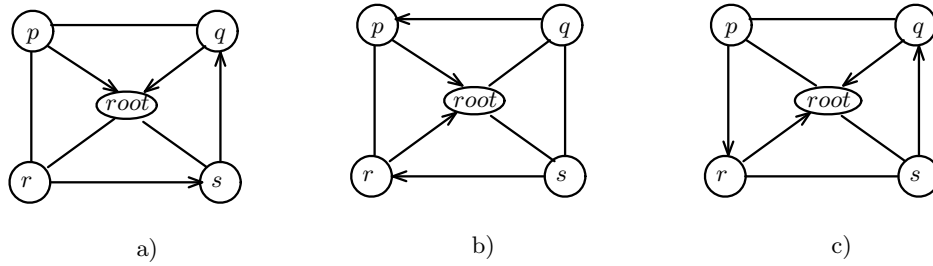
Figure 1(a) shows the initial choice of each process. Process q notices that the path through p is ordered higher than its current path, and changes its path to $q:p:root$. Next, s notices that the path through q is ordered lower than the path directly to *root*, and updates its path accordingly. Then, r notices that the path directly to *root* is ordered higher than the new path through s , and changes its path to $r:root$. Finally, s notices that the path through r is ordered higher than its current path directly to *root*, and changes its path to $s:r:root$. The resulting configuration is shown in Figure 1(b). Similarly, Figure 1(c) is derived from Figure 1(b) after processes p , q , and s , in this order, change their paths. Finally, Figure 1(a) is obtained from Figure 1(c) after processes r and p change their paths. Thus, a stable configuration is never reached.

Given the path ordering, the stable paths problem consists of deciding if the greedy protocol will always reach a stable configuration. However, as shown in [11], this problem is NP-complete. Therefore, we propose to strengthen the greedy protocol to ensure it always reaches a stable configuration. In our enhanced protocol, any path ordering may be used, and path histories are unnecessary. However, to ensure convergence, a process is prevented from choosing a path with higher order if by doing so it causes a conflict with other processes.

5 Monotonic Path Orderings

In [8] [9], it is shown that if the path ordering is consistent with a cost function, then a stable configuration is always achieved. Below, we give a more general

² This is the classical “bad gadget” example first introduced in [8].



process	lowest order	medium order	highest order
p		$p:root$	$p:r:root$
q		$q:root$	$q:p:root$
r	$r:s:root$	$r:root$	$r:s:q:root$
s	$s:root$	$s:r:root$	$s:q:root$

Fig. 1. Unstable path selection

result based on path monotonicity³. Path monotonicity formalizes the notion of consistency in a path ordering.

Definition 3. A path ordering $\langle V, \preceq \rangle$ is monotonic if and only if, for every pair of processes p and q , and for every pair of rooted paths Q and Q' originating at q , such that $p \notin Q$ and $p \notin Q'$,

$$Q \preceq Q' \Rightarrow p:Q \preceq p:Q'$$

If a path ordering is monotonic, and q is the next process along the path of p , then any change of path performed by q to improve the order of its path results in an increase in the order of the path of p . That is, q is consistent with p .

Theorem 1. If $\langle V, \preceq \rangle$ is a monotonic path ordering, then the greedy protocol always converges to a stable configuration. Furthermore, at this stable configuration, the path chosen by each process is the highest-ordered rooted network path originating at that process.

From above, if a network has a monotonic path ordering, then a stable state is guaranteed to be reached. However, path orderings are chosen independent at each process, and thus, they cannot be guaranteed to be monotonic. Below, we show how we can modify the greedy protocol to ensure that even if the path ordering is not monotonic, the network is guaranteed to reach a stable state.

6 Enforcing Monotonic Path Orderings

If a path ordering is monotonic, then the order of the paths chosen by each process is nondecreasing. To see this, consider a process p and the next process

³ Path monotonicity is similar to the monotonicity of routing metrics presented in [7].

q along the path of p . If q chooses a new path whose order is at least the order of its previous path, then the order of the new path of p is at least the order of its previous path. Thus, stability is assured.

Since we have chosen to allow any path ordering, we strengthen the protocol to ensure that the order of the chosen path at each process is non-decreasing. We refer to this restriction as the *monotonic ordering property*.

Note that it is not sufficient for each process to choose a new path whose order is greater than the order of its previous path. This is because, when a process p changes its path, each process q whose path includes p must update its path to reflect the new path of p . The new path of q may have a lower order than its original path, violating the monotonic ordering property.

The above violation is prevented as follows. Before p adopts a new path, p asks q if this change of path will cause the order of the path of q to decrease. If this is the case, p refrains from adopting the new path.

The coordination between p and q is performed via a diffusing computation [5] along the *routing tree*. The routing tree is defined as follows. For every process p and its next-hop neighbor q along its path, consider the directed edge (p, q) . The union of all these edges over all processes form a routing tree. In addition, if edge (p, q) is on the routing tree, then p is a *child* of q and q is the *parent* of p . Similarly, if there is a path from p to q along the routing tree, then p is a *descendant* of q (denoted $desc(p, q)$), and q is an *ancestor* of p (denoted $anc(q, p)$). In particular, every process is its own ancestor and descendant.

Diffusing computations are performed along the subtree of the process desiring a new path. When process p desires a new path, p propagates its new path along its subtree. When process q in the subtree of p receives the path of p , it determines if this new path, along with the network path from q to p , has a lower order than its current path. If so, q rejects the new path, and p is prevented from adopting the new path. Otherwise, q continues the propagation of the new path of p down the subtree. If all processes in the subtree allow the new path, a positive feedback is sent to p , and p adopts the new path.

Process p maintains two path variables, P and P^* , that store the current path of p and the tentative path of p , respectively. In addition, p maintains a set, called *clean*, where it stores the identifiers of neighboring processes that have allowed p to proceed with the tentative path P^* .

Neighboring processes periodically exchange a single message between them, called *path*. This message contains three fields: the current path of the process, the tentative path of the process, and a boolean bit indicating if all neighbors have allowed the process to adopt its tentative path, that is, if *clean* contains all neighbors of the process.

As an example, consider Figure 2. The current path from p to *root* is P , and the tentative path from p to *root* is P^* . Before p adopts the new path P^* , it must consult with all descendants in its subtree. To do so, p includes P^* in the next *path* message it sends to its neighbors⁴. This is depicted in Figure 2(a).

⁴ Process p also includes two other fields in this message. However, only the necessary fields at each step in the figure will be shown.

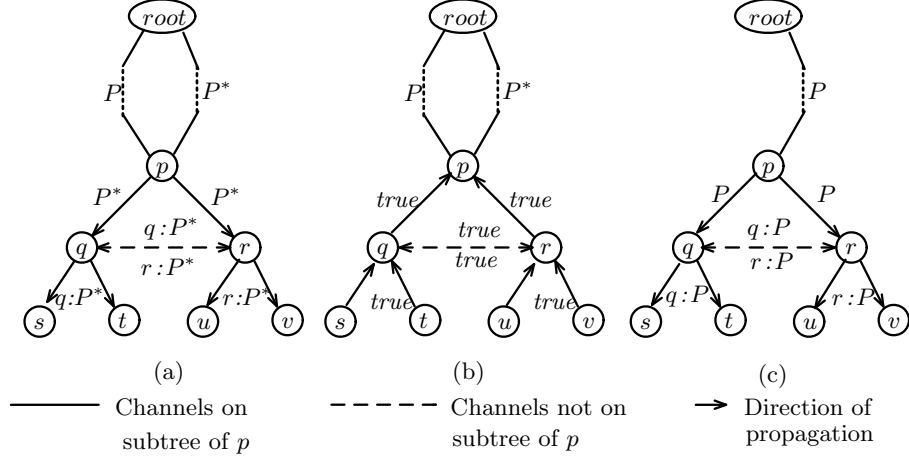


Fig. 2. Diffusing computation example

When q and r receive this message, they check if the order of the new path is at least the order of their current path. If so, their tentative paths are set to $q:P^*$ and $r:P^*$, respectively, and these tentative paths are included in the *path* messages they send to their neighbors. Similarly, s and t receive the tentative path $q:P^*$ from q , and u and v receive the tentative path $r:P^*$ from r .

Since $s, t, u,$ and v are childless, they return *true* bit of the next *path* message to q and r . This is shown in Figure 2(b). Note that q and r reply *true* to each other since neither is a child of the other. Thus, q and r receive a positive reply from each neighbor, and thus, each replies a *true* bit to p .

The final step is shown in Figure 2(c). Process p assigns P^* to P , effectively adopting P^* as its current path. Then, the updated path P is propagated to all descendants of p . Each process along the way appends its own identifier to the path, ensuring that each process receives the entire path to *root*.

7 Monotonic Paths Protocol

In this section, we present the specification of the monotonic paths protocol that was briefly described in the previous section. Each process p maintains variables P, P^* and *clean*, described earlier. In addition, it maintains a set of neighbor id's, named *wait*. If $g \in \text{wait}$, then p ignores the next *path* message from g when deciding if the diffusing computation ended along the subtree of g . The reason for this is explained below.

The specifications of process *root* and of a non-root process p are given below.

```

process root
inp   $N$       : set of neighbors
var   $P, P^*$  : path,           // path and tentative path of root //
       $G, G^*$  : path,           // path and tentative path of neighbor  $g$  //
       $b$       : boolean,         // boolean reply bit from neighbor  $g$  //
       $wait$   : subset of  $N$ ,    // ignore next message //
       $clean$  : subset of  $N$     // neighbors in agreement with root //
par   $g$       : element of  $G$ 
begin
  rcv  $path(G, G^*, b)$  from  $g \rightarrow$ 
     $P, P^*, wait, clean := root, root, \emptyset, N;$ 
    send  $path(P, P^*, true)$  to  $g$ 
   $\square$ 
  timeout  $\#ch(root, g) = 0 \wedge \#ch(root, p) = 0 \rightarrow$ 
    send  $path(P, P^*, true)$  to  $g$  if  $p < g$ 
end

process  $p$ 
inp   $N$       : set of neighbors
var   $P, P^*$  : path,           // path and tentative path of  $p$  //
       $G, G^*$  : path,           // path and tentative path of neighbor  $g$  //
       $b$       : boolean,         // boolean reply bit from neighbor  $g$  //
       $wait$   : subset of  $N$ ,    // ignore next message //
       $clean$  : subset of  $N$     // neighbors in agreement with  $p$  //
par   $g$       : element of  $G$ 
begin
  rcv  $path(G, G^*, b)$  from  $g \rightarrow$ 
     $P := p : G$            if  $g = P_2 \vee (clean = N \wedge P^* = p : G \wedge G = G^*);$ 
     $P^* := P$            if  $(g = P_2^* \wedge P^* \neq p : G^*) \vee$ 
       $(p = G_2 \wedge G^* \neq g : P^* \wedge G = G^* \wedge g \notin wait);$ 
     $P^* := G^*$          if  $P = P^* \wedge P \preceq p : G^* \wedge (g \neq P_2 \Rightarrow P \prec p : G^*);$ 
     $P, P^* := \lambda, \lambda$  if  $\neg sound(P) \vee \neg sound(P^*);$ 
     $wait, clean := N, \emptyset$  if  $new(P, P^*);$ 
     $wait := wait - \{g\};$ 
     $clean := clean \cup \{g\}$  if  $p \neq G_2 \vee (G = g : P \wedge G^* = g : P^* \wedge b);$ 
    send  $path(P, P^*, clean = N)$  to  $g$ 
   $\square$ 
  timeout  $\#ch(p, g) = 0 \wedge \#ch(g, p) = 0 \rightarrow$ 
    send  $path(P, P^*, clean = N)$  to  $g$  if  $p < g$ 
end

```

Each process consists of two actions: a timeout action to retransmit the *path* message, and a receive action to receive the *path* message from a neighbor g .

Process *root* has only minor changes. Thus, consider process p , and consider first the receive action. Upon receiving a *path* message from g , p must decide whether or not to perform each of the following steps:

- Process p may update its path P to follow the path G of neighbor g , that is, P is assigned $p:G$, and g becomes the parent of p .
- Process p may choose to abort its current diffusing computation. This is represented by assigning its current path P to its tentative path P^* .
- Process p may choose to start a new diffusing computation to propagate the tentative path G^* of g , in an attempt to gain approval for this path from the descendants of p . Thus, P^* is assigned $p:G^*$.

These three steps are performed by the first three assignment statements of the receive action.

Consider the first step of choosing g as the parent of p by updating P from G . This is performed under two conditions. First, if g is the parent of p (i.e., $g = P_2$), then p must update P from G , as described in Section 3. Second, p may have started a diffusing computation using the tentative path from a neighbor g who is not current parent of p . If so, p chooses g as its new parent provided all descendants agreed (i.e., $clean = N$), the path offered by g is the desired path (i.e., $P^* = p:G$), and g is not involved in a diffusing computation (i.e., $G = G^*$).

Next, consider when p aborts its diffusing computation by assigning P to P^* . This is done in two occasions. One occasion is when g is the tentative parent of p and the path being propagated by g is not the desired path (i.e., $g = P_2^* \wedge P^* \neq p:G^*$). The other occasion is when g is a child of p (i.e., $p = G_2$), g has not adopted the tentative path of p (i.e., $G^* \neq g:P^*$), and the diffusing computation along g has finished (i.e., $G = G^*$). This indicates that g rejected the tentative path. The last conjunct, $g \notin wait$, is explained below.

Consider the third step of initiating a new diffusing computation. This is performed when p is currently not involved in a diffusing computation, and the tentative path offered by g has an order at least that of the current path of p (i.e., $P = P^* \wedge P \preceq p:G^*$). Also, if g is not the parent of p , then the order of the tentative path must be strictly greater than the order of the current path (i.e., $g \neq P_2 \Rightarrow P \prec p:G^*$). This is necessary to ensure the network reaches a stable configuration, rather than alternating between paths of the same order.

Next, process p checks if either P or P^* have changed (i.e., $new(P, P^*)$). If either of these changed, then $clean$ is assigned the empty set, because the paths of the descendants of p may no longer be consistent with the paths of p .

Assume that the paths of p did change, and subsequently, p receives a *path* message from a child g , where $G = G^* \neq P^*$. In this case, p cannot determine whether g is rejecting the diffusing computation, or simply g has not yet received the new paths from p . The neighbor set $wait$ aids in distinguishing between these two cases. Whenever the paths of p change, $wait$ is assigned the entire set of neighbors, indicating that the next message from each neighbor should not be considered to determine if the diffusing computation has terminated. When the next message is received from g , g is removed from $wait$.

Before propagating the newly obtained paths to its neighbors, p checks if these paths are sound. If they are not, both are set to the empty path.

Finally, before sending a *path* message to g , p checks whether the diffusing computation has finished along the subtree of g . This is true if either g is not

a child of p (i.e., $p \neq G_2$), or g has the desired paths and reports all of its descendants as consistent (i.e., $G = p:P \wedge G^* = p:P^* \wedge b = true$).

The timeout action of p is similar to the timeout action in the greedy protocol, except that both P and P^* are included in the message, plus a bit indicating if all descendants agree with p (i.e., $clean = N$).

8 Stabilization

A protocol is said to be stabilizing iff, when started from an arbitrary initial state, it converges to a state contained in a set of legitimate states, and the set of legitimate states is closed under execution [15, 12]. In this section, we perform a slight modification to the monotonic paths protocol to ensure it is stabilizing. We refer to this final protocol as the stabilizing monotonic-paths protocol.

To distinguish between variables with the same name but in different processes, we prefix variables with the name of their process.

The following relationship is crucial for the correct behavior of the monotonic paths protocol: for every p and q , where $desc(p, q) \wedge q.P \neq q.P^* \wedge q.clean = q.N$:

$$p.P = rt(p, q):q.P \wedge p.P^* = rt(p, q):q.P^* \wedge p.clean = p.N$$

where $rt(p, q)$ corresponds the path in the routing tree from p to q .

This relationship, however, is not restored automatically, for the following reason. Assume q propagates its current and tentative paths to a descendant p . Although p is required to adopt the current path of q , it is *not* required to adopt the tentative path of q , because its order may be lower than the current path of p . In this case, p is free to reject the tentative path of q .

We solve this problem by removing p from the routing tree whenever this relationship does not hold. That is, p assigns the empty path to both its current and tentative paths. In addition, to enforce monotonicity, we must ensure that $P \preceq P^*$. Note that this is violated only in faulty states, and it can be restored by assigning an empty path to both P and P^* .

Both requirements above are accomplished by weakening the **if** condition on the fourth assignment statement in the receive action to the following:

$$P, P^* := \lambda, \lambda \quad \mathbf{if} \quad \neg sound(P) \vee \neg sound(P^*) \vee P \not\preceq P^* \vee (g = P_2 \wedge G \neq G^* \wedge b) \not\Rightarrow (P = p:G \wedge P^* = p:G^* \wedge clean = N)$$

9 Correctness Proof

We next present an outline of the correctness proof of the stabilizing monotonic-paths protocol. Due to space constraints, the detailed proofs are found in [2].

Given that our network is based on message passing, and to ensure stabilization, we make some assumptions about the timing of sending and receiving events. Specifically, each process must receive and handle incoming messages faster than its neighbors can send messages to it. This is easily implemented by

placing an upper limit on the rate at which each process can send a new *path* message, and by placing a lower limit on the rate at which incoming messages are handled. Assuming each channel can only store a finite number of messages, then, eventually, for every pair of neighboring processes, only a single path message circulates between them.

Lemma 1. *Starting from an arbitrary state, eventually the following hold and continue to hold.*

- For every pair of neighboring processes p and q , $\#ch(p, q) + \#ch(q, p) \leq 0$.
- All path variables of all processes and all path values in all path messages are either sound or are equal to the empty path.
- For every process p , $p.P \preceq p.P^*$, and for every message $path(G, G^*, b)$ in any channel, $G \preceq G^*$.

Our proofs are based on induction over the number of hops between a process and its descendants. We define $hops(p, q)$ to be the number of hops from p to q along the routing tree. Furthermore, for all p , $hops(p, p) = 0$. In addition, we refer to the values contained in a *path* message using the following notation. The first field of the *path* message in the channel from p to q is referred to as $path(p, q).P$, while the second and third fields of the same message are referred to as $path(p, q).P^*$ and $path(p, q).b$, respectively.

Lemma 2. *From any arbitrary state, the network reaches a state where both of the following hold and continue to hold.*

- For every process p and neighbor g of p , if $p.P \neq p.P^* \wedge g \in p.clean \wedge \#ch(p, g) > 0$, then:

$$path(p, g).P = p.P \wedge path(p, g).P^* = p.P^*$$

- For every process p and neighbor g of p , such that

$$\#ch(p, g) > 0 \wedge path(p, g).P \neq path(p, g).P^* \wedge path(p, g).b$$

we have

$$p.P_1 \neq g \vee (p.P = path(p, g).P \wedge p.P^* = path(p, g).P^* \wedge p.clean = p.N)$$

We define $Diffusing(k)$, where $k \geq 1$, as follows: for every process p and every neighbor g of p , such that $p.P \neq p.P^* \wedge g \in p.clean$, and for every descendant q of g , such that $hops(q, p) \leq k$, and for every neighbor r of q ,

$$q.P = rt(q, g):p.P \wedge q.P^* = rt(q, g):p.P^* \wedge q.clean = q.N \wedge (\#ch(q, r) > 0 \Rightarrow (path(q, r).P = q.P \wedge path(q, r).P^* = q.P^* \wedge path(q, r).b))$$

Lemma 2 serves as a base case for an induction proof of $Diffusing(k)$, resulting in the following theorem.

Theorem 2. *For every k , $k \geq 1$, starting from an arbitrary state, eventually $Diffusing(k)$ holds and continues to hold.*

Theorem 2 shows that eventually the values indicating the successful completion of a diffusing computation may be trusted. Its proof is similar to the proofs of diffusing computations presented in [1].

Although $Diffusing(k)$ holds for all k , this does not imply, however, that all processes are part of the routing tree. In particular, to reestablish $Diffusing(k)$ some processes may choose an empty path. We must show that this eventually stops, and all processes obtain and retain a rooted path.

As an additional notation, for a network rooted path R , $R_{j,root}$ denotes the path along the routing tree from R_j to $root$.

We define $SoundPath(R)$, where R is a rooted network path, as follows. There exists a k , $0 \leq k \leq |R| - 1$, such that both of the following hold.

- for every j , $k < j < |R|$,

$$R_j.P = R_{j,root} \wedge (\#ch(R_{j+1}, R_j) > 0 \Rightarrow path(R_{j+1}, R_j).P = R_{j+1,root})$$

and for every neighbor r of R_j ,

$$\begin{aligned} \#ch(R_j, r) > 0 \Rightarrow (path(R_j, r).P = R_{j,root} \vee \\ path(R_j, r).P \neq path(R_j, r).P^* = R_{j,root}) \end{aligned}$$

- if $0 < k$, then

$$\begin{aligned} R_k.P \neq R_k.P^* = R_{k,root} \wedge R_k.clean = R_k.N \wedge \\ \#ch(R_k, R_{k+1}) > 0 \Rightarrow (path(R_k, R_{k+1}).b \wedge \\ path(R_k, R_{k+1}).P \neq path(R_k, R_{k+1}).P^*) \end{aligned}$$

Predicate $SoundPath(R)$ indicates that the processes along the rooted path are in agreement with each other. That is, the path can be divided into two segments. In the first segment, from R_{k+1} to $root$, all processes have in variable P the current path along the routing tree to $root$. In the second segment, from R_1 up to R_k , each process has a value for variable P that is different from the path along the routing tree, but the process is expecting to receive the routing-tree path in the final step of the diffusing computation.

Theorem 3. *From an arbitrary initial state, the following eventually holds and continues to hold:*

(for every R , where R is a network and rooted path, $SoundPath(R)$)

Finally, since all paths along the routing tree are in agreement, eventually all processes take part in the routing tree.

Theorem 4. *Starting from an arbitrary initial state, all of the following eventually hold and continue to hold.*

- Every process is located in the routing tree.
- For every process p , $p.P$ ceases to change value, and is equal to the path along the routing tree from p to $root$.
- If the path ordering is monotonic, then for every process p , $p.P$ is the highest ordered network path from p to $root$.

10 Concluding Remarks

We have assumed throughout the paper that although a process may prefer some paths over others, it will not reject paths, that is, it will always choose one of the paths being offered by its neighbors. However, as mentioned in [8][9], it is possible for a process to maintain a list of unacceptable paths. If all its neighbors offer only paths from this unacceptable-paths list, then the process will choose the empty path, and disconnect itself from the routing tree. In a future paper, we will enhance our model and protocols to handle lists of unacceptable paths.

References

1. Cobb, J.A., Gouda, M.G., "Stabilization of General Loop-Free Routing", *Journal of Parallel and Distributed Computing*, Vol. 62, No. 5, May, 2002.
2. Cobb, J.A., Gouda, M.G., Musuburi, R., "A Stabilizing Solution to The Stable-Paths Problem", Tech. Report, Dept. of Comp. Science, Univ. of Texas at Dallas.
3. Gao, L., Rexford, J., "Stable Internet Routing Without Global Coordination", *IEEE/ACM Transactions on Networking*, Vol. 9, No. 6, Dec. 2001.
4. Gao, L., Rexford, J., "Stable Internet Routing Without Global Coordination", *Proc. of the ACM SIGMETRICS*, June 2000.
5. Garcia-Lunes-Aceves, J.J., "Loop-Free Routing Using Diffusing Computations", *IEEE/ACM Transactions on Networking*, Volume 1, No. 1, Feb., 1993.
6. Gouda, M.G., *Elements of Network Protocol Design*, John Wiley & Sons, 1998.
7. Gouda, M.G., Schneider, M., "Maximizable Routing Metrics", *Proc. of the IEEE International Conference on Network Protocols*, 1998.
8. Griffin, T.G., Shepherd, F.B., Wilfong, G., "Policy Disputes in Path Vector Protocols", *Proc. of the IEEE Int'l Conf. on Net. Protocols*, Oct., 1999.
9. Griffin, T.G., Shepherd, F.B., Wilfong, G., "The Stable Paths Problem and Inter-domain Routing", *IEEE/ACM Tran. on Networking*, Vol. 10, No. 2, Apr. 2002.
10. Griffin, T.G., Wilfong, G., "A Safe Path Vector Protocol", *Proc. of the INFOCOM Conference*, 2000.
11. Griffin, T.G., Wilfong, G., "An Analysis of BGP Convergence Properties", *Proc. of the ACM SIGCOMM Conf.*, 1999.
12. Herman, T., "A Comprehensive Bibliography on Self-Stabilization", *Chicago Journal of Theoretical Computer Science*, working paper, 2002.
13. Mao, Z.M., Govindan, R., Varghese, G., and Katz, R., "Route Flap Damping Exacerbates Internet Routing Convergence", *ACM SIGCOMM Conf.*, Aug., 2002.
14. Rekhtar, Y., Li, T., "A Border Gateway Protocol" RFC1771, 1995.
15. Schneider, M., "Self-Stabilization", *ACM Computing Surveys*, Vol. 25, No. 1, 1993.
16. Varadhan, K., Govindan, R., Estrin, D., "Persistent Route Oscillations in Inter-Domain Routing", *Computer Networks*, Jan. 2000.
17. Villamizar, C., Chandra, R., Govindan, R., "BGP Route Flap Damping", RFC 2439, 1998.