Self-Stabilizing End-to-End Communication*

Baruch Awerbuch† Boaz Patt-Shamir‡ George Varghese§

February 9, 1995

Abstract

Self-stabilizing protocols must begin operating correctly even when started from an arbitrary state. The end-to-end problem is to ensure reliable message delivery across an unreliable network under the weakest possible guarantee from the network – that the sender and receiver are never separated by a cut of permanently failed links. In this paper we present the first self-stabilizing end-to-end protocol. Our solution has message complexities comparable with the best known non-stabilizing solutions. Our solution also has good stabilization time complexity: the time for the protocol to stabilize has the same complexity as the time the protocol takes to deliver a message.

1 Introduction

Informally, a protocol is *self-stabilizing* if when started from an arbitrary global state it exhibits "correct" behavior after finite time. While typical protocols are designed to cope with a specified set of failure modes (e.g., message loss, link failures), a self-stabilizing protocol essentially copes with a set of failures that subsumes most previous categories and is also robust against transient errors (such as memory corruption and malfunctioning devices that send out incorrect messages). There is evidence from real networks that such transient errors do occur [Ros81]

*The ideas in this paper were first described in an extended abstract that appeared in the 32nd IEEE Symposium on Foundations of Computer Science, October 1991.

†Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

‡Supported by ONR contract N00014-85-K-0168, by NSF grants CCR-8915206, and by DARPA contracts N00014-89-J-1988.

§Washington University in St. Louis supported by NSF Research Grant NCR-9405444; part of this work was done while the author was at Lab. for Computer Science, MIT.

and cause systems to fail unpredictably. Thus stabilizing protocols are attractive because they offer *increased robustness* (especially to transient faults) as well as *potential simplicity* (because a stabilizing protocol can avoid the need for a slew of independent mechanisms to deal with a catalog of anticipated faults.)

Self-stabilizing protocols were introduced by Dijkstra [Dij74]. Since then, they have been studied by various researchers; refer to [Sch93] for a recent survey. In this paper, we will describe the first stabilizing solution to a basic networking problem called the *end-to-end communication* problem.

The essence of *end-to-end* communication is delivery, in finite time, of a sequence of data items, generated at a designated sender, to a designated receiver across an unreliable network, without duplication, omission or re-ordering. If network links never fail, the end-to-end task is performed easily by establishing a fixed communication path of FIFO (First-In First-Out) links between sender and receiver. Unfortunately, real networks, are *dynamic* – links may repeatedly fail and recover.

The "classical" approach to handle the problem is to construct a new communication path when an old path fails and to reroute the existing virtual circuit (or transport connection) along the new path. This approach works reasonably for small networks but assumes [Fin79, AAG87] that the *entire network* stabilizes long enough to allow construction of a new path. As networks get larger, this assumption is overly optimistic. For example, if every edge has a constant probability of being operational, then the probability of an entire path being operational is exponentially small in the path length.

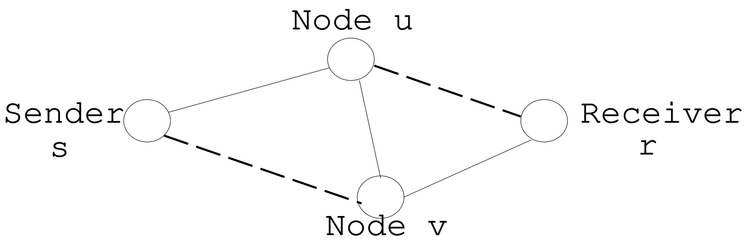

Figure 1: In the network above, dotted lines represent links that crash and recover so frequently that they cannot be used to send a single packet.

In the network shown in Figure 1, assume that the dotted lines represent links that crash and recover so frequently that they cannot be used to send a single packet. Suppose also that the link from s to u is up only in the first 5 seconds of every minute and the link from v to r is up only in the last 5 seconds of every minute; then the network is disconnected at all instants of time! However, one can send packets from sender to the receiver as follows: in the first 5 seconds of every minute, the sender sends packets to u, u then sends packets to v; finally, v buffers the packets till it can send them to the receiver (in the last 5 seconds of every minute).

However, a conventional routing protocol (e.g., the OSI Routing Protocol [Per83]) can do very badly on such a network. If the link from u to r has sufficiently lower "cost" than the link from v to r, then the routing protocol can cause packets to keep looping. When packets reach u, if u knows that its link to r is down, u may route packets to v. However, suppose that when a packet reaches v, v has heard that the link from u to r is up; then v may send the packet back to u and this process can continue indefinitely. Clearly any control system, where the feedback delay is large compared to the rate at which state changes, is subject to oscillation.[1] Are there protocols that can deal with a pathological network like Figure 1?

As we have seen in Figure 1, the existence of an operational communication path is *not* necessary for communication between a sender and a receiver. The necessary condition [AE86] is *eventual connectivity* – i.e., there is no cut of permanently failed edges between the sender and receiver. Observe that in networks with frequently changing topology, feedback about link failures is useless. So why not ignore such notifications?

Most Data Link protocols use retransmissions to deal with link errors; however if the retransmissions exceed some limit, the protocol will decide the link is down, and notify the routing protocol. In networks with slowly changing topology, this notification allows the routing protocol to use other links. However, suppose that we modify the Data Link protocol to keep retransmitting packets indefinitely till an acknowledgement is received. If the link comes up eventually, the packets will be transmitted successfully. *No failure notifications are sent.* This modification (introduced in [AG88]) reduces the extremely dynamic eventually connected model to a simpler and more static *fail-stop model.*

In the fail-stop model, the network topology is fixed, but some links may fail forever, without notification. A link is *non-viable* if, starting from some time, the link does not deliver further messages; otherwise the link is *viable.* In essence, a non-viable link is a link in which the sequence of received messages is a *proper* prefix of the sequence of sent messages. We say that the sender is *viably connected* to the receiver if there exists a simple path of viable links between them. Thus in Figure 1, after reduction to the fail-stop model, we can consider the dotted and solid links to represent non-viable and viable links respectively. The network in Figure 1 is viably connected using the path from s to u, u to v and v to r.

The introduction of the fail-stop model lead to the first polynomial time solution [AMS89]

[1] A simple heuristic fix ([RS]) is to use hysteresis in bringing up links such that links that keep failing and recovering are "suspended" for exponentially increasing amounts of time each time they fail. However, in the network of Figure 1 that will result in suspending the link from s to u and the link from r to v, effectively disconnecting the network forever.

and finally to perhaps the simplest and most elegant solution [AGR92]. For the rest of this paper we will use the fail-stop model. Earlier solutions were not stabilizing. In this paper we address the problem of designing a *self-stabilizing end-to-end protocol.* We also assume self-stabilization of the Data Link layer – i.e., the Data Link begins to operate correctly from some bounded time onwards, for all links, both viable and non-viable.[2] Since the specifications of the End-to-end and Data Link problems are identical, our task can be viewed as extending the guarantees of a self-stabilizing Data Link protocol to the end-to-end layer.

One way to solve the end-to-end problem in a fail-stop network is to flood every message that is to be sent through the network along with a sequence number. The sequence number allows the receiver to reject duplicate copies and prevents messages from looping indefinitely in the network.[3] However, if all the bits in the sequence number are erroneously set to 1 at a node, the sequence number cannot be incremented further, and the protocol will not be self-stabilizing. One way to make this scheme self-stabilizing [Per83] is to bound the lifetime of packets; if the sequence number reaches the maximum value, the protocol essentially stops for a period long enough for all old packets to die.

The disadvantage of the approach in [Per83] is the need to know time bounds and the fact that timers have to be set conservatively to take into account delay variances and maximum network diameters; this results in large stabilization times. Our solution, however, shows that it is possible to build a self-stabilizing end-to-end protocol under the weakest possible assumptions – i.e., the network is *eventually connected* and *completely asynchronous.*

The rest of the paper is organized as follows. First in Section 2, we provide a formal model of a fail-stop network model and a formal specification of what a stabilizing end-to-end protocol must provide. In Section 3 we show how to reduce a fail-stop network to what we call a *C-channel* in a stabilizing fashion. Our solution consists of applying a general method called *local checking* to the original Slide protocol of [AGR92]. Next, in Section 4 we show how to implement reliable message transfer over a C-channel using a stabilizing bounded channel protocol. The formal specification of a C-channel in Section 3 forms a logical firewall between the two parts of the solution in Sections 3 and 4.

2 The model and problem statement

We model the nodes and links of a fail-stop network using a variant of the Input/Output Automaton (IOA) model, [LT89, MMT91]. For example, we can model a FIFO link by an

[2]Self-stabilizing Data Link protocols are well-known [AB89].

[3]A similar approach is used in the OSI Link State Packet propagation algorithm described in [Per83].

automaton (state machine) in which the state of the link is the queue of packets stored on the link. Two transitions that can change the state of the link are actions by which the external world or environment sends a packet to the link (e.g., SEND) and actions by which the link delivers stored packets to the environment (e.g., RECEIVE) .

In the IOA model, transitions by which the environment affects the automaton (e.g., SEND) are called *Input* actions while transitions by which the automaton affects the environment (e.g., RECEIVE) are called *Output* actions. Input actions are under the control of the environment while output actions are under the control of the automaton. Finally, there are *internal actions* which only change the state of the automaton without affecting the environment.

Formally, an I/O Automaton (henceforth IOA) is characterized by its *state set* S, a *action set* A, an action signature G (that classifies the action set into input, output, and internal actions), a *transition relation* $R \subseteq S \times A \times S$, a set of initial states $I \subseteq S$. The set of output and internal actions are called the *locally controlled* actions of the automaton. Fairness is specified by dividing the set of locally controlled actions into a finite number of equivalence classes. For stabilization, we will often limit ourselves to a special type of IOA that we call a *UIOA* (for *uninitialized IOA*) in which the state set is finite and $I = S$. In other words, any state is a possible start state for a UIOA.

An action a is said to be *enabled* in state s if there exists $s' \in S$ such that $(s, a, s') \in R$. By definition, input actions are always enabled. An *execution* of the automaton is modeled by an alternating sequence of states and actions $(s_0, a_1, s_1, \ldots)$, such that $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$, s_0 is a start state, and the execution is fair. An execution E is fair if every locally controlled class C is given a "fair turn"; more formally, if some action of C is enabled in some state s of E, then either some action in E occurs after s or there is some later state in which no action of C is enabled.

There is a notion of composition of automata that allows automata to be "plugged together" using simultaneous performance of shared actions. For example, consider a node automaton that had an action to send a packet SEND(p) as an output action. If this is the same name as the input action for a channel automaton, when the two automata are composed, whenever the node performs a SEND(p) output action, the channel simultaneously performs a SEND(p) input action. Formal details can be found in [LT89]; intuitively the composition of automata is a new automaton whose state set is the cross-product of the component state sets, whose transition relation is obtained from the component automata in the natural way, and whose locally controlled classes are the union of the locally controlled classes of the component automata.

Finally, a behavior is the subsequence of an execution consisting of external (i.e., input and output) actions. Thus each automaton generates a set of behaviors. We specify the correctness of a protocol using a set of behaviors P; an automaton A is said to solve P if the behaviors of A are a subset of P. This definition reflects a belief that the correctness of an automaton should be specified in terms of its externally observable behavior. For example, to specify a FIFO Data Link we might require that the sequence of received packets be identical to the sequence of sent packets.

Modeling the Topology of Fail Stop Networks The topology of a fail-stop network (e.g., Figure 1) is specified using a special graph, that we call an end-to-end graph. An end-to-end graph is a directed graph $G = (V, E)$ with the following additional properties:

1. *Symmetry:* If there is a link from u to v there is a link from v to u. – i.e. $(u, v) \in E$ iff $(v, u) \in E$.

2. *Distinguished nodes*: There are two distinguished nodes $s, r \in V$ called the sender and receiver nodes respectively.

3. *Edge Labels*: Each edge in E is labeled as either viable or non-viable subject to two conditions. First for every $(u, v) \in E$ the label of (u, v) is the same as the label of (v, u). Second, there must be at least one path between s and r in G consisting of only viable links.

In what follows, when we use the word graph we will mean an end-to-end graph. Notice that the third part of the definition models the minimum fairness condition needed to solve the end-to-end problem. The consequences of an edge being labeled viable or non-viable were intuitively described earlier and will be formally described below. For a graph $G = (V, E)$, we use $n = |V|$ and $m = |E|$ to denote the number of nodes and links respectively. An actual end-to-end system for a graph G will consist of node automata N_u for every node in G and a link automaton for every edge.

Modeling Links in a Fail-Stop Network Traditional models of a FIFO Data Link allow a link to store an unbounded number of packets. For the stabilizing end-to-end problem, however, we can initialize non-viable links with an unbounded number of bad packets and deliver these bad packets an unbounded amount of time later. Thus a stabilizing solution to the end-to-end problem would be impossible under this "unbounded capacity" assumption. Since real links are bounded and bounded links can be modeled elegantly, we restrict ourselves to bounded link models.

For each unidirectional link in G, we will assume there is a special type of channel automaton called a unit capacity Data Link (UDL). Intuitively, a UDL can store at most one packet at any instant. Node automata communicate by sending packets to the UDLs that connect

Each p belongs to the packet alphabet $\mathcal{P}$ defined above.
The state of the automaton consists of a single variable $Q_{u,v} \in \mathcal{P} \cup \{nil\}$.

$\text{SEND}_{u,v}(p)$ (*input action by which u sends a packet to link*)
Effect:
If $Q_{u,v} = nil$ then $Q_{u,v} := p$; (* packet stored if link was empty *)

$\text{FREE}_{u,v}$ (* output action that tells node u that link is free *)
Precondition: $Q_{u,v} = nil$ (* enabled whenever link is empty *)
Effect: None

$\text{RECEIVE}_{u,v}(p)$ (* output action by which link delivers a packet to node v *)
Precondition: $p = Q_{u,v} \neq nil$
Effect: $Q_{u,v} := nil$; (* remove delivered packet from link *)

If (u, v) is a viable link the FREE and RECEIVE actions are in separate classes. If (u, v) is non-viable there are no locally controlled classes.

Figure 2: Unit Capacity Data Link automaton

them. We show [Var93] that a UDL can be implemented over real physical channels and can be generalized to bounded capacity links.

Besides the usual actions to send and receive packets a UDL also has an output action FREE to tell the sender that the link is ready to accept a new packet. This allows the sender to cope with the fact that the link has bounded capacity without causing packets to be dropped. Formally, we say that $C_{u,v}$ is the UDL corresponding to link (u, v) in an end-to-end graph G if $C_{u,v}$ is the IOA defined in Figure 2.

We parameterize the a UDL by the sender and receiver nodes. For the UDL $C_{u,v}$, the sender is u and the receiver is v. The external interface to $C_{u,v}$ includes an action to send a packet at node u ($\text{SEND}_{u,v}(p)$), an action to deliver a packet at node v ($\text{RECEIVE}_{u,v}(p)$), and an action $\text{FREE}_{u,v}(p)$ that tells node u that the link is free. The state of $C_{u,v}$ is simply a single variable $Q_{u,v}$ that stores a packet or has the default value of *nil*.

Notice three points about the specification of a UDL. A UDL is a UIOA – i.e., we have not defined any start states for the UDL. Second, if the UDL has a packet stored, then any new packet sent will be dropped. The FREE action is enabled whenever the UDL does not contain a packet. Notice also that if the link is viable, then any stored packet will eventually be delivered; also if the link is empty, eventually, a FREE action is delivered. However, with a non-viable link there are no such guarantees.

Modeling a Fail-Stop Network We use a second unit of data transfer called message. Informally, messages are the units of data that are input to the fail-stop network at the sender node and delivered from the network at the receiver node; packets are the unit of data transfer that the protocol uses *within* the network. We use the letter m to denote messages and the letter p to denote packets. Formally, messages are drawn from a fixed message alphabet $\mathcal{M}$.

We model a fail-stop network for end-to-end graph G as the composition of arbitrary node automata and UDLs for every edge. Wee also require that the sender and receiver nodes have special interfaces to send and receive messages. Formally, an *end-to-end automaton* for end-to-end graph G is the composition of arbitrary node automata $\{N_u, u \in V\}$ and UDL's $\{C_{u,v}, (u,v) \in E\}$. The sender node s must have an input action SEND_MESSAGE$(m), m \in \mathcal{M}$ and an output action FREE_MESSAGE; also the receiver node r must have output action RECEIVE_MESSAGE$(m), m \in \mathcal{M}$.

Correctness and Performance Metrics A behavior of an end-to-end automaton is the subsequence of executions containing SEND_MESSAGE, FREE_MESSAGE and RECEIVE_MESSAGE actions. We would like an end-to-end automaton to deliver messages sent at the sender reliably and in FIFO order to the receiver. We state this correctness condition formally using what we call a unit capacity message link (UML). Formally, a UML is identical to a *viable* UDL except with messages replacing packets and the SEND$_{u,v}$, FREE$_{u,v}$ and RECEIVE$_{u,v}$ actions replaced by SEND_MESSAGE, FREE_MESSAGE and RECEIVE_MESSAGE actions respectively.

For stabilization, we allow the channels initially to have stored packets and the nodes to be uninitialized automata. However, we will only require that the network "eventually" begins to behave like a unit capacity message link.

Formally, let G be any end-to-end graph. We say that N is a stabilizing end-to-end protocol for graph G if:

- N is an end-to-end automaton for graph G in which all node and channel automata are UIOA (i.e., every state is a possible start state).
- Every behavior of N has some suffix that is a behavior of a UML.

Our definition says nothing about how long a solution N takes to stabilize. Thus we use time complexity measures to evaluate solutions to the end-to-end problem. For an end-to-end automaton, to evaluate time complexity[4] we assume that no packet can be stored on a viable link for more than 1 time unit, and that a viable link cannot be empty for more than 1 time unit without delivering a free signal. With this assumption, we can define the following metrics for a stabilizing end-to-end protocol N for graph G. Intuitively, *Stabilization time* is the worst case time it takes N to stabilize; *Message delivery time* is the worst case time to deliver a message. Formal definitions can be found in [APV95]. *Space* is the the maximal amount of space required by a node program.

3 Reducing a Fail-Stop Network to a C-Channel using SLIDE

The (non-stabilizing) end-to-end protocol described in [AGR92] has two parts: first part is the SLIDE protocol that is used to reduce a fail-stop network to a non-FIFO bounded capacity link that we call a C-Channel. In this section, we review their solution and briefly describe the modifications required to create a stabilizing SLIDE. The second part of [AGR92] shows how to implement reliable message delivery over a C-channel. Our protocol to implement the second part is quite different from the original solution. We describe the second part of our solution in Section 4. We start by formally defining a C-channel.

A C-Channel is a Data Link that can store up to C packets but can deliver stored packets in any order. The state of a C-channel state consists of two components: an input queue IQ that can store a single packet and a multiset M that can store up to $C-1$ packets. The state machine has three main transitions that can change its state: an action SEND_PACKET by which a sender sends a packet, an action FREE_PACKET by which the channel announces that its input queue is free, and an action RECEIVE_PACKET by which the channel delivers a packet to a receiver. There is also an internal action called FILL_CHANNEL.

The formal specification for the C-channel is given in Figure 3. The state machine works as follows. Whenever the input queue is empty, the channel can perform a FREE_PACKET event. A SEND_PACKET(p) event can occur at any time but if occurs when the input queue already stores a packet, the packet is dropped (as in a UDL); otherwise, the packet p is stored in the Input queue. A FILL_CHANNEL event can occur whenever the input queue stores a packet p and the channel multiset has strictly less than $C-1$ packets; it results in emptying the input queue and placing packet p in the multiset. Finally, a RECEIVE_PACKET(p) event is enabled whenever $p \in M$, after which p is removed from the multiset.

Notice that all locally controlled actions are in a single class. Thus the "liveness" guarantees

[4] Note this assumption is only used to evaluate time complexity; the solutions must work correctly regardless of timing assumptions.

Let P be the data packet alphabet. IQ belongs to the domain $P \cup \{nil\}$
M is a multiset of size at most C containing packets that belong to
domain P. We use p to denote an element of P.

SEND_PACKET(p) (* input action to send packet to channel *)
  Action:
    If $IQ = nil$ then $IQ := p$; (* store packet if input queue is empty *)

FREE_PACKET (* output action to tell sender that channel is not full *)
  Precondition: $IQ = nil$ (* enabled whenever input queue is empty *)

FILL_CHANNEL(p) (* internal action to remove packet from input queue and place in multiset *)
  Precondition: $IQ = p$, $|M| < C - 1$
  Action: $IQ := nil$; $M := M \cup \{p\}$;

RECEIVE_PACKET(p) (* output action to deliver packet stored in multiset *)
  Precondition: $p \in M$
  Action: $M := M - p$; (* remove delivered packet *)

The FREE_PACKET, FILL_CHANNEL and RECEIVE_PACKET actions are in a single class.

Figure 3: Formal Specification of a C-channel

of the C-channel are very weak. For example, it guarantees that if the C-Channel is completely empty then a FREE_PACKET signal will eventually be delivered; similarly, if the C-channel is full, then some packet will be eventually delivered. This specification does not seem much stronger than the original fail-stop network; nevertheless, protocols can be devised that use the weak guarantees of a C-channel to create reliable Data Links.

Why SLIDE Implements a C-channel We now review the original SLIDE protocol and describe why it implements a C-channel when properly initialized. Figure 4 is an expanded view of Figure 1, together with the key protocol variables used to implement the SLIDE protocol. Notice that Figure 4 has the same network interface as a C-channel. Each node u has a "pile" of bounded size for each incoming link L where u stores data packets arriving on link L. If the sender keeps sending packets to the network, then packets "slide" along a viable path to the receiver.

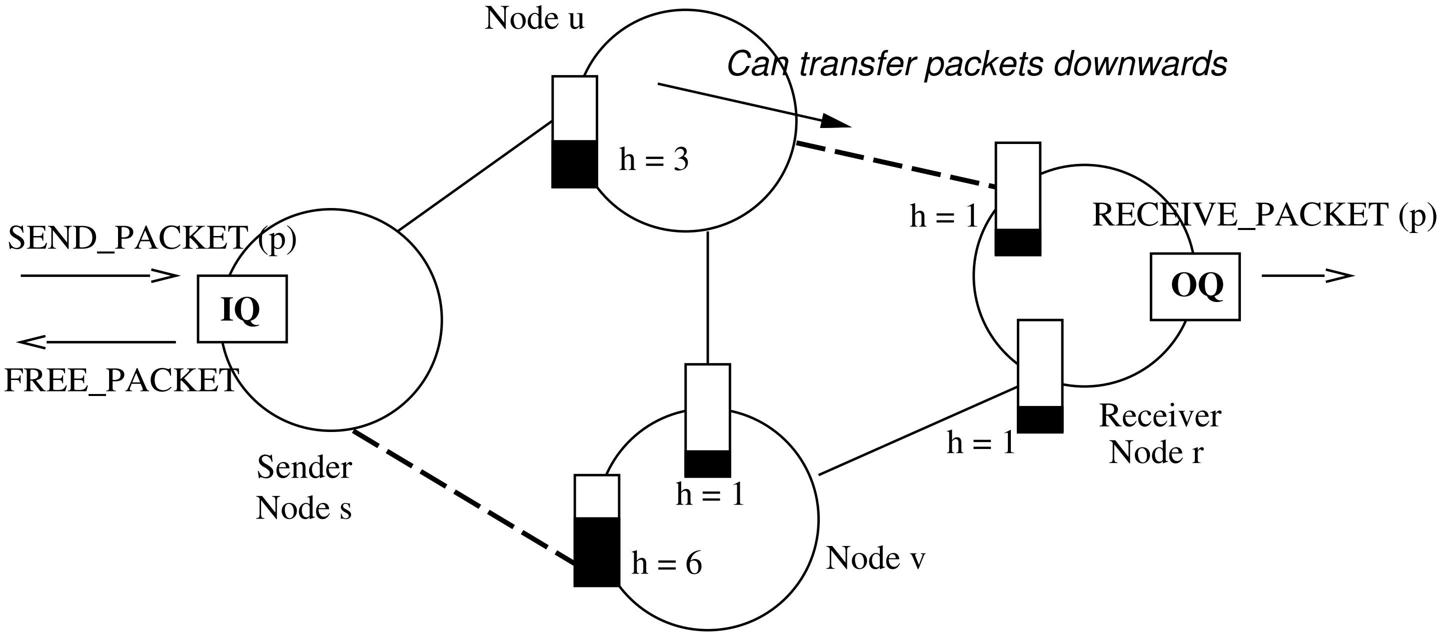

Figure 4: The dotted lines represent non-viable links while solid lines represent viable. Without knowing in advance which links are viable, SLIDE pushes packets from the sender to receiver along a path of viable links by pushing packets from piles of higher height to piles of lower height.

Recall that n is the number of nodes in graph G. All nodes have a pile of n slots *for each incoming link*; a pile is an array of packets. Each slot or array element has a unique array index that we call its *height* which ranges from 1 to n. Packets in a pile are placed consecutively starting from the lowest numbered slot. The *height of a pile* is the height of the highest occupied slot in the pile. In Figure 4 we use dark ink to denote the portion of a pile that contains packets.

The sender and receiver have one additional pile each. The sender's (Figure 4) has an input queue which stores one packet. New data packets that arrive to the network in a SEND_PACKET action are placed in the input queue if the queue does not already contain a packet. Similarly the receiver has an output queue which stores one packet. Any packet that is in one of the other piles at the receiver can be removed and placed in the output queue for a later RECEIVE_PACKET action.

The sender's input queue, by definition, is considered to be a single slot of height n and the receiver's output queue is considered to be at height 1. The basic idea is that data packets only travel from higher numbered slots to lower numbered slots. Each node u tries to remove a packet from slot h on one of its incident piles and send it to a lower numbered (than h) slot in some neighbor v. If the sender's input queue is empty, the protocol will deliver a FREE_PACKET event which indicates willingness to receive new packets.

For node u to send a packet over link (u, v), u needs an upper bound on the highest occupied slot at the (u, v) pile at node v. This bound is maintained by a simple incremental bound estimation protocol between u and v (Figure 5). Assume that u initializes (recall we are discussing the non-stabilizing protocol) the bound to 1 initially and the pile at v is initially empty. The bound at u is incremented whenever u sends a data packet to v; v sends a special *Decrement* packet whenever v removes a packet from its incident pile; finally, u decrements its bound when it gets a *Decrement* packet from v.

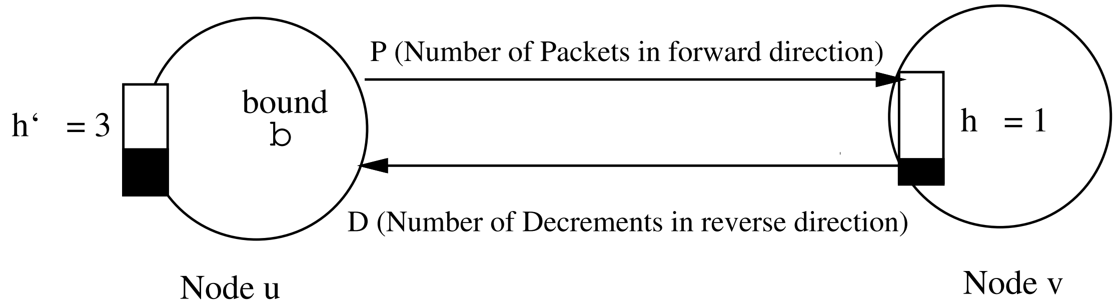

Figure 5: Key variables for the bound estimation protocol on a link. The protocol maintains the invariant $b = h + P + D + 1$.

Let P represent the number in packets in transit (Figure 5) from u to v, and D represent the number of *Decrement* packets in transit from v to u. It is not hard to see that the bound estimation protocol maintains the following predicate that we will call $L_{u,v}$. This predicate states that $b = P + D + h + 1$, where b is the bound at u for neighbor v, and h is the height of the incident pile for node u at node v. This simple predicate ensures correct operation because it ensures that b is indeed an upper bound. It also ensures that for any viable link (u,v), if u stops sending packets to v, then eventually all transit packets will be delivered, resulting in a state where the bound is exact – i.e., $b = h + 1$.

This predicate guarantees that a properly initialized SLIDE protocol behaves like a C-Channel with $C = O(nm)$. Observe (from $L_{u,v}$) there are at most $O(n)$ packets in transit on every edge, and at most n packets in every pile. Thus there can be at most $O(nm)$ packets stored in the network. Also since the bound is an upper bound, no packets are ever dropped. Thus we can map the input queue in the SLIDE to the input queue in the C-channel and the packets stored in the SLIDE network[5] to the packets stored in the C-channel multiset. We map the FILL_CHANNEL event to any action at the sender node by which a packet is moved from the input queue to some adjacent link.

However, the C-channel also guarantees a liveness condition: at least one of the FREE_PACKET, FILL_CHANNEL, or RECEIVE_PACKET events will eventually occur. So suppose this is not true starting from some state s. Then it must be true that the input queue remains full and the output queue remains empty after state s. Consider any path of viable links between the sender and receiver. Packet sending must eventually stop after s because packets only move downwards and no packets enter or leave; thus the bounds on all links in the viable path eventually become exact. But if the bounds are exact we can build a chain of inequalities (see Figure 6) linking the heights of the output and input queues which shows that at the sender node the bound for the outbound link (to the second node in the viable path) is strictly less than N. But this implies that the sender could remove a packet from its input queue (which is at height N), a contradiction. Thus the liveness condition must be true.

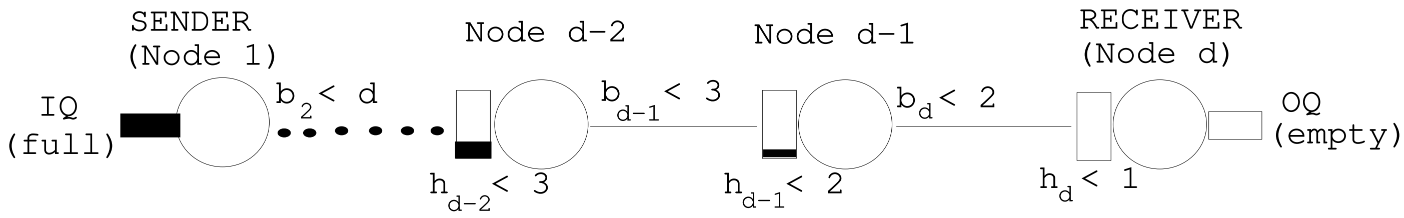

Figure 6: To prove that the SLIDE protocol does not deadlock, we consider a viable path $v_1, v_2, \ldots, v_d$. Using the facts that the bounds are exact in a deadlocked state, we build a chain of inequalities linking the heights of the output and input queues.

[5]including the output queue

Making SLIDE **Stabilizing** We saw that SLIDE behaves "correctly" as long as the predicate $L_{u,v}$ holds for every edge (u, v) in the graph. Clearly, a stabilizing protocol cannot rely on initialization. Thus we have each node u run a snapshot protocol [CL85] to check the state of the bound estimation protocol to node v and check if predicate $L_{u,v}$ holds; if not, node u initiates a local reset which basically results in u initializing its bound to 1 and v emptying its incident pile corresponding to node u. Thus we essentially replace initialization with *periodic local checking and correction* of of the bound estimation protocols on each link. Of course, the snapshot and reset protocols must themselves work correctly without initialization [Var93].

However, non-viable links may stop delivering any packets. Hence any snapshot or reset procedures initiated on such links may never terminate and so $L_{u,v}$ may never hold on such links. Our solution to this problem is to always perform one invocation of the snapshot/reset protocol between the sending of any two data packets on a link. Our snapshot and reset protocols on a single link can be made stabilizing. Thus we can show ([Var93]) that within a constant number of invocations of the snapshot/reset protocol on link (u, v), $L_{u,v}$ will become and stay true. Thus, for any link we guarantee that *we send at most a constant number of "bad" data packets when the link predicate does not hold.* Essentially, this allows us to bound the damage done when the link predicate does not hold.

"Bad" packets cause two kinds of problems; a packet may be sent to a pile of higher height (if all packets can be sent to piles of arbitrary height, the SLIDE protocol can livelock). Second, it can lead to packets being dropped in three ways: i) on the link because the sender sends a packet when the link has a packet ii) at the receiver because a packet arrives to find a full pile at the receiver iii) at the receiver when the receiver does a local reset and empties its incident pile.

However, alternating checking/correction phases with the sending of data packets guarantees that in any execution: **i)** At most $O(n)$ packets per link can be dropped, leading to an upper bound of $O(nm)$ dropped packets per execution. **ii)** At most a constant number of packets per link are sent upwards. With this "bounded" amount of damage, the modified SLIDE protocol continues to behave like a C-channel except that we increase the capacity of the C-channel to account for the $O(nm)$ dropped packets; also the liveness arguments have to be modified slightly to account for $O(m)$ aberrant packets that may move "upwards".

A single phase of either a snapshot or reset procedure consists of u sending a request that is received by v, followed by v sending a response that is received by u. The snapshot and reset procedures are summarized in Fig 7, where as before b denotes the bound at u, h denotes the height at v and D denotes the number of queued *Decrement* packets at node v when a request reaches v, and P denotes the number of queued data packets at u when the response reaches u. At the end of a snapshot phase if $b \neq h + P + D + 1$, then u concludes that local property

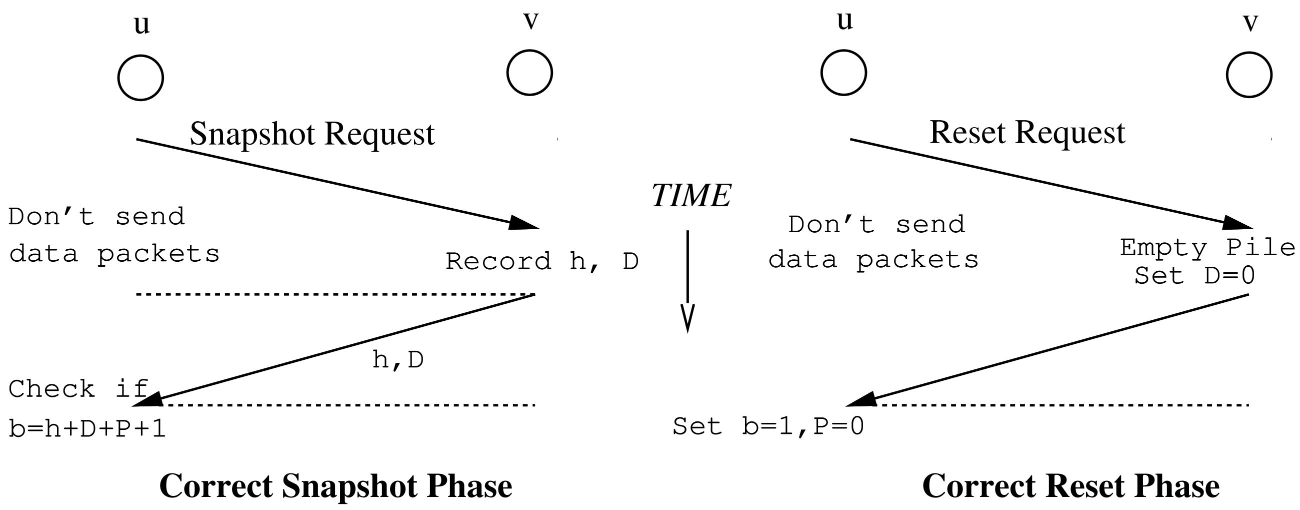

Figure 7: Local Snapshots and Resets.

$L_{u,v}$ does not hold, and initiates a reset. The reset effectively resets all variables in the bound estimation protocol.

The snapshot protocol we described may fail if there are spurious response packets in the initial state of the protocol. To make the snapshot and reset protocols stabilizing, we consecutively number all request and response packets and only accept responses at u if they match the number of the last request sent. Using this we can show [Var93] that within 4 invocations of the snapshot protocol that the snapshot/reset protocol will begin correctly even after starting in an arbitrary state.

The formal code for our stabilizing SLIDE is given in [APV95]. We do not provide the details here for lack of space and because our changes to the SLIDE protocol are quite simple. Instead we concentrate next on providing details of the second part of our solution.

4 Stabilizing Bounded Channel Protocol

We turn to the second part of our solution, that of implementing a stabilizing and reliable message delivery protocol over a C-channel. We describe our solution in terms of a sender process SP and a receiver process RP that communicate over a C-channel (Figure 8). We first describe our solution and then compare it with the (non-stabilizing) solution given in [AGR92].

The specification of the C-channel (Figure 3) provides two useful properties, *unlimited*

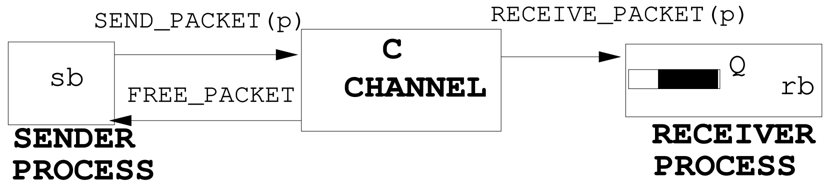

Figure 8: The Bounded Channel Protocol (BCP) consists of a Sender Process and a Receiver Process that communicate over a C-channel. The key variables are a sender bit sb, a receiver bit rb, and a packet queue (Q) at the receiver.

acceptance and *set delivery*:

Unlimited Acceptance: All packets that the Sender Process wishes to send will eventually be accepted by the network[6] if the Sender Process follows the sending discipline where it waits for a FREE_PACKET signal before sending another packet. This follows from the liveness condition and from the fact that SEND_PACKET and FILL_CHANNEL events cannot keep occurring without a FREE_PACKET occurring.

Set Delivery: If in some interval the sender sends a set of S packets that are accepted by the C channel and $|S| > C$, then in the same interval the receiver will receive a set S' that is derived from S by at most C omissions and at most C additions. This follows because the C-channel can store at most C packets and can deliver at most C previously stored packets.

If the sender and receiver were perfectly synchronized, the following protocol is sufficient to deliver a single message m. To send m, the sender sends $2C+1$ copies of a packet p containing m using the SEND_PACKET(p) interface to a C-channel. Once this is done, we know from the set delivery property that at least $C+1$ packets containing m are delivered and at most C "bad packets (containing arbitrary information) are delivered. Thus the receiver can extract m from the packet that is received a majority of times. Abstractly, this amounts to encoding a message m as a set S of $2C+1$ packets and decoding the received set using a simple majority rule.

This simple scheme breaks down when we send more than one message. To allow for sending a sequence of messages, especially when the sender and receiver are not initially synchronized:

- We modify the abstract encoding and decoding functions to allow for $2C$ omissions and C deletions. The extra "slop" of C omissions is important for our protocol.
- We add a bit to every packet such that the bit toggles on the packets in alternate messages. This is used for stabilization.

[6]A packet is said to be accepted if it is stored in the input queue of the network

So far we have not committed to the format of a packet. We let a packet be a message with a bit and an integer field appended to it. - i.e., $p = (m, b, i)$, where b is a bit and i is a positive integer whose size will depend on the encoding used. We use the bit to implement an alternating bit protocol that will help the sender and receiver recover synchronization even after starting in an arbitrary state. The integer i is a *position index field* that is used for one particular encoding of messages described in Section 4.3.

We now abstract the notion of an Encode-Decode function. This allows us to define a generic message delivery protocol using the Encode-Decode function, and then customize the protocol by plugging in different Encode-Decode functions. This motivates:

Definition 4.1 *An encode-decode specifier (E, D, l) is a pair of functions E, D such that:*

E takes as argument a message drawn from $\mathcal{M}$ and a bit and returns a multiset of packets of size at most l; D takes as argument a multiset of length at most l and returns a message drawn from M.

If $E(m, b) = S$ then all packets p in S will have bit b (i.e., the encoding must preserve the bit passed as an argument.)

For any m if $E(m, b) = S$, and S' is any set that contains at most C packets not in S and all but $2C$ packets in S, then $D(S') = m$. We assume that $l > 2C$.

4.1 Protocol Code

We use a generic encode-decode specifier (ENCODE, DECODE, X) to build a self-stabilizing message delivery protocol over a C-channel. Recall that $X > 2C$ (the reason for this assumption will become apparent below). Figure 8 shows the main data structures used by our protocol. In order to allow the sender and receiver to synchronize even after starting from an arbitrary state, the sender and receiver run a protocol that is a generalization of the alternating bit protocol. All encodings of a message carry the sender's current bit sb. Similarly the receiver only uses packets that match the current receiver bit rb in order to decode packets into messages.

In Figure 8, the receiver also uses a queue Q (of size X) which stores the last X packets received. Let *R_Set* be the subset of packets in queue Q with bit equal to receiver bit rb. When *R_Set* becomes "large enough" (see Figure 10), the receiver decodes *R_Set* to a message, outputs this message, and then flips bit rb. The sender protocol is even simpler: the sender simply encodes each message into a set of packets and sends them to the network, flipping its bit sb between messages.

Variable Range Restrictions
  sb is a bit, and *freepacket* is a boolean
  S_Set is a multiset containing at most X packets, $X > 2C$.

SEND_MESSAGE(m) (* input action to send a message *)
  If S_Set is empty (*ignore input if busy sending a previous message*)
    $sb := \overline{sb}$; (*flip sender bit for new message*)
    $S\_Set := \text{ENCODE}(m, sb)$; (*fill set with encoding of message*)

SEND_PACKET(p) (*output action to send packet to network*)
  Preconditions:
    $p \in S\_Set$; (* packet in sender's set?*)
    $freepacket = true$; (* network is ready to accept a packet *)
  Effect:
    $S\_Set := S\_Set - \{p\}$ (* remove packet from sender set*)
    $freepacket := false$; (* assume network is busy until notified*)

FREE_MESSAGE (*output action that denotes readiness to accept a new message*)
  Precondition: S_Set is empty;

FREE_PACKET (*input action that says network can accept a new packet*)
  $freepacket := true$;

Figure 9: Sender Message Delivery Protocol (SP)

The sender also uses a flag *freepacket* (that tells it when it can send another packet to the C-channel). The sender also uses a multiset S_Set of size X which is used to store the encoded version of the current message being sent; more precisely, S_Set stores the remainder of the current message encoding that has not yet been sent to the network.

The sender and receiver protocols (SP and RP) are described in Figure 9 and Figure 10. We assume that all locally controlled actions are in different classes; thus every continuously enabled action will eventually occur. We also assume that the operation ADDQ(p, Q) shifts the queue Q forward and adds packet p to the end of Q. If Q had X packets before p was added, the packet at the head of Q is removed to make room for p. It is easy to implement such a queue in a self-stabilizing fashion using an array of X packets. Let BCP (for Bounded Channel Protocol) denote the composition of the receiver and sender processes with a C-channel.

Variable Range Restrictions
  rb is a bit
  Q is a queue containing exactly X packets, $X > 2C$
  R_Set is a multiset containing at most X packets

RECEIVE_PACKET(p) (*input action, executed on receipt of a packet*)
  ADDQ(p, Q) (* add p at the end of receiver queue*)
  $R\_Set := \{p \in Q : p.bit = rb\}$; (* extract packets with bit equal to receiver bit *)
  If $|R\_Set| \geq X - C$ then (* enough packets with correct bit? *)
    Let m = DECODE(R_Set) (* make message out of them *)
    RECEIVE_MESSAGE(m); (* output message m to external user *)
    $rb := \overline{rb}$; (* flip receiver bit *)

Figure 10: Receiver Message Delivery Protocol (RP)

4.2 Correctness Arguments for the Message Delivery Protocol

First we notice (from the code of the C-channel) that any packet p sent by by SP will result in p being stored in the channel if the following predicate holds in the prior state. We say the channel is drop-free in a state s of BCP if whenever ($freepacket = true$) then ($IQ = nil$). (Recall that IQ is the input queue of the network.) It is easy to see that the channel being drop-free is a stable property that holds after the first SEND_PACKET event in any execution. It is also easy to see that in any well-formed execution of BCP, the channel is drop-free in all states that follow the second SEND_MESSAGE event because in between two such events there must have been a SEND_PACKET event.

We can divide any execution into non-overlapping message intervals using the following definition. In any execution the i-th message interval is the interval $[B(i), E(i)]$ where $B(i)$ is the state immediately after the i-th SEND_MESSAGE event and $E(i)$ is the state just before the $i+1$-st SEND_MESSAGE event. $B(i)$ and $E(i)$ can be read as the beginning and end of the i-th interval. Let $Bit(i)$ be the value of sb in state $B(i)$ (i.e., the value of the sender bit at the start of the i-th interval). Notice that $Bit(i)$ is also the value of the sender bit for any state in the i-th interval, since the sender bit only toggles at the start of an interval. Define the set of received packets in an interval I to be the multiset containing an element p for every RECEIVE_PACKET(p) event that occurs in I. Clearly if the channel is drop-free in some interval all packets that are sent in the interval must either be received or stored in the channel.

We will use $A + B$ to denote the union of multisets A and B that creates a new multiset which includes *all* elements in both A and B. Let CHANNEL denote the multiset containing the multiset M of the C-channel together with any packet in the input queue of the C-channel. More precisely, CHANNEL $= M \cup \{IQ\}$. The following notation is convenient for our proofs. For any bit b we denote by Q_b, CHANNEL$_b$ and S_Set_b the multiset of packets in Q, CHANNEL and S_Set respectively with bit equal to b.

We will see that the following notion of synchronization is the key to ensuring that messages are delivered.

Definition 4.2 *We say that the sender and receiver are* synchronized *if whenever S_Set is empty:*

- $sb = \overline{rb}$ *(i.e., the receiver is expecting the opposite bit) and*
- $|Q_{\overline{sb}} + \text{CHANNEL}_{\overline{sb}}| \leq C$. *(i.e., there is only a small number of potentially confusing old packets in the network and the receiver with bit equal to the bit of the next message that will be sent.)*

Let m_i denote the message that is sent at the start of the i-th interval. Informally, it is easy to see that if the sender and receiver are synchronized at the end of Interval $i-1$, then during Interval i message m_i (and only message m_i) is delivered to the receiver message queue. This is because when we go from the end of Interval $i-1$ to the start of Interval i, the value of sb toggles (see code). Thus, at this point, the sender and receiver bits are equal and the number of old packets with bit equal to the sender bit are small. Given this it is easy to show that the receiver must output exactly one message during this interval. This is because the sender sends X packets during this interval with bit equal to sb. Of these at least $X - C$ must get to the receiver, which will force the receiver to output a single message.

Second, we know that at the beginning of the interval there were at most C old packets with bit equal to sb. Thus of the $X - C$ packets that the receiver uses to output a message, there can be at most C packets with bit equal to sb that are not part of the encoding of message m_i. The remaining ($\geq X - 2C$) packets must belong to to encoded multiset corresponding to m_i. Thus applying the decode function to these packets will correctly recover m_i.

We will prove this below. However, we first prove a self-stabilization property: regardless of the initial state of BCP, the sender and receiver are synchronized after the second message is sent.

Lemma 4.1 *After the second* SEND_MESSAGE *event in any execution, if S_Set is empty then* $|Q_{\overline{sb}} + \text{CHANNEL}_{\overline{sb}}| \leq C$.

Proof: Consider some state s that occurs after the second SEND_MESSAGE event in any execution of BCP, and such that $s.S\_Set$ is empty. Now s is part of some message interval, say the i-th message interval. By assumption, $i \geq 2$. In this interval, the channel is drop-free. Recall that $B(i)$ denotes the state that begins the i-th messsage interval and $Bit(i)$ denotes the sender bit in $B(i)$. Clearly, $|B(i).S\_Set| = X$ and $|s.S\_Set| = 0$. Notice also that the value of sb remains unchanged in an interval; hence all packets in set $B(i).S\_Set$ have bit equal to $Bit(i) = s.sb$.

Let R denote the set of received packets in the interval $[B(i), s]$. We know that, all packets sent in this interval must either be received or be stored in the channel. Now we consider two cases:

Case 1: All packets in set R are in Q in state s. Recall that packets can be lost from Q when new packets are added; this case essentially says that any received packets in this interval do not overflow. All X packets sent during this interval are either in the channel or in the receiver queue at the end of the interval. But since all X packets that were sent have bit $Bit(i)$, and the total capacity of channel plus receiver queue is at most $X + C$, the lemma follows.

Case 2: There is at least one packet in R that is not in Q in state s. (i.e., at least one received packet has overflowed the receiver queue during this interval.) Let $\overline{Bit(i)} = b$. Since the receiver packet queue is FIFO, any packets with bit b that were in the queue at the start of the i-th interval must have been removed from the queue by the time we reach state s. Thus the only packets that could be in the queue plus channel in state s must consist of packets sent during the i-th interval as well as packets that were in the channel at the start of the i-th interval. But the former set consists entirely of packets with bit $\overline{b} = Bit(i)$ and the size of the latter set is at most the channel capacity C. Thus the number of packets in the queue plus channel with bit equal to b in state s is at most C. The lemma follows. □

Lemma 4.2 *For any execution of* BCP, $|Q_{rb}| < X - C$ *in all states following the second* SEND_MESSAGE *event.*

Proof: It is easy to see from the code that $|Q_{rb}| < X - C$ is true after the first RECEIVE_PACKET event. This is because whenever a packet is received that causes $|Q_{rb}| \geq X - C$, the receiver toggles the value of rb which then makes the condition untrue. (Recall that we have assumed that $X > 2C$.) It is also not hard to see that between the first and second SEND_MESSAGE events at least one RECEIVE_PACKET event must have occured. □

Lemma 4.3 *After the second* SEND_MESSAGE *event in any execution of* BCP, *the sender and receiver are synchronized.*

Proof: Consider any state s after the second SEND_MESSAGE event in the execution. If $s.S\_Set$ is empty, we know from Lemma 4.1 that in s, $|Q_{\overline{sb}} + \text{CHANNEL}_{\overline{sb}}| \leq C$. But $|Q + \text{CHANNEL}| \geq X + C$. Hence $|Q_{sb} + \text{CHANNEL}_{sb}| \geq X - C$.

Suppose, for contradiction, that in s, $sb = rb$. Then it follows that $|Q_{rb} + \text{CHANNEL}_{rb}| \geq X - C$. But this contradicts Lemma 4.2, which states that in s, $|Q| < X - C$. Hence $s.sb = s.rb$. □

We now prove carefully what we intuitively expect, that after the sender and receiver are synchronized, any behaviour of the protocol is a behaviour of a unit capacity message link. First we need some invariants and we also need to define two new history variables that are used for a mapping proof.

For any state s in any execution we define the history variables $s.Previous\_Message$ and $s.Previous\_Set$ as follows. If there is no SEND_MESSAGE event before s, then both history variable are undefined. undefined. If not, let s' be the first state before s that is immediately preceded by a SEND_MESSAGE event, say SEND_MESSAGE(m). Then $s.Previous\_Message = m$ and $s.Previous\_Set = s'.\text{CHANNEL}_b + s'.Q_b$ where $b = s'.sb$. In other words *Previous_Message* records the last message that was input to BCP; also *Previous_Set* records the set of packets (with bit equal to the sender bit) in the channel and receiver queues at the instant after the last message arrives.

Lemma 4.4 *Consider any well-formed execution that starts with a state in which S_Set is empty, and such that the sender and receiver are synchronized in all states. Then the following predicates hold in all states of* α.

1. *If* $(sb = \overline{rb})$ *then* $|Q_{rb} + \text{CHANNEL}_{rb}| \leq 2C$.
2. *If* $(sb = rb)$ *then* $S\_Set_{sb} + \text{CHANNEL}_{sb} + Q_{sb} \subseteq \text{ENCODE}(Previous\_Message) + Previous\_Set$ *and* $|Previous\_Set| \leq C$.

Proof: Simple inductive proof. See [APV95] for details. □

Armed with the invariants we can easily prove that any synchronized behavior of the protocol is a behavior of a UML, as we desire.

Lemma 4.5 *Consider any well-formed execution that starts with a state in which S_Set is empty, and such that the sender and receiver are synchronized in all states. Then the behavior corresponding to this execution is a behaviour of UML.*

Proof: To do so we exhibit a mapping function f from the states of the protocol to the states of UML. We define $f(s)$ formally as follows:

- If $s.sb = s.rb$ then $f(s).Q = s.Previous\_Message$ (i.e., if the sender and receiver bits match, the message stored in UML's queue is the last message sent.)
- If $s.sb = \overline{s.rb}$ then $f(s).Q = nil$ (i.e., if the sender and receiver bits do not match there is no message stored in UML's queue.)

First we see that BCP and UML have the same external action signature. and for any s, $f(s)$ is a valid state of UML. We now show that for all transitions (s, π, s') of BCP, the UML transition $(f(s), \pi, f(s'))$ has the same external behavior. This can be used to show, by induction, that any behavior of BCP is a behavior of a UML [LT89]. The details can be found in [APV95].

We also need to verify that the liveness properties are preserved by this mapping. It is clear that at the start of any message interval of BCP, *S_Set* is non-empty; the mapping then implies that there is a message in the corresponding queue of UML. We also know from the Unlimited Acceptance property of a C-channel that eventually within this interval *S_Set* will become empty; the invariants (Lemma 4.4) then imply that the the sender and receiver bits are unequal, which in turn implies that the message is removed from the UML queue. This remains true till the end of the interval, after which the argument can be repeated. □

Theorem 4.6 *Consider any behavior b of the message delivery subsystem formed by the composition of the sender process (Figure 9, the receiver process (Figure 10), and the C-channel (Figure 3). Any suffix of b starting after the third* SEND_MESSAGE *event in b is a behavior of a UML.*

Proof: Follows from Lemma 4.5 and Lemma 4.3. □

We would like to conclude from Theorem 4.6 that:

Theorem 4.7 *The message delivery subsystem stabilizes to the behaviors of a UML.*

Unfortunately, this theorem is not true for the protocol we described because it is possible that the user never sends three messages, in which case we cannot claim that the message delivery protocol has stabilized. But this is just a technicality that can be solved as follows. We modify the sender protocol SP to keep two outstanding messages, the current message being sent in *S_Set* as well as an additional buffer to store the next message to be sent. If after finishing sending the current message (i.e., *S_Set* is empty), SP finds that the additional buffer is empty, then SP inserts a encoding of a "Dummy" message into *S_Set*. In effect, this allows a continuous stream of messages even when the user does not send messages. If the receiver protocol "sieves" out such "Dummy" messages, it will not affect the correctness of the protocol. However, Theorem 4.7 applies to this modified protocol because we can bound the time taken for the protocol to stabilize.

4.3 Improved Efficiency for Large Messages

The simplest message delivery protocol is to use the majority encoder we described earlier Since each packet within a message can travel n hops, the resulting communication complexity is $O(Cn) = O(n^2m)$. The normal and stabilization time complexity is $O(Xn^2m)$ which in this case is $O(n^4m^2)$.

If the protocol has to send large messages (that can be modeled as a group of smaller messages) the message complexity can be improved considerably. The basic idea is to send a large number of messages in a batch to amortize the effect of old messages. Since the network reorders packets we append to each packet its position in the batch (this is why we included a *position index* in the packet format) in order to reconstruct the order at the receiver. It is easy to see that if we send a batch of size X then the receiver (after reconstruction) will receive at least $X - 3C$ correct messages in correct positions. The receiver decodes using a batch of $X - C$ packets of which C can be "bad" packets. Thus the code must reconstruct the message using $X - 2C$ original packets and up to C bad packets. But the C bad packets can collide with the position indices of up to C original packets, causing a total of $3C$ errors. All that remains is to add an error-correcting code to every batch that corrects for up to $3C$ errors.

The amortized message complexity of this second protocol is $O(n)$ if we have blocks of size at least $X = \Omega(C \log C)$ since it takes $O(\log C)$ extra messages to correct each message error (using, say BCH codes). The normal and stabilization time complexity for the second protocol is only slightly higher – i.e., by a factor of $\log C$. A formal description of the large message encoding is in [APV95].

References

[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, October 1987.

[AB89] Yehuda Afek and Geoffrey Brown. Self-stabilization of the alternating bit protocol. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 80–83, 1989.

[AE86] Baruch Awerbuch and Shimon Even. Reliable broadcast protocols in unreliable networks. *Networks*, 16(4):381–396, Winter 1986.

[AG88] Yehuda Afek and Eli Gafni. End-to-end communication in unreliable networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 131–148. ACM SIGACT and SIGOPS, ACM, 1988.

[AGR92] Yehuda Afek, Eli Gafni, and Adi Rosen. Slide - a technique for communication in unreliable networks. In *Proceedings of the 11th PODC*, Vancouver, British Columbia, August 1992.

[AMS89] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. End-to-end communication with polynomial overhead. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989.

[APV95] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilizing end-to-end communication. Technical Memo to appear, MIT, Lab. for Computer Science, October 1995.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comput. Syst.*, 3(1):63–75, February 1985.

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.

[Fin79] Steven G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Trans. on Commun.*, COM-27(6):840–845, June 1979.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[MMT91] M. Merritt, F. Modugno, and M.R. Tuttle. Time constrained automata. In *CONCUR 91*, pages 408–423, 1991.

[Per83] Radia Perlman. Fault tolerant broadcast of routing information. *Computer Networks*, December 1983.

[Ros81] E. C. Rosen. Vulnerabilities of network control protocols: An example. *Computer Communications Review*, July 1981.

[RS] Thomas Rodeheffer and Michael Schroeder. Automatic reconfiguration in the Autonet. Proceedings of the 14th Symposium on Operating Systems Principles, Nov 1993.

[Sch93] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25, March 1993.

[Var93] George Varghese. Self-stabilization by local checking and correction. Ph.D. Thesis MIT/LCS/TR-583, Massachusetts Institute of Technology, 1993.