Composite Routing Protocols

Anish Arora, Mohamed Gouda and Ted Herman
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712–1188

May 1990

Abstract

A novel routing protocol for computer networks is designed by combining two routing protocols that *adapt* to changes in the network topology. The first routing protocol is expensive to operate but yields shortest-paths in the networks; therefore, it is best used when changes in the network topology are rare. The second routing protocol is not as costly to operate, but may yield longer paths; hence, it is best used when changes in the network topology are frequent. By combining these two protocols, the resulting protocol has a wide domain of applicability that does not depend on the frequency of changes in the network topology.

Keywords: adaptive routing, multi-phase protocols, self-stabilization, shortest-paths, spanning trees.

1 Introduction

The characteristics of a computer network (e.g., topology, number of nodes, link speeds, waiting times for service) change over time. These changes are due to a variety of reasons that include node and link failures and subsequent repairs, changes in the demands over the network, and network growth and evolution. It is, therefore, important that network protocols be designed to adapt to the changes in the network characteristics.

The traditional method for designing an *adaptive protocol* is to design the protocol in terms of some parameters whose values reflect the network characteristics relevant to the protocol. Changes in these network characteristics are eventually reflected as changes in the values of the parameters which in turn affect the protocol operation. One example of an adaptive protocol is the adaptive routing protocol of the ARPANET [QRR80]. In this protocol, changes in the network are eventually reflected as changes in the routing tables of different nodes in the network. The changed routing tables are ultimately used by the protocol to determine the routes of all messages through the network.

One problem with the traditional method is that it is based on the assumption that an adaptive protocol remains fixed during its operation, and only the values of its parameters are changed to reflect the changes in the network characteristics. Such adaptive protocols cannot be used in networks where changes in the network characteristics can be so large that they require changing the protocol itself, and not merely its parameters.

In [GH89], we develop a new method for designing adaptive protocols as a "combination" of two or more constituent protocols. Each constituent protocol is adaptive in its own right: its operation depends on a number of parameters whose values reflect the current network characteristics that are of relevance to the protocol. At each instant, one of the constituent protocols executes and its operation adapts to the "small" changes in the network characteristics. When these changes become "large" another constituent protocol commences execution and its operation adapts to the "small" changes in the network characteristics, and so on.

An important issue in constructing such a composite protocol is to determine how to switch between constituent protocols. As there is no strong synchronization between computers in a network, different computers in the network perform the switch at different times. Thus, although all computers eventually end up switching to the same constituent protocol, the global state of the network may, at some instant in the switch, be a "bad" state for this protocol. The way we chose to solve this problem is to design constituent protocols to be self-stabilizing; that is, if a protocol starts at a bad global state it eventually converges of its accord to a "good" global state, and from that point on, it operates as expected [BGW89, D74]. An advantage of this design strategy is that the resulting composite protocol is itself self-stabilizing which implies that if it reaches a bad global state due to some fault it eventually converges to a good global state.

In this paper, we apply this method in designing an adaptive routing protocol by combining two protocols. The first routing protocol is expensive to operate but yields shortest-paths in the networks; therefore, it is best used when changes in the network topology are rare. The second routing protocol is not as costly to operate, but may yield longer paths; hence, it is best used when changes in the network topology are frequent. The protocol that results from combining these protocols has a wide domain of applicability that does not depend on the frequency of changes in the network topology. As we show later, all three protocols are self-stabilizing.

The rest of this paper is organized as follows. In the next section, we define the routing problem in computer networks. Two different protocols for solving the routing problem are presented in Sections 3 and 4. Then, in Section 5, we compare the behaviors of these two protocols: in particular, we show that the first protocol is best suited to networks in which topology changes occur infrequently while the second protocol is best suited to networks in which topology changes are frequent. In Section 6, we discuss how these protocols can be combined to yield a composite routing protocol. Finally, we make concluding remarks in Section 7.

2 Routing Protocols in Computer Networks

We represent a computer network as an undirected, connected graph (V, E), where V is the set of *nodes* and E is the set of *links* in the graph. Any two nodes in V are said to be *adjacent* iff there is an link between them in E. Associated with each pair (x, y) of adjacent nodes in the graph is a positive real number $C.x.y$ called the *link cost* from node x to node y. In general, $C.x.y \neq C.y.x$.

A *path* from node x to node y is a sequence of nodes such that the first node in the sequence is x, the last node is y, every pair of consecutive nodes in the sequences are adjacent and no node appears more than once in the sequence. The *cost* of a path $(w, x, y, ...)$ is the sum $(C.w.x + C.x.y + ...)$. A *shortest-path* from node x to node y is a path from x to y whose cost is less than or equal to the cost of any path from x to y. We are now ready to state the distributed routing problem in computer networks.

Distributed Routing Problem: Let z denote a distinguished node in V called the destination node, and let *Limit* denote a positive real number. It is required to design for each node x in V a process that reads and writes two private variables, $d.x$ and $sc.x$, such that

- The process for node x can read the link cost $C.x.y$ for each adjacent node y.
- Processes for different nodes can communicate through their private variables: each process can read but not write the private variables of its adjacent processes.
- Eventually, the process for each node x is required to set its private variables to the following values:
  $d.x = min$ $\{Limit$, the cost of a shortest-path from x to destination vertex $z\}$

$sc.x =$ the second node in a shortest-path from x to destination vertex z,

provided that $d.x < Limit$. $\square$

A set of processes that solves the distributed routing problem is usually referred to as a *routing protocol*.

Two comments concerning our statement of the distributed routing protocol are in order. First, fail-stop failures and of links and nodes and their subsequent repairs can be accommodated into the problem. In particular, the failure of a link can be "simulated" by resetting both the costs associated with it to *Limit*. Similarly, the failure of a node can be simulated by resetting the costs of all links incident at that node to *Limit*. Link and node repairs can be simulated in the opposite way. Second, for simplicity, our statement of the problem assumes only one destination node, z. It is straightforward, however, to extend any solution of this problem to one in which every node in the graph is a destination node.

2.1 Programming Notation

We adopt the following notation for our solutions. A process has the form

$$\textbf{begin} \quad \langle \text{action} \rangle \; [\!] \; \ldots \; [\!] \; \langle \text{action} \rangle \; \textbf{end}$$

where each action is of the form

$$\langle \text{guard} \rangle \quad \longrightarrow \quad \langle \text{assignment statement} \rangle \; .$$

A guard is a boolean expression over the private variables of the process and the private variables of an adjacent process. An assignment statement updates one or more private variables of the process.

The operational semantics for a protocol that consists of such processes is as follows. A *protocol state* is defined by a value for each private variable in each process. An action in a process is said to be *enabled* at a protocol state if its guard is true in that protocol state. A *fixed point* is a protocol state at which no action is enabled. A set Q of protocol states is *closed* iff for each state q in Q and for each action that is enabled at q, execution of the assignment statement of the action at s yields a state in Q.

An ordered pair of protocol states (p, q) is a *protocol transition* iff there exists an action in some process of the protocol such that the action is enabled at p and execution of its assignment statement at p yields q. A *computation* is a maximal sequence of protocol states such that every consecutive pair of states in the sequence is a protocol transition. Maximality of a computation implies that the computation is not the proper prefix of another computation. It follows that each computation is either infinite or its final state is a fixed point of the protocol.

For a protocol S and a set of protocol states Q, S **self-stabilizes to** Q iff every computation of S can be partitioned into a prefix and a non-empty suffix such that no state in the prefix is in Q, and every state in the suffix is in Q. Operationally, S **self-stabilizes to** Q means every computation of S will eventually reach a state in Q, and once such a state is reached, all subsequent states are also in Q (i.e., Q is closed.)

3 A Protocol for Infrequently Changing Topology

The protocol presented in this section, Protocol $\mathcal{A}$, is based on the following idea. Each process maintains a "distance" variable $d.x$ and a "successor" variable $sc.x$, with respect to its corresponding node x. Two actions are associated with the process – one that "corrects" the distance and another that "corrects" the successor. Correcting the distance involves determining the current distance of the successor (i.e, $d.(sc.x)$) and the cost of the edge $C.x.(sc.x)$, while correcting the successor involves finding an adjacent node y such that $d.y+C.x.y$ is strictly less than the current distance. These corrections ensure that, at fixed point, the distance equals the cost of a shortest-path from x to z and the successor is a node that lies next to x in such a path.

In Protocol $\mathcal{A}$, the domain of variable $d.x$, for $x \neq z$, is the set of real numbers.

Variable $d.z$ constantly satisfies $d.z = 0$. Let $N.x$ denote the set of nodes adjacent to x. The domain of variable $sc.x$, for $x \neq z$, is $N.x$. Variable $sc.z$ constantly satisfies $sc.z = z$. Since $d.z$ and $sc.z$ are always correct, the process for node z has no actions. In stating the actions of other processes, we employ for notational convenience a *parameter* variable. An action with a parameter is said to be enabled at a state iff its guard holds at that state for some value of the parameter. Any parameter value for which the guard is enabled can be used for executing such actions.

Protocol $\mathcal{A}$:
Process $x : (x \neq z)$
parameter $y : N.x$;
var $d.x$: **real**;
$sc.x : N.x$;

begin

$d.x \neq min\,\{d.(sc.x) + C.x.(sc.x)\,,\, Limit\} \quad \rightarrow \quad d.x := min\,\{d.(sc.x)+C.x.(sc.x)\,,\, Limit\}$

$[\!]$

$d.x > d.y+C.x.y \quad \rightarrow \quad d.x, sc.x := d.y+C.x.y, y$

end

Observe that execution of the second action always lowers the distance. Execution of the first action either lowers the distance or raises it to a value not exceeding *Limit*. Bounding the increases serves two purposes. First, it avoids the problem of counting-to-infinity [G89] in which distance values are repeatedly bumped up due to the existence of a cycle of successors. Second, it also ensures that the distance is eventually set to *Limit* if the sum of the link costs in a shortest path exceeds *Limit*. The proof of these facts, as well as the proof that the protocol converges to a correct fixed point inspite of changes in link costs, and node / link failures and repairs, is presented next.

Proof of Correctness.
For our proofs, we first introduce some concepts.

Let $SP.x$ denote the minimum of *Limit* and the cost of a shortest-path between node x and node z. Let $L.x$ for node x be the minimum number of links in any shortest-path between x and z; that is, if we consider the set of shortest-paths between x and z, $L.x$ denotes the minimum of the lengths of paths in this set.

Let *sc–graph* be the directed graph with node set V and arc set $\{(v, sc.v) \mid v \in V \wedge v \neq z\}$. Informally, the sc-graph can be viewed as the structure defined by the sc-variables. Observe that each node of the sc-graph has exactly one outgoing link. Therefore, each node in the sc-graph is either in a directed cycle, on a directed path leading to a directed cycle, or on a path leading to node z. Thus, an sc-graph consists of a set of $sc-cycles$ and $sc-paths$. The set of sc-paths leading to (and including) node z is called an $sc-tree$. Note that every sc-graph contains an sc-tree (the sc-tree may consist of the single node z).

Theorem 1 Every fixed point of $\mathcal{A}$ satisfies $(\forall x :\ x \in V :\ d.x = SP.x)$.

The full text of the proof is in the Appendix. The proof has three parts. First, we prove that each d-variable is at most the cost of a shortest-path to z. Second, we show that the sc-tree contains all nodes whose d-variable value is less than *Limit*. Third, we use the sc-tree to prove that each d-variable contains the cost of a path to z. Together, the first and third parts of the proof establish the conclusion. □

Theorem 2 Protocol A self-stabilizes to $(\forall x :\ x \in V :\ d.x = SP.x)$.

Observe that any state that satisfies $(\forall x :\ x \in V :\ d.x = SP.x)$ is a fixed point. Now, by Theorem 1, it suffices to show that every computation of

$\mathcal{A}$ has a fixed point. We prove this by contradiction. Suppose that $\mathcal{A}$ has some non-terminating computations. We claim that in any non-terminating computation of $\mathcal{A}$, variable $d.x$, for some node x, increases without bound. (The proof of this claim is presented in the Appendix.) However, in the (two) actions that assign to $d.x$, it is either the case that $d.x$ is assigned a value that is at most $Limit$ or $d.x$ decreases. The contradiction follows. $\square$

4 A Protocol for Frequently Changing Topology

The protocol presented in this section, Protocol $\mathcal{B}$, is based on the technique of diffusing computations. This technique was first introduced by Dijkstra and Scholten [DS80] for detecting termination in a class of distributed systems. More recently, Garcia-Luna-Aceves showed that diffusing computations can be used in order to unify different approaches to the routing problem [G89]. However, previous work in this area has been restricted to systems that start in a predesignated initial state; i.e, systems that do not self-stabilize. In contrast, the diffusing computations of Protocol $\mathcal{B}$ are self-stabilizing.

Operation of Protocol $\mathcal{B}$ consists of a number of sessions. After some initial, finite number of sessions, the protocol self-stabilizes in a manner which guarantees that every subsequent session correctly computes the shortest-path costs and successors upon termination; that is, the final state of each subsequent session satisfies, for each $x \in V$, $d.x = SP.x$ and $sc.x$ is the node next to x in some shortest path from x to z.

Once self-stabilization has occurred, the behavior of the protocol is as follows. Each session proceeds in two stages. The first stage begins with the destination node being "active" while all other nodes are "passive". Passive nodes become active upon detecting an active adjacent node, in which case they assign the activating node to their successor variable; it is convenient to think of such changes from passive to active mode as comprising an activation "wave" that spreads through the system. Note that the structure created by the successor relation of the active notes is always tree. Once a node is active, it can change its successor to another active node provided that a better distance approximation is established via the new successor.

Each time that its successor variable $sc.x$ is assigned a new value, node x starts waiting on all its adjacent nodes. Intuitively, this waiting can terminate only when two conditions are satisfied. First, all descendents of x in the sc-tree should have terminated their wait. And second, any other node, y, that is adjacent to x should have a distance value that betters $d.x+C.y.x$. The reader will note that the leaf nodes of the tree are the first to terminate waiting; in fact, one can think of changes from the waiting to non-waiting mode as comprising a "termination wave" that propagates from the leaf nodes in the sc-tree to the root node z. This wave constitutes the second stage.

To describe the protocol formally, we introduce variable sets sn and w in addition to variable sets d and sc. For node $x \in V$, $sn.x$ contains the "session number" of node x. The variable $w.x$ is a vector of $|N.x|$ booleans: for any node $x \in V$ and any $y \in N.x$, the variable $w.x.y$ is either *true* or *false*. We interpret $w.x.y \equiv true$ to mean that process x "waits for" process y. For convenience, we use vector notation $\widehat{w}.x$ to refer to a conjunction of booleans associated with a particular node. Our aim is to interpret $\widehat{w}.x \equiv false$ to mean that process x is not waiting on any process. To ensure self-stabilization, not all nodes adjacent to x are included in the definition of $\widehat{w}$:

$$(\widehat{w}.x \equiv b) \;\equiv\; (\forall y : \; y \in N.x \;\wedge\; y \neq z \;\wedge\; y \neq sc.x : \; w.x.y \equiv b).$$

A similar notation is used for assignment: the post-condition of the assignment $\widehat{w}.x := b$ is $\widehat{w}.x \equiv b$. This post-condition also holds in multiple assignments, in particular, statements that assign to both $\widehat{w}.x$ and $sc.x$.

Protocol $\mathcal{B}$ specifies actions for all processes:

Protocol $\mathcal{B}$:

Process z

parameter $y : N.z$;

var $sn.z$: **integer**;

$w.z.y$: **Boolean**;

begin

$(\widehat{w}.z \equiv \mathit{false}) \quad \rightarrow \quad sn.z,\ \widehat{w}.z \ := \ sn.z+1,\ \mathit{true}$

$[\!]$

$(w.z.y \equiv \mathit{true} \ \wedge \ sn.y > sn.z) \ \vee$

$(w.z.y \equiv \mathit{true} \ \wedge \ sn.y = sn.z \ \wedge$

$((sc.y \neq z \ \wedge \ d.y \leq \mathit{min}\,\{C.y.z, \mathit{Limit}\}) \ \vee$

$(sc.y = z \ \wedge \ \widehat{w}.y \equiv \mathit{false}))) \quad \rightarrow \quad w.z.y := \mathit{false}$

end

Process $x : (x \neq z)$

parameter $y : N.x$

var $d.x$: **real**;

$sc.x : N.x$;

$sn.x$: **integer**;

$w.x.y$: **Boolean**;

begin

$(w.x.y \equiv \mathit{true} \ \wedge \ sn.y > sn.x) \ \vee$

$(w.x.y \equiv \mathit{true} \ \wedge \ sn.y = sn.x \ \wedge$

$((sc.y \neq x \ \wedge \ d.y \leq \mathit{min}\,\{d.x + C.y.x, \mathit{Limit}\}) \ \vee$

$(sc.y = x \ \wedge \ \widehat{w}.y \equiv \mathit{false} \ \wedge$

$d.y \leq \mathit{min}\,\{d.x + C.y.x, \mathit{Limit}\}))) \quad \rightarrow \quad w.x.y := \mathit{false}$

$[\!]$

$(sn.x > sn.y) \ \vee$

$(sn.x = sn.y \ \wedge \ d.x > \mathit{min}\,\{d.y + C.x.y, \mathit{Limit}\}) \quad \rightarrow \quad sn.x,\ sc.x,\ d.x,\ \widehat{w}.x \ :=$

$sn.y,\ y,\ \mathit{min}\,\{d.y + C.x.y, \mathit{Limit}\},\ \mathit{true}$

end

The first action for process z initiates a session; the second action detects "termination" of a session. For reference in proofs we call the first action of process z the *initiation action* ; instances of the initiation action in a computation are called *initiation transitions.* For process x, $x \neq z$, the first action is for distributed termination detection; by the second action, process x either joins a new session or computes lower cost.

The normal situation for Protocol $\mathcal{B}$ is a state in which process z has the maximum session number (i.e., $sn.z \geq sn.x$ for all $x \in V$). Further, to initiate a new session, we expect $sn.z > sn.x$ for all $x \in V$. The following lemmas show that any computation of $\mathcal{B}$ has an infinite number of initiation transitions.

Lemma 1
Every computation of Protocol $\mathcal{B}$ contains an initiation transition.

The proof is by contradiction; the full text of the proof appears in the Appendix. □

Lemma 2
Every computation of Protocol $\mathcal{B}$ contains a state that satisfies

$$\widehat{w}.z \equiv \text{true} \;\wedge\; (\forall\, x\, (x \in V \;\wedge\; x \neq z) \;\;\Rightarrow\;\; sn.z > sn.x).$$

Proof. By Lemma 1, every computation contains an initiation transition. Let τ be some computation and let (s,t) be an initiation transition in τ. Consider the suffix, also a computation, that begins with state t. By Lemma 1, this suffix contains an initiation transition.

This argument may be repeated any number of times; thereby we conclude that every computation contains an infinite number of initiation transitions. The initiation action is the only action capable of increasing the maximum sn-value (i.e., $\mathbf{max}\, x \,:\, x \in V \,:\, sn.x$); no action decreases the maximum sn-value. Consequently, after a finite number of initiation transitions, the maximum sn-value is increased. The result of such a transition proves the conclusion. □

We use the result of Lemma 2 to further characterize states at which a new session is initiated. At such states, for each process x, the value of $d.x$ is the cost of a shortest-path from x to z.

Theorem 3
Every computation of Protocol $\mathcal{B}$ contains a state that satisfies

$$(\forall\, x\; x \in V \quad \Rightarrow \quad d.x = SP.x).$$

The proof is in the Appendix. $\Box$

Finally, the result of Theorem 3 is the basis for an infinite computation in which, for every state and for each process x, the value of $d.x$ is the cost of a path from x to z.

Theorem 4
Every computation of Protocol $\mathcal{B}$ has a suffix β satisfying
$\forall\, s :\; s \in \beta :\; (\forall\, x :\; x \in V \quad \Rightarrow \quad d.x(s)$ is the cost of a path from x to z)).

The proof is in the Appendix. $\Box$

5 Comparison of Protocols $\mathcal{A}$ and $\mathcal{B}$

In Protocol $\mathcal{A}$, convergence to a fixed point may occur slowly. Note that changes in edge costs can result in the formation of cycles in the *sc*-graph. These cycles may persist for a long time. We illustrate this with a simple example. Consider a scenario in which the *sc*-graph has a cycle of two nodes, x and y. Let $d.x=0, d.y=1$, and $C.x.y=C.y.x=1$. Now, if the first actions of the processes for x and y are repeatedly executed in alternation, $d.x$ and $d.y$ are alternately increased by 2. This "bouncing-effect" continues until either $d.x$ or $d.y$ is assigned the value *Limit*.

In contrast, once a computation of Protocol $\mathcal{B}$ has stabilized, the *sc*-graph is a tree for the rest of the computation, even if edge costs changes. (See Theorem 4.) In fact, the time taken to complete a session depends only on the maximum height of the *sc*-tree during the session. However, in a session, the distance and successor computed need not always correspond to a shortest path to the destination. Hence, this protocol is best suited to networks in which topology changes occur frequently and it is desired that routing tables are changed quickly, even if suboptimal routes are computed occasionally. On the other hand, this method is unsuitable for routing purposes in a scenario where changes in the network topology occur infrequently, and consequently, establishing a shortest path is imperative even if time consuming. Finally, Protocol $\mathcal{B}$ incurs unnecessary waiting overhead in situations where it changes the *sc*-tree even when the topology is unchanged.

6 Protocol Composition

In this paper, adaptivity and self-stabilization are the primary properties of protocols and computations. Constituent protocols with suitable adaptivity properties can be combined, using a composition operator, to form composite protocols that retain adaptivity properties. The definitions presented in this section are drawn from the paper [GH89], which presents a logic for reasoning about and construction of protocols with adaptivity properties.

To define adaptivity, we introduce notation for an input predicate. A predicate P is an *input predicate* iff P does not refer to any variables that are assigned by any process action; that is, P is a predicate over the "read only" variables of the processes. We assert that P is an input predicate with respect to protocol S by the notation: **input** P **in** S.

Adaptivity is defined by a relation between a pair of predicates. Given protocol S, and predicates P and Q, the adaptivity relation is defined as follows:

$$P \ \textbf{secures} \ Q \ \textbf{in} \ S$$

iff **input** P **in** S and for every computation τ of S such that P holds at each state in τ, there exists a non-empty suffix γ of τ such that Q holds at each state in γ.

Operationally, P **secures** Q **in** S says that if P holds continuously during execution of S, then eventually Q also holds continuously. If $P \equiv true$ then the situation resembles self-stabilization: from an arbitrary initial state, execution of the protocol leads to Q holding continuously. The following inference rule is proved in [GH89]:

$$\frac{S \textbf{ self-stabilizes to } Q}{\mathit{true} \textbf{ secures } Q \textbf{ in } S}$$

Given two protocols S and T, and a boolean variable e that does not appear in the processes of either protocol, we write $S[e]T$ to denote the following composition operation: $S[e]T$ is the protocol whose process actions are obtained by conjoining "$e \wedge$" to the guards of the process actions of S and "$\neg e \wedge$" to the guards of the process actions of T. The set of variables of $S[e]T$ is the union of the variables of S and T and $\{e\}$ (we assume that any variable named by both S and T has the same domain in both protocols). For example, if S is the single action $x > 0 \rightarrow x := 0$ and T is the action $x \leq 0 \rightarrow x := 1$, then $S[e]T$ is the protocol

$$\begin{array}{ll} & (e \equiv \mathit{true}) \;\wedge\; (x > 0) \quad \rightarrow \quad x := 0, \\ [\!] & \\ & (e \equiv \mathit{false}) \;\wedge\; (x \leq 0) \quad \rightarrow \quad x := 1. \end{array}$$

Clearly, if the variable e is *true* throughout a computation of $S[e]T$, then protocol $S[e]T$ behaves exactly like protocols S. In particular, if S and T are self-stabilizing protocols, we have the composition rule

$$\frac{S \textbf{ self-stabilizes to } Q, \quad T \textbf{ self-stabilizes to } P}{\begin{array}{c}(e \equiv \mathit{true}) \textbf{ secures } Q \textbf{ in } S[e]T, \\ (e \equiv \mathit{false}) \textbf{ secures } P \textbf{ in } S[e]T.\end{array}}$$

The composite protocol $S[e]T$, is unfit for distributed implementation because a single variable e is used in all actions; however, it is straightforward to extend the composition operator for distributed implementation. Let $S.v$ denote the set of actions of S located at process v, and let $T.v$ be similarly defined. We use a set of boolean variables, $e.v$, $v \in V$, to make the composite protocol $S.v[e.v]T.v$ for process v. We extend the notation $S[e]T$ to mean the protocol obtained by the set of compositions $S.v[e.v]T.v$, for all $v \in V$. The new composition rule is

$$\frac{S \textbf{ self-stabilizes to } Q, \quad T \textbf{ self-stabilizes to } P}{\begin{array}{c}(\forall v : v \in V : e.v \equiv \mathit{true}) \textbf{ secures } Q \textbf{ in } S[e]T, \\ (\forall v : v \in V : e.v \equiv \mathit{false}) \textbf{ secures } P \textbf{ in } S[e]T.\end{array}}$$

We conclude this section with the remark that the problem of switching between constituent protocols (i.e., setting the $e.v$ variables appropriately) is an instance of a more general problem that we solved in [AG90].

7 Conclusion

The contribution of this paper is two-fold. Firstly, a new self-stabilizing routing protocol is presented. To the best of our knowledge, this is the first self-stabilizing shortest-path solution to appear in the literature. Unlike several previously presented protocols, our protocol has a wide domain of applicability that does not depend on the frequency of changes in the network topology.

The second contribution is methodological. We have used self-stabilization as a basis for achieving adaptivity in protocols. Furthermore, we have employed a compositional method. The method enables designs in which not only does an executing protocol adapt to changes in input parameters but, also, the protocol being executed is adapted to current network characteristics. Lastly, the method preserves the self-stabilization properties of the protocols being combined by it.

Indirect benefits of self-stabilization include higher reliability. Thus, our protocols can tolerate transient failures, fail-stop failures of nodes and links as well as their subsequent repairs, and the loss of coordination.

References

[AG90] A. Arora and M.G. Gouda, "Distributed Reset", to appear in *Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, (Dec. 1990).

[AGH90] A. Arora, M.G. Gouda, and T. Herman, "Composite Routing Protocols", in preparation.

[BGW89] G.M. Brown, M.G. Gouda, and C.-L. Wu, "Token systems that self-stabilize," *IEEE Transactions on Computers*, Vol. 38, No. 6, pp. 845-852, 1989.

[D74] E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control", *Communications of the ACM*, 17(11) (1974), pp. 643-644.

[DS80] E.W. Dijkstra and C.S. Scholten, "Termination detection for diffusing computations", *Information Processing Letters*, 11(1) (1980), pp. 1-4.

[G89] J.J. Garcia-Luna-Aceves, "A unified approach to loop-free routing using distance vector or link states", *Communications Architectures and Protocols, SIGCOMM 1989*, pp. 212-213.

[GH89] M.G. Gouda and T. Herman, "Adaptive Programming", UTCS Technical Report (august 89), The University of Texas at Austin; also submitted for journal publication.

[QRR80] J. McQuillan, I. Richer, and E.C. Rosen, "The new routing algorithm for the ARPANET, "*IEEE Transcations on Communications*, Vol 28 (1980).

8 Appendix

8.1 Proof of Theorem 1

Observe that $d.x \leq \mathit{Limit}$ holds by contradiction (the first action is not enabled at a fixed point).

Claim 1. For any fixed point of $\mathcal{A}$, for each $x \in V$, $d.x \leq SP.x$.
The proof is by induction on $L.x$.

Basis: $L.x = 0$. $d.z = 0$ trivially satisfies the basis.
Induction: $L.x = k$, $k > 0$. By definition of $L.x$, node x is adjacent to some node y satisfying $L.y = k-1$. By the inductive hypothesis, $d.y \leq SP.y$. Since the second action is not enabled, $d.x \leq d.y + C.x.y$. These observations hold for any y that is adjacent to x and satisfies $L.y = k-1$; therefore

$$d.x \leq \mathbf{min}\,\{SP.y + C.x.y \mid y \in N.x \;\wedge\; L.y = k-1\}.$$

Since $d.x \leq \mathit{Limit}$ holds, it now follows that $d.x \leq SP.x$. (end of Claim 1).

Claim 2. At any fixed point of $\mathcal{A}$, for each $x \in V$, if $d.x < \mathit{Limit}$ then x is in the sc-tree.

The proof is by contradiction. Suppose there is a node x such that $d.x < \mathit{Limit}$ and x is not in the sc-tree. From the fact that no action is enabled it follows that, $d.x = \mathit{min}\,\{d.(sc.x) + C.x.(sc.x)\,,\, \mathit{Limit}\}$. Now $d.x < \mathit{Limit}$, and therefore $d.x = d.(sc.x) + C.x.(sc.x)$ holds. In turn we can conclude that $d.(sc.x) < \mathit{Limit}$. The same argument can be repeated to show that there is an sc-cycle such that for each node y in the sc-cycle, $d.y = d.(sc.y) + C.y.(sc.y)$ and $d.y < \mathit{Limit}$ holds. Let w be some node in this sc-cycle. By transitivity, following the cycle, $d.w = c + d.w$ for some c that is the sum of positive costs; hence we have a contradiction. (end of Claim 2).

Claim 3. At any fixed point of $\mathcal{A}$, for each $x \in V$, the value $d.x$ is the cost of a path from node x to node z.

For each node x not in the sc-tree, $d.x = \mathit{Limit}$ follows from Claim 2 and the fact that $d.x \leq \mathit{Limit}$. Also, by Claim 1, $d.x \leq SP.x$ holds, and by definition of $SP.x$, $SP.x \leq Limit$ holds. Consequently, $d.x = SP.x$ is true at any fixed point of $\mathcal{A}$.

Next, for each node x in the sc-tree, $d.x = min\,\{d.(sc.x) + C.x.(sc.x)\,,\, Limit\}$ holds as no action is enabled. By induction on the length of the sc-path to node z, it follows that $d.x$, for all x in the sc-tree is the cost of the sc-path from node x to node z. (end of Claim 3). $\square$

8.2 Proof of Theorem 2

The remaining proof obligation is to verify the following claim.

Claim: In any non-terminating computation of $\mathcal{A}$, variable $d.x$, for some node x, increases without bound.

Proof of Claim. We define a relation between nodes with respect to a transition. For a transition $t = (r, s)$ of $\mathcal{A}$, let $x \rhd_t y$ hold iff $x \neq y$ and state s is obtained from state r by an action of $\mathcal{A}$ that assigns to variable $d.x$ and makes reference to variable $d.y$. Observe that every action of $\mathcal{A}$ assigns to one d-variable based on the value of another d-variable. Thus, for each transition t, exactly one pair of nodes satisfies the $\rhd_t$ relation.

Consider some non-terminating computation of $\mathcal{A}$; there is a suffix β of the non-terminating computation that satisfies:

$$(\forall t, x, y :\ (t \in \beta \ \wedge\ x \rhd_t y) \ \Rightarrow\ (\exists u, z :\ u \text{ follows } t \text{ in } \beta :\ x \rhd_u z)).$$

An operational view of β is that every process that executes in β does so infinitely often. Let K be the set of nodes for which a d-variable is assigned by some transition in β :

$$K \ \equiv\ \{x \mid (\exists t, y :\ t \in \beta \ \wedge\ x \rhd_t y)\}.$$

Let $d.x(s)$ denote the value of variable $d.x$ at state s. We make the following definitions to construct a variant function on states of β.

$$\begin{aligned}
Dmin.s \ &\equiv\ \mathbf{min}\ (\ \{d.x(s) \mid x \in K\} \cup \{Limit\}\ ) \\
Nmin.s \ &\equiv\ |\ \{x \mid x \in K \ \wedge\ Dmin.s = d.x(s)\}\ | \\
\#s \ &\equiv\ (\ Dmin.s,\ |K| - Nmin.s\ )
\end{aligned}$$

The variant function assigns to state s the pair of numbers $\#s$; comparison between such pairs is the lexicographic ordering over pairs of numbers. The rest of the proof is divided into four assertions.

Claim 1. Let α be a suffix of β; then

$$(\exists t, r, s :\ t \in \alpha \ \wedge \ t = (r, s) \ \wedge \ \#r \neq \#s).$$

Proof by contradiction. Assume

$$(\forall t, r, s :\ t \in \alpha \ \wedge \ t = (r, s) \ \wedge \ \#r = \#s).$$

Therefore $Dmin.q$ is constant for all states $q \in \alpha$. Consider some node x, $x \in K$, satisfying $d.x(q) = Dmin.q$. There exists no transition in α that assigns to $d.x$ because $d.x(q)$ is constant for all states q. This contradicts the definition of K. (end of Claim 1).

Claim 2.

$$(\forall t, r, s, x, y :$$
$$(t \in \beta \wedge t{=}(r, s) \wedge x \rhd_t y \wedge \#r{>}\#s \wedge d.x(s){\neq}Limit) \ \Rightarrow \ y \notin K).$$

Proof by contradiction. Suppose we have some transition $t = (r, s)$, $x \rhd_t y$, satisfying $\#r > \#s$, $d.x(s){\neq}Limit$ and $y \in K$. Observe that $y \in K$ implies $d.y(r) \geq Dmin.r$, therefore $d.x(s) > Dmin.r$, which contradicts $\#r > \#s$. (end of Claim 2).

Claim 3. Let c be an arbitrarily chosen real constant. If α, a suffix of β, satisfies

$$(\forall t, r, s :\ (t \in \alpha \ \wedge \ t = (r, s)) \ \Rightarrow \ \#r \leq \#s),$$

then there exists a state $q \in \alpha$ such that $d.x(q) > c$ for some node x.

Proof. Claim 1 proves that the variant function $\#$ changes infinitely often in any suffix of β. The antecedent of Claim 3 specifies that, in each transition in α, the value assigned by $\#$ can only increase. From the definition of $\#$, after at most $|K|-1$ increases in the variant function, the minimum d-variable value increases. Therefore an infinite number of variant function increases implies that the minimum d-variable value exceeds c at some state in α. (end of Claim 3).

Claim 4. There exists α, a suffix of β, that satisfies

$$(\forall t, r, s : \ (t \in \alpha \ \wedge \ t = (r, s)) \ \Rightarrow \ \#r \leq \#s).$$

Proof by contradiction. Suppose, for any α, suffix of β,

$$(\exists t, r, s : \ (t \in \alpha \ \wedge \ t = (r, s)) \ \Rightarrow \ \#r > \#s).$$

Consider the set of transitions in β in which the value assigned by # decreases. Of these transitions, we choose a transition, t, that assigns the smallest value to the d-variable of any node in K. That is, let t be a transition satisfying $t \in \beta$, $t = (r, s)$, $x \rhd_t y$, $\#r > \#s$, and

$$\begin{aligned}(\forall u, p, q, v, w : \ (u \in \beta \ \wedge \ u = (p, q) \ \wedge \ v \rhd_u w \ \wedge \ \#p > \#q)\\ \Rightarrow \ d.x(s) \leq d.v(q) \ ).\end{aligned}$$

Let δ be a suffix of β with initial state s, that is, δ is the suffix originating after transition t. Since $\#r > \#s$, it follows that $Dmin.s = d.x(s)$. Observe also, that the value $d.x(s)$ is at most the $Dmin$ value for all states in δ (t was chosen to have precisely this property). There are two cases to consider. If $d.x(s) = Limit$, then no transition in δ assigns to $d.x$. Else, if $d.x(s) \neq Limit$, then, by claim 2 we have that $y \notin K$. Again, no transition in δ can assign to $d.x$. This contradicts the definition of K because $x \in K$ implies that every suffix of β contains a transition that assigns to $d.x$. (end of Claim 4).

The existence of a suffix specified by Claim 4 satisfies the antecendant of Claim 3 and thereby completes our proof obligation. □

8.3 Proof of Lemma 3

For reference in proofs, we reformulate the actions of $\mathcal{C}$ as the following labeled list, which ranges over $x \in V$ and $y \in N.x$:

$$
\begin{array}{ll}
0: & w.x.y \;\wedge\; sn.y > sn.x \\
 & \rightarrow\; w.x.y := \mathit{false} \\[1ex]
1: & w.x.y \;\wedge\; sn.y = sn.x \;\wedge\; sc.y \neq x \;\wedge\; d.y \leq min\,\{d.x + C.y.x, Limit\} \\
 & \rightarrow\; w.x.y := \mathit{false} \\[1ex]
2: & w.x.y \;\wedge\; sn.y = sn.x \;\wedge\; sc.y = x \;\wedge\; \widehat{w}.y \equiv \mathit{false} \;\wedge\; d.y \leq min\,\{d.x + C.y.x, Limit\} \\
 & \rightarrow\; w.x.y := \mathit{false} \\[1ex]
3: & x \neq z \;\wedge\; sn.y > sn.x \\
 & \rightarrow\; sn.x,\; sc.x,\; d.x,\; \widehat{w}.x \;:=\; sn.y,\; y,\; min\,\{d.y + C.y.x, Limit\},\; \mathit{true} \\[1ex]
4: & x \neq z \;\wedge\; sn.y = sn.x \;\wedge\; d.x > min\,\{d.y + C.x.y, Limit\} \\
 & \rightarrow\; sn.x,\; sc.x,\; d.x,\; \widehat{w}.x \;:=\; sn.y,\; y,\; min\,\{d.y + C.x.y, Limit\},\; \mathit{true} \\[1ex]
5: & x = z \;\wedge\; \widehat{w}.x \equiv \mathit{false} \\
 & \rightarrow\; sn.x,\; \widehat{w}.x \;:=\; sn.x + 1,\; \mathit{true}
\end{array}
$$

The proof obligation for Lemma 3 is to show that every computation of $\mathcal{C}$ contains a 5-transition.

The proof is by contradiction: we show that any computation without a 5-transition has a fixed point and that action 5 is enabled at that fixed point. Let τ be some computation that has no 5-transition. To prove that τ is finite we make the following definitions.

$$
\begin{array}{lcl}
kw & = & \mathbf{max}\,\{sn.x \mid x \in V\} \\
rw & = & \sum\,\{kw - sn.x \mid x \in V\} \\
dw & = & \sum\,\{d.x \mid x \in V\} \\
nw & = & \mid \{(x,y) \mid (x,y) \in E \;\wedge\; w.x.y\} \mid
\end{array}
$$

($\sum$ returns the sum of the elements in a set.) Observe that kw is constant for all states in τ since action 5 is the only action capable of increasing the maximum sn-variable.

Consider the evaluation of the triple (rw, dw, nw) for each state in τ. We claim that lexicographic order between triples is a decreasing variant function on the states of τ. The claim can be verified by examining each action (except action 5) and showing that the action's output state yields a triple that is lexicographically smaller than the triple for the input state. We leave this verification to the reader. Observe that (0,0,0) is the minimum possible triple. Further, it can be shown that each reduction of some value in a triple reduces that value at least by some fixed decrement, so the number of reductions to a triple is finite before a minimum is reached (the fixed limit for reduction is necessary since cost is a real number; we could not show finiteness if infinitesimally small reductions were possible). Hence, τ is finite, that is, τ has a fixed point at which no action 0-4 is enabled.

Let q be the final state of τ. To show that action 5 is enabled at q we need to show that $\widehat{w}.z \equiv \mathit{false}$. To prove this we show a more general property of q. Let S denote the sc-tree in state q. To complete the lemma, we prove the

$$\textbf{sublemma}: \quad (\forall\, x, y: \ (x \in S \ \wedge \ y \in N.x) \ \Rightarrow \ \ w.x.y \equiv \mathit{false}).$$

Proof. Consider some $x \in S$ and $y \in N.x$. Observe that, at state q, $sn.y < sn.x$ does not hold since action 3 is not enabled at process y. Observe that $sn.y > sn.x$ implies that $w.x.y \equiv \mathit{false}$ because action 0 is disabled at process x. Therefore we suppose $sn.y = sn.x$. We complete the proof by induction on the sc-tree, S.

Basis. For the basis, let y satisfy $sc.y \neq x$. Observe that if x is a leaf of the sc-tree then any adjacent node y satisfies this condition. Consider two cases for $d.y$: If $d.y \leq min\,\{d.x + C.y.x, \mathit{Limit}\}$ then, because action 1 is disabled at process x, we have $w.x.y \equiv \mathit{false}$. And $d.y > min\,\{d.x + C.y.x, \mathit{Limit}\}$ is impossible since action 4 is disabled at process y.

Induction. Let y satisfy $sc.y = x$, that is, y is a child of x in the sc-tree. By the inductive hypothesis, it follows that $\widehat{w}.y \equiv \mathit{false}$. Because action 2 is disabled at process x, we have $w.x.y \equiv \mathit{false}$. (end of sublemma).

Thus we have shown that every computation that does not contain a 5-transition is finite and that $\widehat{w}.z \equiv \mathit{true}$ in the final state of the computation, whereat action 5 is enabled. This contradiction proves the lemma. $\square$

8.4 Proof of Theorem 3

Let s be a state that satisfies the conclusion of Lemma 2. We show that any computation that begins with state s contains a state t satisfying the proof requirement. Further, state t is the first state where a 5-transition is enabled, and in fact, the 5-transition is the only transition enabled at state t. For the remainder of the proof let R be the set $\{x \mid sn.x = sn.z\}$. To characterize state t, we introduce six propositions.

- (i) $(\forall x: \ sn.z \geq sn.x)$.
- (ii) R is an sc-tree.
- (iii) For any node x, $x \in R$, the variable $d.x$ contains the cost of some path from x to z.
- (iv) $(\forall x, y: \ (x \in R \ \wedge \ y \in N.x \ \wedge \ \neg w.x.y) \ \Rightarrow \ y \in R)$.
- (v) $(\forall x, y: (x \in R \ \wedge \ y \in N.x \ \wedge \ \neg w.x.y \ \wedge \ y \in R \ \wedge \ sc.y = x) \ \Rightarrow$
  $\widehat{w}.y \equiv \mathit{false})$.
- (vi) $(\forall x, y: \ (x \in R \ \wedge \ y \in N.x \ \wedge \ \neg w.x.y \ \wedge \ y \in R) \ \Rightarrow$
  $d.y \leq \mathit{min}\,\{d.x{+}C.y.x, \mathit{Limit}\})$.

We prove that each proposition (i)-(vi) holds at every state in any computation beginning with state s. The proof is by structural induction.

Basis for the induction is state s, which trivially satisfies (i)-(vi) : for state s, $R = \{z\}$, and verification is immediate. To complete the induction, we examine each proposition and verify that no transition can falsify that proposition.

Proposition (i) cannot be falsified by any action, since $sn.z$ contains the maximum sn-value, which is increased only by a 5-transition. (Note that the result of a 5-transition satisfies (i)-(vi) by the same argument we used for the induction's basis).

We verify (ii) and (iii) jointly by examining actions 3 and 4, since no other action can falsify (ii)-(iii).

First, consider the result of a 3-transition; we consider this result for a process x with respect to (ii) and (iii). Hence we consider the case where $sn.x$ increases. If the result of the 3-transition satisfies $sn.x \neq sn.z$, then $x \notin R$ and (ii) and (iii) hold trivially. If $sn.x = sn.y$, then (ii) and (iii) hold by induction for the multiple assignment of action 3.

Second, consider the result of a 4-transition. Since (ii) and (iii) hold prior to the 4-transition, it follows transitively that every descendent w of x in the sc-tree R satisfies

$$d.w \leq min\,\{d.x{+}c, Limit\}$$

for some $c > 0$. Furthermore, because (iii) holds prior to the 4-transition, $d.x \leq Limit$. Therefore action 4 is not enabled if y is a descendent of x in R. So, in a 4-transition, y is not a descendent of x in R; in this case the 4-transition can be viewed as removing the arc $(x, sc.x)$ from R, adding arc (x, y) to R, and assigning to $d.x$ to establish (ii) and (iii).

To verify (iv) we consider the effect of each type of transition. There are two cases to consider for the falsification of (iv), where the process variables change by a transition corresponding to y or to x.

First, consider the case of assignment to y. If it is by a 3-transition establishing $y \in R$, then $w.x.y$ is true since, by induction, (iv) holds prior to the transition; thereby (iv) is not falsified. Suppose $y \notin R$ is established by some transition: the invariance of (i) assures that $y \in R$ is falsified only by a 5-transition, for which (iv) is verified by the basis argument for state s.

Second, consider the case of assignment to x. If $x \in R$ is established by a transition, then $\widehat{w}.x \equiv true$ holds and (iv) is not falsified. By the invariance of (i), if $x \notin R$ is established we appeal to the basis argument. Therefore we suppose $x \in R$ holds prior to and after the transition, and actions 0, 1, 2 and 4 require inspection. A 0-transition is impossible by the invariance of (i); a 1-transition does not falsify (iv) since $y \in R$ is implied by action 1's guard; a 2-transition does not falsify (iv) since $y \in R$ is implied by action 2's guard; and a 4-transition establishes $\widehat{w}.x \equiv true$, which does not falsify (iv).

Verification of (v) and (vi) is similar to verification of (iv) : we consider cases for transitions corresponding to y or to x.

First, consider the case of assignment to y. If $y \notin R$ is established by a transition, we appeal to the basis argument for state s. If $y \in R$ is established then $w.x.y$ is false by the invariance of (iv), therefore neither (v) nor (vi) are falsified. We therefore suppose $y \in R$ holds prior to and after the transition. Hence we consider the result of actions 1, 2 and 4. A 1-transition and a 2-transition do not falsify (vi). A 1-transition and a 2-transition do not falsify (v), since by inductive hypothesis, (v) holds prior to such a transition — if $\widehat{w}.y \equiv false$ is established by the transition, then $w.x.y \equiv true$ holds before and after the transition. Let us examine a 4-transition for process y, where $y \in N.x$. Here, the invariance of (vi) holds trivially. To see that (v) is invariant, assume $\neg w.x.y$; by the inductive hypothesis for (vi),

$$d.y \leq min\,\{d.x + C.y.x, Limit\}$$

and action 4 is not enabled.

Second, consider the case of assignment to x. If $x \in R$ is established as a result of a transition, then $\widehat{w}.x \equiv true$ holds and (v) and (vi) are verified. If $x \in R$ holds prior to and after a transition, we consider any transition that establishes $\neg w.x.y$ or changes variable $d.x$. If $d.x$ is changed, then $\widehat{w}.x \equiv true$ holds by either a 3-transition or a 4-transition and (v) and (vi) are verified. Suppose $\neg w.x.y$ is established; we therefore look at actions 1 and 2. A 1-transition does not falsify (v) since $sc.y \neq x$ for a 1-transition; a 1-transition verifies (vi) by its guard. A 2-transition verifies (v) and (vi) by its guard.

Having established the invariance of (i)-(vi) we can complete the proof as follows. By Lemma 2 there is a state t satisfying $\widehat{w}.z \equiv false$ in any computation beginning with state s. Therefore, by (ii), (iv), (v) and transitivity, $(\forall x : x \in R \wedge \widehat{w}.x \equiv \mathit{false})$. Consequently (vi) can be simplified for state t to

$$(vii) \qquad (\forall x, y : \ (y \in N.x) \ \Rightarrow \ d.y \leq \mathit{min}\,\{d.x + C.y.x, \mathit{Limit}\}).$$

The final step of the proof is to show, by induction on $L.x$, that $d.x = SP.x$.

Basis. $L.x = 0$. $d.z = 0$ trivially satisfies the basis.

Induction. Assume $(\forall x : L.x < k \ \Rightarrow \ d.x = SP.x)$. Let $R \equiv \{x \mid L.x = k-1\}$ and let $S \equiv \{x \mid L.x = k\}$. Each $x \in S$ satisfies

$$(\exists y : \ y \in R \ \wedge \ (x, y) \in E \ \wedge \ SP.x = SP.y + C.x.y).$$

Observe by (vii) that $d.x \leq \mathit{min}\,\{d.x + C.y.x, \mathit{Limit}\}$ for some $y \in R$. By (iii), $d.x \neq SP.x$ contradicts $L.x = k$; thus the induction step is proved. □

8.5 Proof of Theorem 4

The proof is by structural induction. The basis for the induction is the state t identified in the proof of Theorem 3: at state t every variable $d.x$ has the cost of a path from x to z. The remainder of the proof follows the form of the induction step in the proof of Theorem 3, and details are omitted. □