Self-Stabilizing Topology Maintenance Protocols for High-Speed Networks*

Hosame Abu-Amara† Brian A. Coan‡ Shlomi Dolev§ Arkady Kanevsky¶
Jennifer L. Welch‖

Abstract

Two self-stabilizing topology maintenance protocols for high speed networks are presented. The protocols tolerate any number and kind of initial faults. The new protocols improve on previous protocols by their *stabilization time* (the amount of time following the last topology change required to notify every processor of the correct topology), by their utilization of limited switch bandwidth, and by their avoiding the use of unbounded sequence numbers.

The first protocol stabilizes in $O(\log d)$ time in the worst case, where d is the diameter of the network. This protocol imposes a high bandwidth requirement on individual network nodes. The second, which is implemented by two software layers, reduces the processing load on individual nodes and stabilizes within $O(d)$ time in the worst case and $O(1)$ time when changes are infrequent.

*An extended abstract of this paper appeared as [AC+94a].

†Department of Electrical and Computer Engineering, University of Nevada, Las Vegas, NV 89154, email: amara@ee.unlv.edu.

‡Bellcore, Morristown, NJ 07960, email: coan@thumper.bellcore.com.

§Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, Israel 84105, email: dolev@cs.bgu.ac.il.

¶Department of Computer Science and Statistics, University of Southern Mississippi; and MITRE Corp., Bedford, MA 01730-1420, email: arkady@mitre.org.

‖Department of Computer Science, Texas A&M University, College Station, TX 77843, email: welch@cs.tamu.edu.

1 Introduction

The high bandwidth of today's fiber-optic technology suggests that processing time may become the main bottleneck in future computer networks. Recently, extensive research has considered the design of fast and primitive hardware switches to eliminate the processing delay. A hardware switch is placed at each node and handles message routing without involving the processor at the node. The result is systems whose behavior differs from that of traditional networks in some important ways: (1) if the entire path is known, the message delay between any two nodes in the network can be considered a constant (one time unit), no matter how many intermediate nodes must be traversed; (2) broadcast can also be accomplished in $O(1)$ time units (e.g. [AA93]).

The design of high-speed networks and protocols has been addressed in many recent papers (e.g., [CG88, CGK88, AC+90, OY90]). [CG88, CGK88, AC+90] present a high-speed network design called PARIS. To obtain a simple hardware switch the design of PARIS uses source routing and tree broadcast for messages. Hence, each processor has to know the current topology (i.e., what links are currently functioning) in order to send messages. Thus, maintaining a correct view of the topology is a core task for such high-speed networks.

Fault-tolerance is especially important for on-going tasks such as distributed network control which can suffer from transient faults of any nature. A useful type of fault tolerance is the *self-stabilizing* property [Dij74]; it guarantees that regardless of the initial state of the system, once each processor begins executing its program correctly, the system converges to a "correct" state. An early topology maintenance protocol designed for the PARIS model [CGK88] is (or at least can be made) self-stabilizing. This protocol stabilizes within $O(d \cdot \log n)$ time, where d is the diameter of the surviving network and n is the number of processors in the surviving network. Abu-Amara [AA93] presented a protocol for the same model that stabilizes within $O(d)$ time.

The *switch bandwidth* is the number of messages a switch can handle at a time. Note that it possible that the number of messages a switch can handle at a time is a function of their arrival links and destination links; to compute the switch bandwidth we consider the worst case of arrival and destination links that yields the minimal number of messages that can be handled at a time. In the protocols of [CGK88] and [AA93] each processor repeatedly broadcasts its view of the topology, implying that $\Theta(n)$ broadcasts can take place simultaneously Therefore, the switch and link bandwidth required by these protocols is $\Omega(n)$.

In [AC+90] the control protocols of the PARIS experimental network are described. Although not stated, we believe that upon non-faulty operation the switch bandwidth required is $O(\Delta)$, where Δ is the highest degree of a switching node. The topology maintenance protocol presented in [AC+90] is event driven, i.e., it is activated when some change in the available link bandwidth is detected. This protocol has a cyclic organization: the topology maintenance module broadcasts over the tree computed by the tree maintenance module and the tree maintenance module uses the result of the topology module in order to compute the tree. This approach is vulnerable to transient faults, which could result from an unfortunate combination

of processor crashes, link failures and recovery, message loss and corruption, and unexpected delays. The behavior of the protocol is not determined when either the topology view or the tree is corrupted (e.g., the "tree" contains cycles). Thus the protocol is not self-stabilizing.

A different approach for the design of high-speed networks is presented in [OY90]. The design is based on having a ring embedded in a general topology. The ring can be obtained by traversing a spanning tree of the network. This approach yields a protocol that only works in the restrictive "one fault model." This protocol does not make use of a topology maintenance protocol and thus does not tolerate general dynamic changes.

A topology change is a change of the status of a link or a processor (from up to down or vice versa). In this paper we present methods to provide fast communication in a high-speed network in the presence of faults, in particular when the topology changes dynamically. We present two self-stabilizing topology maintenance protocols. Our first topology maintenance protocol assumes $O(n)$ bandwidth for a switch and stabilizes in $O(\log d)$ time in the worst case. Its stabilization time compares favorably with the $O(d)$ of the best previously known protocol [AA93]. The second protocol stabilizes in $O(d)$ time in the worst case and $O(1)$ time when changes are infrequent and assumes $O(\Delta)$ bandwidth for a switch[1], where Δ is the maximum degree of a node in the network. Figure 1 presents a comparison with related work. Note that the time complexity for [AC+90] is actually for an execution beginning with a legitimate initial state, since the protocol is not self-stabilizing. Our second topology maintenance protocol uses the existence of a clock to optimize its stabilization time after a single topology change.

	Worst Case Stabilization Time	Stabilization Time after a Single Topology Change	Switch Bandwidth	Is It Self-Stabilizing?
[CGK88]	$O(d \cdot \log n)$	$O(\log n)$	$O(n)$	Yes
[AA93]	$O(d)$	$O(1)$	$O(n)$	Yes
[AC+90]	$O(n)$	$O(1)$	$O(\Delta)$	No
Protocol 1	$O(\log d)$	$O(\log d)$	$O(n)$	Yes
Protocol 2	$O(d)$	$O(1)$	$O(\Delta)$	Yes

Figure 1: Comparison of Topology Maintenance Protocols

In our first self-stabilizing topology maintenance protocol, which we call the doubling protocol, each processor repeatedly uses its partially correct view of the topology to help all nodes in the network converge to a correct view of the current topology. Roughly speaking, each processor attempts to broadcast to all other processors using its own, possibly incorrect, view

[1] One can distinguish between (1) messages that are transferred from an input port to an output port of the switch and (2) messages that are transferred only to the processor (and not to an output port). While messages of type (1) cannot be queued (to avoid delays and complicated switches) messages of type (2) can be delivered to the processor queue. When we only count the messages of type (1) that a switch has to handle simultaneously, then $O(1)$ (instead of $O(\Delta)$) switch bandwidth is required.

of the topology. The broadcast will reach some processors and may fail to reach others. A processor P believes information that P receives from another processor Q concerning particular nodes and links if Q is in the middle of a shortest path to the node or link in question. This protocol utilizes the power of high-speed networks to achieve a stabilization time significantly better than the $\Theta(d)$ required in traditional network models.

Our second topology maintenance protocol, which we call the optimistic-with-backup protocol, has the pleasing property that it requires limited switch bandwidth. This protocol is obtained by combining two layers. The bottom layer is a self-stabilizing point-to-point topology maintenance protocol (e.g., [SG89, Do93]) which is permanently executed in the "background." After this slow topology maintenance protocol has stabilized, links can go up or down. One way to maintain the topology information would be to simply wait for the slow protocol to stabilize again. Instead we use a layer on top of the bottom layer to quickly adjust the topology view. The top layer is fast, but it is not self-stabilizing and can make mistakes if changes occur too frequently[2]. The mistakes of the top layer are corrected through the systematic use of the results of the slower bottom layer.

The bottom layer uses only communication between neighbors and hence limits the number of topology maintenance messages received by a node in a single time unit to be the degree Δ of the node. A priority scheme ensures that the messages of the bottom layer are never discarded due to congestion. Every message includes few bits in its header that specify the priority of the message. When an overloaded switching subsystem simultaneously receives messages with different priorities, the switching subsystem transfers the messages with the highest priority and, it if is overloaded, discards messages with lower priorities.

Note that in addition to the communication traffic of the topology maintenance protocol (which is part of the network control), messages are sent by the application layer. Obviously, the application layer uses topology information to communicate data (including video and voice) through the network. If the topology information is incorrect and the topology maintenance is blocked by the traffic of the application layer, then the level of service of the communication network can be drastically reduced. The priority scheme is especially important in the context of self-stabilizing protocols where the system can be started with an application layer that use the entire bandwidth. Therefore, if congestion occurs due to simultaneous arrival of messages, then messages of the application layer are the first to be discarded. Then messages of the top layer are discarded before any message of the bottom layer is discarded. The priority scheme is used to ensure delivery of the messages of the bottom layer and to guarantee progress.

When no topology change occurs for a long enough time, every processor knows the correct topology of the system. The goal of the top layer is to achieve fast updates when the topology changes are not too frequent. The top layer uses the fast communication capabilities of the network hardware in order to try to *foresee* the result of the bottom layer, and therefore to

[2]The approach of dealing differently with frequent and infrequent events has been proposed in a different context in, e.g., [La87] and [AM91].

achieve faster response time, thus eliminating the effect of temporarily inconsistent states that might be produced while the slower bottom layer is stabilizing.

The top layer uses the result of the bottom layer initially to discover the topology and periodically as a fault-tolerant back-up mechanism.

Because two layers are attempting to keep track of the topology, there must be a rule for selecting which results to use. Each processor takes the results of the topology foresee layer if that layer has recently indicated a topology change. Otherwise, it takes the results of the bottom layer. The argument for self-stabilization is that in a sufficiently long interval with no changes, the foresee layer will eventually stop sending messages, the bottom layer will eventually stabilize to the true topology, and all processors will eventually choose to believe the results of the bottom layer.

The remainder of the paper is organized as follows. In the next section we describe the model for the high-speed network. Section 3 contains the first self-stabilizing topology maintenance protocol, which converges in $O(\log d)$ time in the worst case. Section 4 contains the two-layer topology maintenance protocol, which stabilizes in $O(d)$ time in the worst case and $O(1)$ time when changes are infrequent. Concluding remarks appear in Section 5.

2 High-Speed Networks

Consider a network of n processors. We model the network as a graph of n nodes, in which each node represents a processor and each link represents a bidirectional communication channel. Henceforth, we will not distinguish between a node and the processor it represents, and we will not distinguish between a link and the channel it represents.

We use the model proposed in [CG88, CGK88, AC+90]. Each node consists of a *switching subsystem* and a *node control unit*. The *switching subsystem* of each node contains the communication hardware responsible for receiving, sending, and forwarding messages. All of the functions of the switching subsystem are implemented in hardware. Thus, the switching subsystem performs its duties very rapidly. Because of the hardware implementation, however, the switching subsystem is not able to make decisions that are based on the information content of the messages. Rather, the switching subsystem is able to examine only the header of messages. The *node control unit* of each processor P contains the processing hardware and software necessary to extract the information content of messages, do some internal computation, and generate messages to be forwarded to other nodes via P's switching subsystem. Most of the functions of the node control unit are implemented in software. Thus, the node control unit may be slow and may require the presence of message queues to buffer the messages forwarded by the switching subsystem.

For each processor P, each attached end-point of a link is assigned a small set of link labels. The set of labels assigned to each end-point of a link may be dynamically configured (under software control). The header of a message consists of a sequence of at most $2N$ *link labels* and

a *copy bit*, where N is an upper bound on the number of processors in the network. A node control unit P sends a message to another node control unit Q by attaching a header with the labels of the links on the route from P to Q. P may also send a copy of the message to every processor in the route to Q by setting the *copy bit* of the message.

When a message m arrives at a switching subsystem and the header of m contains at least one label, then the switching subsystem removes the first label, l, and the shortened message is sent on all links that (1) are not the link that the message arrived on and (2) have l as one of its labels. Additionally, when the copy bit is set, a copy of the message is sent to the node control unit. Otherwise, when there is no link label left, m is delivered to the node control unit.

Figure 2 illustrates a 4-node network and a possible message. If the indicated message is inserted into the network by the node control unit at node 1, it will be routed through the switching subsystems at nodes 2, 3, and 4, with one copy delivered to each node control unit. If, on the other hand, the message is inserted by the node control unit at node 2, it will be discarded by the switching subsystem at that node because the first label in the route does not match any of the labels on the incident links.

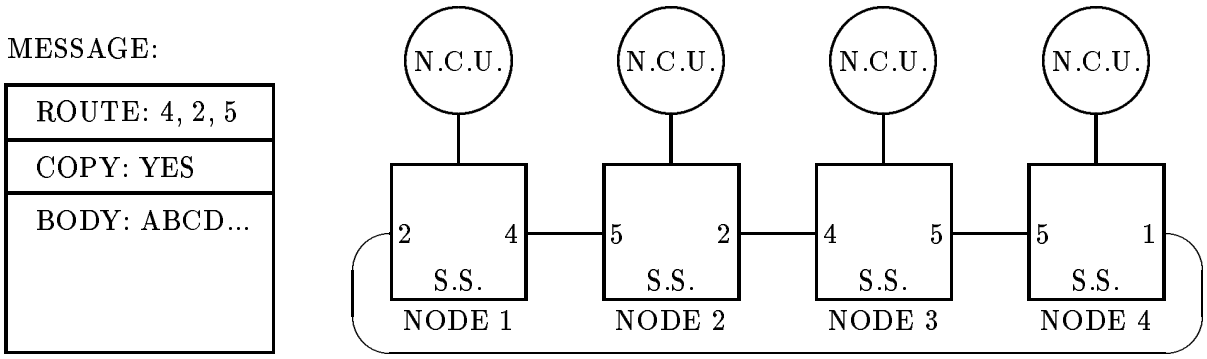

Figure 2: A Message and a Network of Four Nodes

In this paper, we assume the set of labels for each link consists of a single unique label within the switching subsystem. In our model the functions of the switching subsystem are implemented in hardware and no message queues are used. Messages simultaneously arriving at a switch might be forwarded to the same outgoing link of the switch. However, if too many messages arrive at the same time in some switching subsystem, the switch may not be able to handle them all, and some of the messages will be lost. Hence, the number of messages a switching subsystem can handle at the same time, which we call the *bandwidth* of the switch, is an important parameter of the design. This parameter is not captured by the formal model presented in [CGK88]. Thus many protocols developed for that model (including that presented in Section 3 of this paper) tend to swamp switches with messages. The protocol presented in Section 4 of this paper is designed to work with switches that have a limit on the number of

messages that they can process simultaneously. Each message has a *priority level*. In case the number of messages arriving at a switch is greater than its capacity, the messages with the lowest priority are eliminated first. The correct operation of the topology maintenance protocol in a heavily loaded network requires that its messages be of the highest priority. It is the protocol's responsibility to ensure that the number of messages with highest priority that simultaneously reach a switching subsystem will not exceed the switch bandwidth.

Due to the above network architecture, it is assumed that the message delay, measured in real time, is bounded by one *time unit*. Thus, the total real time it takes to initiate a message delivery by the node control unit, to communicate the message to the destinations switching subsystem and then from the switching subsystem to the node control unit, takes one time unit. If the process of sending m starts at real time t then the data part of m (m without the header) is received by all of its destinations before real time $t+1$. Moreover, due to the "label removal" technique and the limitation of at most $2N$ link labels in a single message, m (and all the messages obtained by "label removals" from m) vanishes from the system before real time $t+1$.

Each processor P always knows P's *local topology*, i.e., all the names of its currently operational neighbors and the labels of the outgoing links that connect the switching subsystem of P to its neighbors. Since processors and links may fail at any time, no processor can be sure that it has the correct graph of the full network topology. We assume that P can test the status (faulty/non-faulty) of each link incident on it (cf., [Taj77, MRR80, CGK88]) This assumption is reasonable in current networks where P periodically can test an incident link by sending a test message on the link and waiting for a response. In a totally asynchronous environment a processor cannot detect whether a link is faulty or the neighbor at the other side is very slow. Thus, real-time clocks are used for local topology detection. We do not include the time to perform these tests in the analysis of our protocol.

Each processor has a local clock that may exhibit bounded drift from real time: at least $1/(1+\rho)$ (real) time units and at most $1+\rho$ (real) time units of the clock elapse in any real time unit. Where ρ is the *drift rate*. The local clocks are not synchronized; the clocks are used only to measure time intervals, e.g., message delay time.

The node control unit of a processor may send messages at approximately every time unit[3]. However, since the clocks are not synchronized processors may send messages at different times rather than simultaneously. A *round* consists of at least one successful message delivery by each (non-crashed) processor. These messages are either sent through one attached link or a copy on each attached link according to the protocol. A round terminates when each of these messages arrives at its destination. The *round complexity* is the number of rounds initiated during the execution. Because clocks are not synchronized and can drift, it is possible for a round to include more than one message sent by a given processor. In the sequel we sometimes use round complexity (instead of time units) to measure time complexity. Note that if each processor sends messages approximately every time unit and no message is lost , the fact that

[3]The approximation is due to the clock drift.

the clock rates are bounded implies an upper bound on the amount of real time that it takes to accomplish a round.

For simplicity, we assume that if a node fails, then the switching subsystem that resides in the same site fails too[4]. The amount of information that a message may contain may vary from system to system. Clearly, if the number of bits that need to be transferred is larger than the capacity of a message, the information can be sent in a sequence of messages, with each message carrying a portion of the information. Standard mechanisms can be used to eliminate out of order delivery of the information portions, e.g., using the same path for all the portions, using bounded sequence numbers (taking into account the lifetime of a message), or using timestamps. Keeping this possibility in mind, we assume, for the sake of readability, that a single message can carry $\Theta(n\Delta \log n)$ bits; this number of bits is sufficient for a complete topology description.

3 The Doubling Protocol

Cidon, Gopal, and Kutten [CGK88] presented a topology maintenance protocol for high-speed networks. In this protocol, each processor periodically performs the following. It broadcasts its local view (list of neighbors with which it can currently communicate), by using a breadth-first broadcast protocol. (*Breadth-first* means that all processors at distance x from the source are visited before any processor at distance $x+1$ is visited.) Upon receiving such messages, the processor uses them to update its global view of the network. (A processor's global view is the list of links and processors that it currently believes to be functioning correctly.)

The protocol of [CGK88] is proven correct by a simple inductive argument. Specifically, it is proven by induction on i, the number of rounds, that after i rounds of broadcast following the last topology change, each processor knows the true network state of all components at distance at most $i+1$ from itself.

The breadth-first broadcast protocol presented in [CGK88] takes $O(\log n)$ time per broadcast. Thus, the time needed for all processors to construct a correct view of the topology with this protocol is $O(d \cdot \log n)$, where d is the diameter of the network.

Subsequently, Abu-Amara [AA93] proved that there is a way to do a breadth-first broadcast in such a network that terminates in time $O(1)$. Roughly speaking, the broadcast of [AA93] traverses a tree by first traversing the closest nodes. Let $l_1, l_2, \cdots, l_i$ be the tree links connected to the broadcast initiator. A message is sent from the broadcast initiator to a neighbor, Q, through l_1 then the message returns to the initiator through l_1 and is forwarded to another neighbor through l_2 and so on until the message is forwarded by the initiator through l_i to a neighbor R. A similar forwarding procedure to traverse the children of R in the tree is used and then the message is forwarded back to the initiator. Next the message is forwarded through

[4]If a node can fail while its switching subsystem does not, then still information concerning the faulty node is needed by the topology maintenance protocol. We believe that our protocols can be adapted to handle efficiently such rare cases too, not only by literally stopping the operation of the switching subsystem of a faulty processor.

l_{i-1} to the neighbor S and to the children of S. This technique repeats itself until all the N labels of the message are eliminated. Note that the message contains topology information that defines the spanning tree used by the initiator. Thus, processors that received the message can deduce from the contents of the message that they are roots of subtrees in which no processor received a broadcast message. These processors produce messages to continue the broadcast in their subtrees in a similar fashion. It is proved that the broadcast is completed within 4 time units. By substituting this broadcast protocol into the [CGK88] protocol, the stabilization time is improved to $O(d)$.

Our new protocol further reduces the stabilization time of this approach by doubling a processor's field of view at each iteration. In more detail, each processor, P, does the following periodically. It broadcasts its current (possibly wrong) global view, $\mathcal{V}_P$, using the [AA93] breadth-first broadcast based on $\mathcal{V}_P$. Upon receiving such a message, say from processor Q, processor P uses it to update its global view. Let R be a processor in $\mathcal{V}_P \cup \mathcal{V}_Q$. The information about processor R is updated due to the received $\mathcal{V}_Q$, if (1) the distance, x, from P to Q according to $\mathcal{V}_P$ and the distance, y, from Q to R according to (the received) $\mathcal{V}_Q$ are equal (or differ by one) and (2) the total distance $x + y$ is less than the distance from P to R according to $\mathcal{V}_P$. Rule (2) disallows an update that would increase the distance from P to R according to $\mathcal{V}_P$. Such a rule cannot guarantee correct operation because it does not allow updating the topology view when "bad news" arrives—namely, increase of the distance to a certain processor. One example for this limitation is when the system is started in an initial state where the distance of every processor from P according to $\mathcal{V}_P$ is 1 (such an initial state is possible when we deal with self-stabilizing systems). Another example is when $\mathcal{V}_P$ is up to date and a topology change that increases the distance to R occurs. Thus, we need some mechanism to ensure the validity of the current (small) distance. Our choice is to recall the identity of the processor, say Q, that is responsible for the last update and to check that this processor repeatedly "stands behind its word" concerning the information on R. Namely, Q must be in the middle of the shortest path from P to R. If this does not hold, we remove the information concerning R from $\mathcal{V}_P$.

The code for the protocol appears in Figure 3. We use the following convention for the representation of the topology view, $\mathcal{V}_P$, of a processor P: $\mathcal{V}_P$ is a set with (up to) N elements. Each element contains the fields $\langle id, dis, source, links \rangle$ which respectively represent, processor identifier, say Q, the distance from P to Q, the source for the information about Q, and (up to) Δ processor identifiers, an identifier (of a neighbor of Q) for every active link of Q. We use $\mathcal{V}_P[Q]$ for the element with processor identifier Q. $\mathcal{V}_P[Q].dis$ is the distance field of the element $\mathcal{V}_P[Q]$ if such an element exists and ∞ otherwise. During the computation of an updated topology view we use the set, $\mathcal{TV}_P$, to store temporary view. $\mathcal{TV}_P$ contains up to $2N$ elements.

The code of a processor P starts with a broadcast of the current topology view $\mathcal{V}_P$. Then $\mathcal{V}_P$ is updated whenever another topology view, $\mathcal{V}_Q$, is received. "Good news" received from a processor in the middle is adopted. Then the function Legalize legalizes the information gathered from $\mathcal{V}_P$ and $\mathcal{V}_Q$. Next we describe the operation of the Legalize function (which

does not appear in Fig. 3).

The function Legalize chooses at most N elements of $\mathcal{TV}_P$ and assigns them to $\mathcal{V}_P$. As an initialization step all the elements in $\mathcal{V}_P$ are deleted. Then, the first element to be included in $\mathcal{V}_P$ is $\mathcal{V}_P[P]$ which contains P as an identifier, 0 as a distance, P as source and the list of the active links that are attached to P. In the next step, an element is added for every processor, Q, at distance 1 that is connected to P via an active link. For such an element to be included, the source field should be Q. The process continues in an iterative fashion. At iteration i, elements whose *dis* field value is i in $\mathcal{TV}_P$ are included in $\mathcal{V}_P$. For such an element to be included it must be connected via a path of active links that are already in $\mathcal{V}_P$ and whose source is a processor at distance $\lceil i/2 \rceil$ from P. In addition an element is not included if its information concerning active/non-active links conflicts with information of elements included during previous iterations. The iterative process stops when either $\mathcal{V}_P$ contains elements for N processors or no element is added in the current iteration.

```
1: do forever
2:     begin
3:       every time unit broadcast(V_P)
4:       for every V_Q received do
5:         begin
6:           TV_P := nil
7:           for every processor R ∈ V_P ∪ V_Q do
8:             if V_P[R].dis ≥ V_P[Q] + V_Q[R] and (V_P[Q].dis − V_Q[R].dis) ∈ {0, 1} then
9:               begin
10:                TV_P[R] := V_Q[R]
11:                TV_P[R].dis := V_P[Q].dis + V_Q[R].dis
12:                TV_P[R].source := Q
13:              end
14:            else TV_P[R] := V_P[R]
15:          V_P :=Legalize(TV_P)
16:        end
17:    end
```

Figure 3: Doubling Protocol, Code for P.

We use the definition of *round of broadcast* below in our correctness proof. Before we define *round of broadcast* we should notice that a broadcast can be ended either in a normal or abnormal way. A broadcast ends normally when the topology view used by the initiator of the broadcast reflects the topology of the network. A broadcast ends abnormally when the topology used by the initiator does not reflect the network. In such a case, if the initiator of the broadcast knows the correct topology only up to distance i, then the the broadcast ends (abnormally) when every processor at distance less than or equal to i receives a message of the broadcast.

Definition 3.1 *A* round of broadcast *consists of at least one broadcast by each processor. The round of broadcast terminates when all the initiated broadcasts have ended.*

Note that the protocol of [AA93] implies that a *round of broadcast* consists of $\Theta(1)$ rounds.

Theorem 3.1 *In the doubling protocol, after $2i+2$ rounds of broadcast following the last topology change, for every processor P: (1) $\mathcal{V}_P$ contains the true local topology (active/non-active links) of every processor at distance at most 2^i from P. (2) For every processor Q that is at distance greater than 2^i from P, $\mathcal{V}_P[Q].dis > 2^i$.*

Proof: Following the first execution of the function Legalize it holds that the right local topology of P appears in $\mathcal{V}_P$. Furthermore, there is no element with distance 0 other than the element of P.

The rest of the proof is by induction on i.

Basis: $i = 0$. We prove that after the second round of broadcast, the hypothesis holds. The result follows from the ability of each processor to test the status of each of its incident links and the success of the first broadcast to reach (at least) the closest neighbors. For every neighbor Q of P the information concerning the local topology of Q is broadcast by Q and used by P. Every other broadcast message that is received is either from a non-neighbor of P or has an element for Q with distance greater than 0. This implies (1) of the Lemma. The Legalize procedure implies (2).

Induction: Assume for i and show for $i+1$. By the inductive hypothesis and because of the use of breadth-first broadcast, each processor's round $2i+3$ broadcast reaches processors within distance 2^i.

For every processor R within distance at most 2^{i+1} from P there is at least one processor Q at distance $x \leq 2^i$ from P, such that Q is on a shortest path from P to R, Q is at distance $y \leq 2^i$ from R and $x - y \in \{0, 1\}$. By the induction hypothesis, for every such processor, Q, Q has the right information concerning R's local topology prior to the $2i+3$'rd round of broadcasts.

We first prove that before the $2i+4$'th broadcast round it holds that $\mathcal{V}_P[R].dis$ is not smaller than the distance z between P and R. If $\mathcal{V}_P[R].dis$ is indeed smaller than z then there must be a "dummy" $\mathcal{V}_P[R].source$ say, X, such that in the last execution of Legalize it was verified that $\mathcal{V}_P[X].dis$ is equal to $\lceil \mathcal{V}_P[R].dis/2 \rceil$. Since $\mathcal{V}_P[R].dis$ is less than the actual distance z then $\mathcal{V}_P[R].dis < 2^{i+1}$ and $\lceil \mathcal{V}_P[R].dis/2 \rceil \leq 2^i$. Therefore, by the induction hypothesis, $\mathcal{V}_P[X].dis$ is indeed the distance from P to X. Thus X's broadcast is received by P during the $2i+3$'rd broadcast round. In this round $\mathcal{V}_X$ contains the correct information concerning the status of the links and processors that are at least up to distance 2^i. Hence, if X is still the source for R it reflects the real distance from P to R via X that is smaller than z, a contradiction. By a similar argument no processor within distance $\leq 2^i$ from P can become a "dummy" source following the $2^i + 3$'rd broadcast round.

During the $2i+4$ broadcast round the broadcast of every such Q reaches P and P uses the information concerning R. ■

The stabilization time of our new protocol is $O(\log d)$ in the worst case. The doubling protocol can be used as a building block in the next protocol. This combination will yield $O(d)$ in the worst case and $O(1)$ in the event that all processors have a correct view of the topology and there is a single topology change. Such a combination requires $O(n)$ switch bandwidth.

4 The Optimistic-With-Backup Protocol

The protocol presented in the previous section pinpoints an important limitation of the [CGK88] model. Since every processor repeatedly broadcasts the topology known to it, there can be as many as n broadcasts in every unit of time. Consequently, a switch has to be able to handle $\Theta(n)$ messages at the same time. In many cases the previous protocol will stabilize even when the bandwidth of the switch is small and some messages are lost due to congestion. However, there are scenarios in which message loss can prevent the protocol from stabilizing. In this section we make sure that the bandwidth used by the message traffic in a switch is controlled.

Toward this goal we use a technique with two layers in each of which a different protocol operates. Figure 4 depicts the relationship between the two protocol layers. The bottom layer is a slow self-stabilizing topology maintenance protocol that uses traditional point-to-point communication. Following a topology change the topology view of the bottom layer is inaccurate for some period of time. During this period the top layer tries to foresee the resulting topology view of the bottom layer.

Because there are two layers keeping track of the topology, there must be rules for determining which results to believe. The bottom layer ignores the top layer and performs its calculation of the topology without reference to any computations done in the top layer. At any processor P, the output of the bottom layer is a topology view $\mathcal{V}_P$. The top layer contains two components, a *foreseer* and a *chooser*. The topology view produced by the foreseer is $\mathcal{F}_P$ and the view produced by the chooser is $\mathcal{E}_P$. The chooser computes its output view to be equal to $\mathcal{F}_P$ if the foreseer has recently indicated a topology change by sending a $\mathcal{V}_c$ message; otherwise the chooser computes its output view to be equal to $\mathcal{V}_P$. The output of the chooser, $\mathcal{E}_P$, is the topology view produced by the entire 2-layer protocol and is also used as input by the foreseer. The foreseer communicates with other processors by sending or broadcasting messages in routes that are based on the topology view $\mathcal{E}_P$. This communication happens whenever a topology change is detected. The two purposes of the communication performed by the foreseer are (1) to send $\mathcal{V}_c$ messages to cause the chooser to stop choosing the output of the bottom layer during the time it takes for the bottom layer's view of the topology to become stable and (2) to compute quickly $\mathcal{F}_P$, the foreseen topology. If no indication of a

topology change and no topology foresee message arrives at the foreseer, then the foreseer assigns $\mathcal{F}_P := \mathcal{E}_P$. Otherwise, the foreseer updates $\mathcal{E}_P$ according to the topology change and assigns the updated topology to $\mathcal{F}_P$.

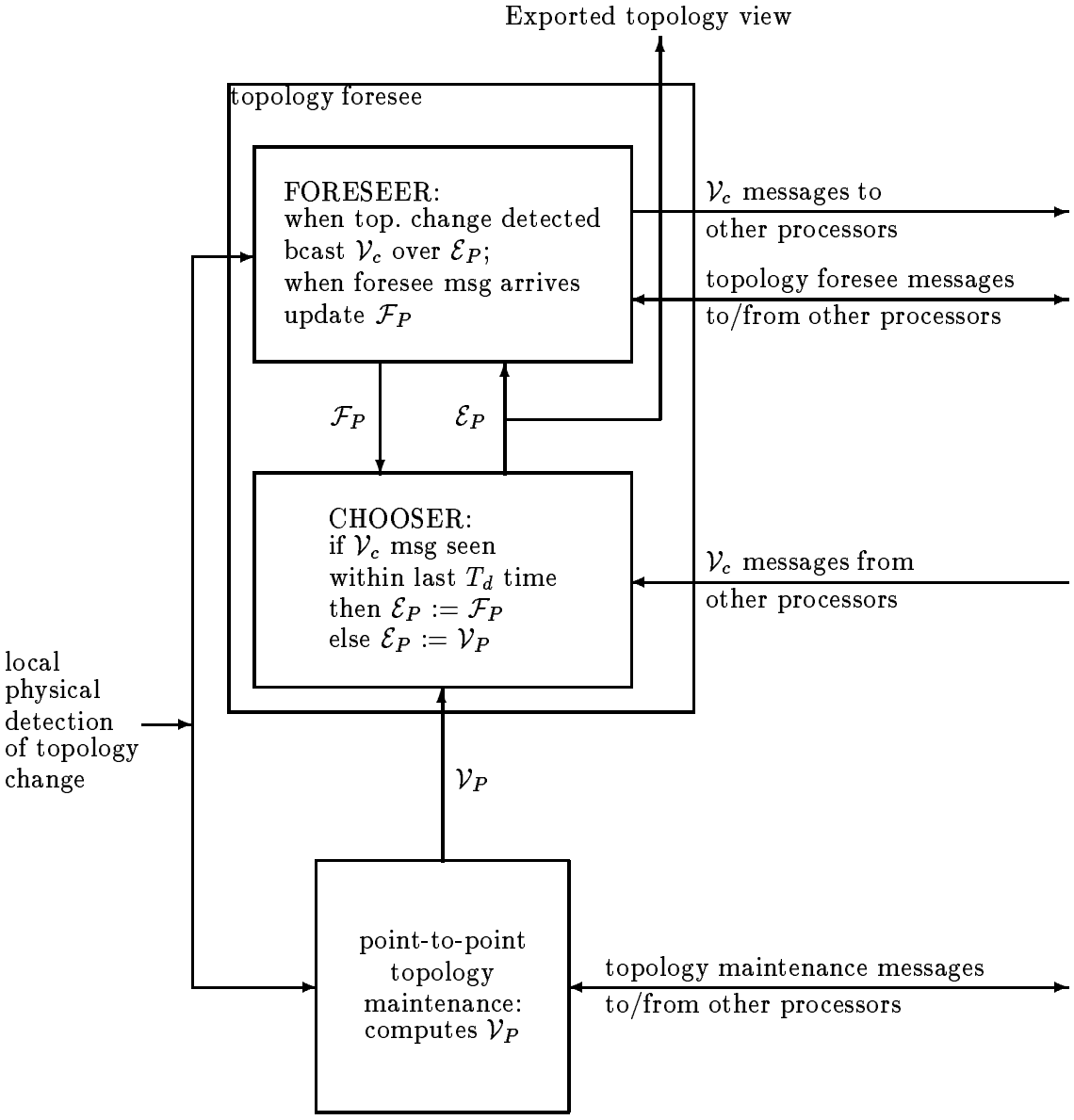

Figure 4: Relationship Between the Two Protocol Layers at P

4.1 Bottom Layer: Point-to-Point Topology Maintenance

For the bottom layer we use a slight modification of the self-stabilizing topology maintenance protocol of [SG89]. [SG89] is chosen for simplicity of presentation. Another possibility is the self-stabilizing topology maintenance protocol of [Do93]. Unlike [SG89], the protocol in [Do93] first determines only the membership relation, i.e., each processor knows the processors that belong to its connected component. Then the information about the links' status is collected over a single tree, which implies a high volume of communication through only $n-1$ links as opposed to $\Theta(n^2)$ links of [SG89].

The modification that we make to the [SG89] protocol is that at each processor a delay (of approximately two time units) is inserted between the time that the processor actually detects a topology change and the time the processor acts on the change. The purpose of this delay is to give the top layer a chance to foresee the new topology before the results of the bottom layer start to change: The bottom layer at the processors that detected a topology change stops its activity until the top layer identifies the nature of the topology change (i.e., a link or processor fails or recovers) and elects a processor to broadcast the new topology. This ensures that the rest of the processors will be notified first with the new topology through the top layer and only then start the convergence of the bottom layer protocol. Thus, the intermediate (incorrect) states of the bottom layer protocol do not affect the topology view of the processors.

In the [SG89] protocol, every processor P maintains its topology view $\mathcal{V}_P$. P repeatedly updates $\mathcal{V}_P$. The update is done according to the local topology and the topology view $\mathcal{V}_Q$ of each neighbor Q. For every neighbor Q, P maintains a local variable that contains the last topology view received from Q; this local variable is updated whenever a message from Q arrives. The current topology view $\mathcal{V}_P$ is defined by the current local topology (i.e., the status of neighboring links) and the last topology view received from P's neighbors (as described later). P repeatedly sends its topology view $\mathcal{V}_P$ to all of its neighbors.

P computes a new value for $\mathcal{V}_P$ whenever P receives a message with a topology from one of its neighbors. The computation of $\mathcal{V}_P$ is by a *local* iterative process similar to the shortest path algorithm of Dijkstra [Dij59]. Note that this iterative process is executed without any communication and thus is assumed to be executed within a negligible time[5]. The iterative process of P uses a topology view variable $\mathcal{V}'_P$. First $\mathcal{V}'_P$ is set equal to P and the links that are connected to P. Then in each iteration $\mathcal{V}'_P$ is extended (if possible) to include the local topology of the processors R that are connected to the outgoing links of $\mathcal{V}'_P$. Thus, in the i-th iteration, P includes in $\mathcal{V}'_P$ the processors that are at distance $i-1$ from P. Each such local topology is taken from the view of the neighbor Q that is closest (in terms of number of hops) to R according to $\mathcal{V}_Q$. In case of ties, i.e., if there are two neighbors of P, Q and Q', such that R is closest to Q (respectively, Q') according to $\mathcal{V}_Q$ (respectively, $\mathcal{V}_{Q'}$), then P includes the local topology of R according to $\mathcal{V}_Q$ if the identifier of Q is larger than the identifier of

[5] Typically, in high-speed networks the actual communication of a message is faster than the processing time. However, the preparations for the delivery of a message is a processing task that involves interaction (using buffers) between the node control unit and the switching subsystem.

Q' (in fact P can arbitrary choose either of the local topologies). The local iterative process stops when no processor is added to $\mathcal{V}'_P$. The value of $\mathcal{V}'_P$ at the end of the iterative process is assigned to $\mathcal{V}_P$, the topology view of P.

The protocol of [SG89] is proven correct by a simple inductive argument similar to what we have already seen in the previous section. Specifically, it is proven by induction on i that i rounds following the last topology change, each processor knows the true network topology of all components at distance at most i from itself. Thus, in terms of rounds as defined in Section 2, the above topology maintenance protocol stabilizes within $O(d)$ rounds. The communication used by this protocol is point-to-point. This ensures the existence of at most one arriving message in any incoming link at any time. Thus, the switch bandwidth used by this protocol is at most $O(\Delta)$.

4.2 Top Layer: Topology Foresee

The drawback of the point-to-point topology maintenance protocol is its response time for topology changes. Each topology change will generally require $O(d)$ time to propagate throughout the network. In fact the bottom layer does not use the power of the high-speed network. We now present a topology foresee layer that utilizes the high-speed property of the network. The combination of the bottom and top layers is a self-stabilizing topology maintenance protocol that stabilizes within $O(1)$ time if topology changes are not too frequent and $O(d)$ time following the last change, otherwise.

We first describe the protocol of the top layer informally. The two components in this layer are the foreseer and the chooser. The chooser at processor P continually picks between $\mathcal{F}_P$, the topology view computed by the foreseer, and $\mathcal{V}_P$, the topology view computed by the bottom layer. It chooses $\mathcal{F}_P$ during periods of time that follow a topology change and $\mathcal{V}_P$ at other times. The view chosen is called $\mathcal{E}_P$.

We define a *clock time unit* to be the quantity $1 + \rho$, a period of clock time large enough to ensure, even for the fastest clock, that at least one (real) time unit has elapsed.

Following a topology change, the processors that are neighboring to the topology change stop sending messages that are related to the bottom layer and use the high-speed capability to notify the processors to stop choosing to assign $\mathcal{V}_P$ to $\mathcal{E}_P$. Then the message delivery of the bottom layer is resumed. As we detail in the sequel, the notification process takes at most two clock time units. The delay in the activity of the bottom layer is essential for the top layer to detect whether the topology change is a link failure/recovery or a processor failure/recovery and to elect a single processor to notify the rest of the processors of the change. (The election of a single processor guarantees limited bandwidth usage upon broadcasting the change.) While the above activities of the top layer are taking place, the bottom layer does not change the topology view. Thus, when the top layer broadcasts the change in the topology, every processor has the topology view prior to the topology change and the information concerning the topology change, which yields the current topology. Then the processors that are neighboring to the

topology change continue executing the traditional point-to-point protocol which starts to stabilize to reflect the new topology. Let t_d be an upper bound on the number of (real) time units required for the point-to-point protocol to stabilize (note that t_d is $O(d)$ since the round complexity of the topology maintenance protocol of [SG89] is $O(d)$). In the period of stabilization that follows this topology change, $\mathcal{V}_P$, the topology view of each processor P, may be changed more than once before it reflects the new topology. Our goal is to eliminate assignment of $\mathcal{V}_P$ into $\mathcal{E}_P$ during this period. Instead the protocol tries to *foresee* the value $\mathcal{F}_P$ that $\mathcal{V}_P$ will have following the stabilization period and assign the value it foresees to $\mathcal{E}_P$.

Whenever a topology change occurs the change is locally detected by the neighboring processors. These processors use their $\mathcal{E}$ variables to broadcast a *change message* $\mathcal{V}_c$ with the topology change. For simplicity and time performance we assume the following broadcast procedure: A processor P uses its topology view to choose a spanning tree of the communication tree (say a breadth-first tree rooted at P). Then a message with copy bit equal to 1 is sent to traverse the tree in a depth-first order (see, e.g., [Ev79] for the definitions of breadth first tree and depth first search). Note that the message traverses each link of the tree at most twice. Thus the number of link traversals experienced by such a message is less than $2N$. As a result, the message traverses the tree in a single time unit.

Upon receiving a change message $\mathcal{V}_c$, a processor Q assigns $\mathcal{F}_Q$ according to the topology reported in $\mathcal{V}_c$, and Q sets a timer with value $T_d = t_d(1+\rho)$. T_d is chosen so that when T_d time has elapsed on the fastest clock, at least t_d real time has elapsed (recall that t_d is the upper bound on the stabilization time of the bottom layer). Q does not refer to $\mathcal{V}_Q$ until at least T_d clock time has elapsed since this timer setting. (If T_d is chosen to be too short, say because the bound on the clock drift is not obeyed, then the top layer might not succeed in foreseeing the topology; however, the combined protocol will eventually stabilize once the bottom layer does so and T_d clock time has elapsed for every processor.)

Ideally, when any two successive topology changes do not occur too close in time, the top layer always foresees the result of the bottom layer, i.e., the correct topology. Thus, the later assignment of $\mathcal{V}_P$ into $\mathcal{E}_P$ will not have any effect. In case the topology changes are too frequent, the top layer may not succeed in maintaining the correct view of the topology. However, even in this case it holds that when there is no topology change for long enough time, every message of the top layer regarding a topology change vanishes and the bottom layer stabilizes; thus every processor eventually has the correct topology.

The detailed description of the way the foresee layer acts upon a single topology change (i.e., a single link/processor failure/recovery) follows. The code appears in Figure 5. Throughout the description it is assumed that $\mathcal{E}_P$ contains the correct topology before the current topology change and that the foresee layer finishes updating the topology before the next topology change occurs (Later we discuss relaxing these assumptions). In general, the processors neighboring to the topology change communicate with other processors to determine the type of topology change that took place (e.g. whether it is a link or a node failure). Once the topology change has been identified by the processors neighboring to the topology change they use $\mathcal{E}_P$ and the identified change to deduce the current topology $\mathcal{E}'_P$. Then (one or more) processors that are

neighboring to the topology change broadcast $\mathcal{E}'_P$; the broadcast procedure itself uses $\mathcal{E}'_P$ as the view of the topology.

• Link (P,Q) or Q recovers: Line 1 of the code checks whether Q was in the connected component of P before the recovery of (P,Q). If Q was not in the connected component of P then (P was not in the connected component of Q either and) P sends the topology view of its component to Q. In case only the link (P,Q) recovers (as opposed to the recovery of Q) P receives the topology view of Q. Then, the one among P and Q, with the greater identifier broadcasts the combined topology. The combined topology is denoted in line 5 of the code by $\mathcal{E}_P + \mathcal{E}_Q$. Line 7 of the code is for the case P and Q were in the same component before the recovery of (P,Q). If P has a larger identifier P broadcasts the updated topology that includes the link (P,Q).

• Link (P,Q) fails or Q fails: A failure of a single processor, Q, can cause all the neighbors, P, of Q to detect approximately at the same time that the link (P,Q) failed. In turn, every neighbor may take action, i.e., send messages to indicate this change, and thus an overflow of the switch bandwidth may occur. Next we show how our protocol eliminates this scenario.

If Q has no other path to P according to $\mathcal{E}_P$, then the network communication graph is disconnected and P cannot discover whether (P,Q) failed or Q failed. The function "connected" executed in line 8 of the code checks whether the topology view of the remaining communication graph is a single connected component. If according to $\mathcal{E}_P$, Q has no other path to P, then P broadcasts the new topology on the portion of the topology in which P resides. This portion is denoted by $(\mathcal{E}_P - (P,Q))|P$ in line 8 of the code — in words, omit (P,Q) from the previous topology $\mathcal{E}_P$ and pick the component in which P resides. Note that if Q did not fail then Q broadcasts the new topology in its connected component according to $\mathcal{E}_Q$.

Otherwise, when Q has at least one other path to P according to $\mathcal{E}_P$, the following procedure is used to distinguish whether (P,Q) failed or Q failed. Let $\mathcal{C}'_P$ be the connected component of P (according to $\mathcal{E}_P$) assuming Q failed, and let $\mathcal{N}$ be the set of Q's neighbors that are in $\mathcal{C}'_P$. In line 11 of the code we use the function neighbor$(X,Q,\mathcal{E}_P)$, which is true if and only if X is a neighbor of Q in $\mathcal{E}_P$. Note that P is in $\mathcal{N}$. If P has either the largest or the smallest identifier among $\mathcal{N}$, then P broadcasts a message "Q failed?" to the nodes in $\mathcal{C}'_P$ (lines 12 and 13). Then (even in the case P does not broadcast) P waits one clock time unit to receive such an inquiring message from a processor in $\mathcal{N}$. If no such message arrives, P concludes that Q is active and the link (P,Q) failed; otherwise P concludes that Q failed.

Note that when Q fails at most two messages are sent in every connected component of the resulting graph. Furthermore, when (P,Q) fails only P and Q may send messages. Thus, we ensure that the switch bandwidth used for this detection is of size of at most two messages.

Following the detection of the type of topology change that took place, i.e., was it a link failure or processor failure, P takes action as follows. If Q failed and P has the largest identifier among the neighbors of Q in P's connected component then P broadcast the new topology in $\mathcal{C}'_P$ (line 16). In case the failure of (P,Q) has been detected and P has a larger identifier than Q, then P broadcasts $\mathcal{E}_P - (P,Q)$ (line 18) using the new topology.

• Processor P recovers: P waits for the neighbors to send their topology views upon detecting the recovery of the links connecting them with P. At this stage it is assumed that $\mathcal{E}_P$ consists of P's local topology, i.e., P and the links to its neighbors. Once P receives the topology views of its neighbors, P combines their topology views and its own local view and broadcasts the combined topology throughout the network.

Next we describe the interaction between the topology foresee layer and the point to point topology maintenance layer. (note that Figure 5 includes only the topology foreseer). For all the above cases, a processor, P, that is neighboring to a topology change does not update $\mathcal{V}_P$ until the broadcast of the new topology should be terminated. Namely, P does not update $\mathcal{V}_P$ for the first $2(1+\rho)^2$ time (measured by its clock) following the detection of the topology change. It is easy to check that, for every topology change the broadcast is done following $(1+\rho)$ time units measured by a processor clock. Thus, at most $(1+\rho)^2$ time units measured by P's clock. The broadcast should arrive to every processor before $(1+\rho)$ additional time units measured by P's clock. This sums up to $(1+\rho)+(1+\rho)^2 < 2(1+\rho)^2$ units that are measured by P's clock. This delay in updating $\mathcal{V}_P$ ensures that a $\mathcal{V}_c$ message arrives at a processor before any (partial) update takes place caused by the (relatively slow) update of the point to point topology maintenance layer.

Whenever a new topology broadcast of the top layer arrives at some processor R, R assigns the new topology to $\mathcal{F}_R$. In particular, the initiator of a broadcast assigns the new topology to $\mathcal{F}_P$.

Up to this point in the protocol description, we have assumed that $\mathcal{E}_P$ is correct when a topology change occurs. When this assumption does not hold, a processor that is incident to a topology change might send wrong messages and/or omit the right messages. Since the switch bandwidth is limited we would like to eliminate unnecessary usage of the switch bandwidth as soon as possible. Consequently, a processor incident to a topology change is restricted to send a topology foresee layer message only during the first $(1+\rho)$ time units following the topology change.

The system is in a *safe* state when it holds that if no further topology change occurs then: (1) $\mathcal{E}_P$ reflects the correct topology, (2) $\mathcal{E}_P = \mathcal{F}_P$ for every P, and (3) no message of the foresee layer is in transit.

Recall that t_d is an upper bound on the number of (real) time units required for the point-to-point protocol to stabilize and is $O(d)$ time units.

Theorem 4.1 *Let $T = (t_d+1)(1+\rho)^2+1$. If no topology change occurs during T real time units, then at the end of this period the system is in a safe state.*

Proof: Fix an interval I of length T time units during which no topology change occurs. Let t be the real time at the beginning of I.

No message of the topology foresee layer is sent after real time $t+(1+\rho)^2$ (recall that a clock time unit is $1+\rho$, which can take as long as $(1+\rho)^2$ real time to elapse). By the

Link (P,Q) recovers or Q recovers:
1: if $Q \notin \mathcal{E}_P$ then {different components}
2: send($\mathcal{E}_P$) to Q
3: wait one clock time unit or until receive(msg,Q)
4: if received($\mathcal{E}_Q$) and $P > Q$ then
5: broadcast($\mathcal{E'}_P = \mathcal{E}_P + \mathcal{E}_Q$)
6: else {same components}
7: if $P > Q$ then broadcast($\mathcal{E'}_P = \mathcal{E}_P + (P,Q)$)

Link (P,Q) fails or Q fails:
8: if connected($\mathcal{E}_P - (P,Q)$) then broadcast($\mathcal{E'}_P = (\mathcal{E}_P - (P,Q))|P$)
9: else { check whether Q failed or (P,Q) failed}
10: $\mathcal{C'}_P := (\mathcal{E}_P - Q)|P$
11: $\mathcal{N} := \{X | neighbor(X, Q, \mathcal{E}_P) \wedge (X \in C'_P)\}$
12: if $P = max(\mathcal{N})$ or $P = min(\mathcal{N})$ then
13: broadcast(C'_P,"Q failed?")
14: wait one clock time unit or until receive("Q failed?")
15: if received("Q failed?") then {Q failed}
16: if $P = max(\mathcal{N})$ then broadcast($\mathcal{C'}_P$)
17: else {Q did not fail}
18: if $P > Q$ then broadcast($\mathcal{E'}_P = \mathcal{E}_P - (P,Q)$)

Processor P recovers:
19: wait one clock time unit or until receive($\mathcal{E}_{Q_i}$) from every neighbor Q_i
20: if for every neighbor Q_i received($\mathcal{E}_{Q_i}$) then
21: broadcast($\mathcal{E'}_P = \mathcal{E}_P + \mathcal{E}_{Q_1} + \mathcal{E}_{Q_2} + \cdots + \mathcal{E}_{Q_j}$)

Figure 5: The Topology Foresee Layer, Code for P

characteristic of the network, no message of the topology foresee layer still exists in the system after another (real) time unit, i.e., after time $t + (1+\rho)^2 + 1$.

Thus the latest real time at which any processor receives a $\mathcal{V}_c$ message is $t + (1+\rho)^2 + 1$. Hence, after real time $t + (1+\rho)^2 + 1$, $\mathcal{F}_P$ is always assigned to be the value of $\mathcal{E}_P$. After a further T_d clock time units elapse, which is at most $T_d(1+\rho)$ real time, every processor subsequently assigns $\mathcal{V}_P$ to $\mathcal{E}_P$. Since $T_d = t_d(1+\rho)$, this means that after real time $t+T$, i.e., the end of I, every processor subsequently assigns $\mathcal{V}_P$ to $\mathcal{E}_P$.

Since the bottom layer stabilizes within t_d time units, every assignment to $\mathcal{E}_P$ after the end of I results in the correct topology view.

The above proves the requirements (1), (2) and (3) for a safe configuration. ■

The proof of the next theorem is straightforward from the description of the protocol.

Theorem 4.2 *When the system is in a safe state and a single topology change occurs then*

the system reaches a safe state within $O(1)$ time following this change.

5 Concluding Remarks

This paper presented two self-stabilizing topology maintenance protocols for high-speed networks.

For a self-stabilizing protocol it is important to avoid sequence numbers; our protocols do not use sequence numbers.

Our first protocol is the first to break the $O(d)$ ($\Theta(d)$ for traditional networks) bound on the stabilization time of topology update protocol. The protocol uses the high-speed capabilities to achieve an exponentially faster protocol.

We addressed an important aspect of the high-speed network—the bandwidth of the switching subsystem. We suggested a priority scheme to ensure progress in spite of congestion. Another contribution of this paper is the new foresee technique which "smoothes" the effect of the stabilization period following a topology change.

One possible optimization is to reduce the communication bandwidth used by the traditional point-to-point topology maintenance protocol by having it send checking information (e.g., check sum and random key) rather than the whole topology. When every processor has the same topology view the check sum (for the chosen key) will be the same for every processor and the topology will not be transferred. When the topology views of two neighbors are different, then within short expected time a key is chosen such that the check sum is different for both processors. This in turn triggers exchange of topology views as done by the traditional point-to-point topology maintenance.

One of the anonymous referees suggested examining whether the topology view of the unchanged nearby portion of the graph converges to the correct view in the presence of frequent remote topology changes. We remark that our doubling protocol as well as the bottom layer of the optimistic with back-up protocol converge to a correct view of the nearby portion of the network, in the presence of frequent remote topology changes.

Our protocols can be used as a building block in the distributed control of high-speed network. Some details on such usage may be found in [AC+94b].

Acknowledgment

We thank Will Leland and three anonymous referees for helpful comments on earlier versions of this paper.

References

[AA93] H. Abu-Amara, "A fast topology maintenance algorithm for high-bandwidth networks," *IEEE/ACM Transactions on Networking* **1**(3), pp. 386–394, June 1993.

[AC+94a] H. Abu-Amara, B. Coan, S. Dolev, A. Kanevsky and J. Welch, "A fault-tolerant layered approach to fiber-optic networks," *Proc. Conference on High-Speed Networking and Multimedia Computing, IS&T/SPIE Symposium on Electronic Imaging Science & Technology*, pp. 380–391, Feb. 1994.

[AC+94b] H. Abu-Amara, B. Coan, S. Dolev, A. Kanevsky and J. Welch, "A fault-tolerant layered approach to fiber-optic networks," Technical Report 94-050, Department of Computer Science, Texas A&M University, 1994.

[AC+90] B. Awerbuch, I. Cidon, I. Gopal, M. Kaplan and S. Kutten, "Distributed control for PARIS," *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pp. 145–159, Aug. 1990.

[AM91] B. Awerbuch and Y. Mansour, "An efficient topology update protocol for dynamic networks," *Proc. 6th International Workshop on Distributed Algorithms*, pp. 185–202, Nov. 1992.

[CG88] I. Cidon and I. Gopal, "PARIS: An approach to private integrated networks," *Journal of Analog and Digital Cabled Systems* **1**(2), pp. 77–86, Apr.–June 1988.

[CGK88] I. Cidon, I. Gopal, and S. Kutten, "New models and algorithms for future networks," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, Toronto, Canada, pp. 75–89, Aug. 1988.

[Do93] S. Dolev, "Optimal time self stabilization in dynamic systems," *Proc. 7th International Workshop on Distributed Algorithms*, pp. 129–144, Sept. 1993.

[Dij59] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numer. Math.* **1**, pp. 269–271, 1959.

[Dij74] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM* **17**(11), pp. 643–644, Nov. 1974.

[Ev79] S. Even, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

[Ga87] E. Gafni, "Generalized scheme for topology-update in dynamic networks," *Proc. of the 2nd International Workshop on Distributed Algorithms*, pp. 187–196, 1987.

[MRR80] J. M. McQuillan, I. Richer, and E. C. Rosen, "The new routing algorithm for the ARPANET," *IEEE Transactions on Communications* **COM-28**(5), pp. 711–719, May 1980.

[La87] L. Lamport, "A fast mutual exclusion algorithm," *ACM Trans. on Computer Systems,* 5(1):1-11, 1987.

[OY90] Y. Ofek and M. Yung, "Principles for high-speed network control: Loss-less and deadlock-freeness, self-routing and a single buffer per link," *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pp. 161–175, Aug. 1990.

[SG89] J. M. Spinelli and R. G. Gallager, "Event driven topology broadcast without sequence numbers," *IEEE Transactions on Communication* **37**(5), pp. 468–474, May 1989.

[Taj77] W. D. Tajibnapis, "A correctness proof of a topology information maintenance protocol for a distributed computer network," *Communications of the ACM* **20**(7), pp. 477–485, July 1977.