Crash Resilient Communication in Dynamic Networks[*]

Shlomi Dolev[†] Jennifer L. Welch[‡]

July 2, 1996

Abstract

An end-to-end data delivery protocol for dynamic communication networks is presented. The protocol uses bounded sequence numbers and can tolerate both link failures *and* (intermediate) processor crashes. Previous bounded end-to-end protocols could not tolerate crashes.

We present a self-stabilizing version of the algorithm that can recover from crashes of the sender and the receiver as well as of intermediate processors. Starting with the network in an arbitrary state, the self-stabilizing version guarantees proper transmission of messages following a finite convergence period.

Key Words and Phrases: communication networks, end-to-end protocols, dynamic networks, crash failures, self-stabilization.

[*]This work was supported by NSF Presidential Young Investigator Award CCR-91-58478, funds from the Texas A&M University College of Engineering and by the Israeli ministry of science and arts grant #6756196. A preliminary version of this work was presented at the 7th Workshop on Distributed Algorithms, September 1993.

[†]Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. e-mail: dolev@cs.bgu.ac.il.

[‡]Department of Computer Science, Texas A&M University, College Station, Texas 77843, e-mail: welch@cs.tamu.edu. Contact author. Phone: 409-845-5076. Fax: 409-847-8578

1 Introduction

A basic communication task in any network is *end-to-end* communication, that is, delivery in finite time of data items generated at a designated sender processor, to a designated receiver processor, without duplication, omission or reordering of data items. End-to-end communication is easy to achieve in a reliable network, where links never fail and processors do not crash. However, in existing communication networks both link failures and processor crashes are possible. A network that is subject to such failures is called a *dynamic network*.

One approach to constructing end-to-end protocols for dynamic networks is to use unbounded sequence numbers to uniquely identify the data items sent by the sender. Such an approach is used in the protocol of [4]. The use of unbounded sequence numbers implies that both message size and the amount of memory needed will grow with the number of data items transmitted. Therefore, much effort has been spent in designing end-to-end protocols that use bounded sequence numbers.

An important aspect of an end-to-end communication protocol is the type of faults that it can tolerate. Clearly the end-to-end task is unsolvable when there is a permanent sender-receiver link cut of the network such that all of its links are down forever. Thus, some assumption on the behavior of faulty links is necessary. Three common assumptions in the literature are:

- *infinitely frequent stability:* Infinitely often the network topology stabilizes for a period of time and there is no sender-receiver link cut in this stabilized topology (e.g., [2, 11]).
- *infinitely frequent path stability:* Infinitely often there is a period of time during which links forming at least one path between the sender and the receiver are operating (e.g., [7, 17]).
- *eventual connectivity:* The only assumption is that there is no permanent sender-receiver link cut — or equivalently, there exists at least one "viable" path between the sender and the receiver, a path that contains no permanently faulty link (e.g., [5, 9, 6, 8]).

Almost all existing end-to-end protocols depend on having physical links that are, or can be made to be, "well-behaved" in that the sequence of messages delivered is always a prefix of the sequence sent, i.e., no messages are lost in the middle. If processors do not crash, then this behavior can be ensured by running the alternating bit protocol [12]. But if processors can crash, then this good behavior cannot be achieved without keeping information, including the message currently being transmitted, in stable storage.[1]. A message that is transferred in the network can be instantly lost if the processor that receives it crashes (after acknowledging the sending neighbor of the message arrival). In addition to the the major problem of losing messages because of crashes, the data link protocol that forwards the message from one processor to the next cannot function correctly in the presence of crashes. Even if only a weaker behavior of the data link protocol is required, namely that once there are no more crashes of the end points, the sequence delivered is a prefix of the sequence sent, either stable storage is required as proved in the impossibility results of [16] or a bound on the capacity of the link must be known, in a similar fashion to the data link protocol presented in [3]. Unfortunately, in existing dynamic networks processors may repeatedly crash and recover losing the contents of their memory, including messages received and the state of the data link protocol. This violation of the assumption made by the end-to-end protocols of [5, 9, 6, 8] may result in a violation of the requirements, e.g., the loss of data items.

We would like to have a protocol that is resilient to crashes of the intermediate processors, i.e., those processors in the network other than the designated sender and receiver, and does not rely on stable storage. For the first version of our protocol, we exclude the possibility of the sender and receiver failing, since if they do, stable storage would be required by the same argument alluded to above for a physical link. The second version of our protocol, which is self-stabilizing, can recover following crashes of the sender or the receiver, if the capacity (i.e., number of messages in transit) of the communication links is bounded and known.

In the presence of processor crashes the previous definition of eventual connectivity is not sufficient for the existence of a protocol, since a permanent cut of any combination of crashed processors and crashed links could eliminate the connection between the sender and the receiver. Thus, we make the weakest assumption possible, namely that there exists at least one "viable" path between the sender and the receiver, a path that contains no permanently faulty link and no processor that is permanently crashed (for this path), i.e., at least one path along which communication is possible. Thus, our protocol works under more severe conditions than the protocols that assume eventual connectivity. We call this new setting *eventual connectivity in the presence of processor crashes*.

A elegant approach to designing a communication protocol is to view the network on which it will run as a black box that provides a message transfer service in which messages can be lost, reordered and duplicated, but not corrupted.[2] However this approach has

[1]The only known end-to-end protocol that can withstand processor crashes (without stable storage) is the randomized protocol presented in [17]. However, this protocol has a bounded probability of failure and uses headers that are not strictly bounded

[2]In practice, there is a non-zero probability of a message being corrupted, however due to error detection schemes it is assumed that those messages are identified and discarded.

limitations: [18] show that no bounded sequence number protocol can tolerate reordering and duplication, while [1] show that although there is a bounded sequence number protocol that can tolerate reordering and loss, any such protocol must have the property that the number of messages needed to transmit a data item increases without bound.

These impossibility results hinge on the assumption that the black box network can reorder messages arbitrarily. This assumption models the situation when the user of the network does not know how the network layer protocols (which implement the black box) route messages or even what the network topology is. An alternative approach, which avoids arbitrary reordering, is to use knowledge of the network topology and explicitly control the retransmissions on the routes. A packet can indeed be lost while it is traveling over a (physical) link. However, duplications are caused by *protocols* that retransmit packets under certain circumstances; if no packet is retransmitted by a protocol, then no duplication exists. Reordering is also a *protocol* property. If the protocol uses only a single path from the sender to the receiver and a single path from the receiver to the sender, the FIFO property in each direction is preserved by the network. Thus our protocol is designed to work on top of the "bare" network, consisting of nodes connected by FIFO non-duplicating links that can lose messages. Although our protocol does retransmit and uses multiple paths, and thus messages are duplicated, reordered, and lost, these activities are carefully coordinated.

Contribution of this paper: In this paper we define the eventual connectivity in the presence of processor crashes paradigm. We present two end-to-end protocols for dynamic networks that can tolerate crashes of nodes and failures of the links in the communication network and use only bounded sequence numbers. The protocols presented do not require any stable storage. Our first protocol does not assume knowledge of the link capacities (assuming that the sender and the receiver do not crash). Our second protocol is self-stabilizing; This protocol does assume knowledge of the link capacities in order to tolerate crashes of processors including crashes of the sender and the receiver. The existence of the above protocols proves (somewhat, surprisingly) that end-to-end communication is such sever environment is possible.

The protocols use source routing (i.e., each message sent has its entire path specified by the sender) instead of flooding, which is used in most other end-to-end protocols (e.g., [5, 7]). The space complexity of the protocol, i.e., the maximum amount of space used by any processor's program, is $O(\mathcal{P}^2 \log \mathcal{P})$, where $\mathcal{P}$ is the number of simple paths in the network. The message size is $O(\mathcal{P} \log \mathcal{P})$ bits. The time complexity is $O(L)$, where L is the length of a viable path, and the message complexity is $O(n\mathcal{P}(1 + L/T))$, where n is the number of processors in the system, and T is the retransmission parameter. Roughly speaking the time and message complexity are measured only for runs in which the time for a message to travel over a viable link is 1; this is comparable to the complexity measures in [8] (which assumes a reliable data link layer). A comparison with previous

Reference	Communication Complexity	Space Complexity	Assumptions
[AE86]	∞	∞	eventual stability
[AG88]	$O(exp(n))$	$O(\log n\Delta)$	eventual connectivity
[AMS89]	$O(n^9)$	$O(n^5\Delta)$	eventual connectivity
[AGR92]	$O(n^2\|E\|)$	$O(n\Delta)$	eventual connectivity
Present work	$O(n\mathcal{P}^2 \log \mathcal{P}(1 + L/T))$	$O(\mathcal{P}^2 log\mathcal{P})$	eventual connectivity in the presence of crashes

Figure 1: Comparison of Routing Schemes

protocols appears in Figure 1. In Figure 1 n, $|E|$ and Δ respectively denotes, the number of processors, the number of communication links and the maximal number of links connected to a single node.

The number of possible paths $\mathcal{P}$ between two processors in the system is theoretically exponential in $|E|$, the number of links in the system. This could be a drawback of our protocol. However, most practical communication schemes are based on sending messages along a single route from source to destination (see [17] for a nice discussion of practical protocols). Thus our protocol is not only of theoretical interest but could be used to improve existing protocols by using a constant number of paths from the source to the destination; as long as at least one of them is viable then the data items will be delivered. Moreover, the delivery time during each time interval will be due to the fastest and most reliable path during that period.

Another application of our protocol is to the case of parallel *physical* links between two processors. Such application would provide an implementation of a very reliable (non-parallel) link. In this situation, the number of paths would obviously be extremely small.

The remainder of the paper is organized as follows. In the next two sections we formalize the network and requirements, respectively. The protocol is presented in Section 4. Section 5 contains the self-stabilizing version of the protocol. Concluding remarks are in Section 6.

2 The Bare Network

We model a communication network as a graph $G(V, E)$, $|V| = n$, $|E| = m$, where the nodes are processors and the edges are undirected communication links. Each undirected link consists of two directed links, delivering messages in opposite directions.

Each communication link connects two processors. Two processors that are connected by a link are called *neighbors*. The communication over the links obeys the FIFO discipline, and no bound on the transmission delay is known. Every processor knows the full network topology — the topology that includes all the processors and links. However, the processor does not know the status of the processors (crashed/active) or links (faulty/operational).

Each processor in the system is viewed as a state machine executing a program. An execution of a program consists of a sequence of *steps*. Each step consists of (1) one *receive* operation, during which zero or one message is received, (2) internal computations, and (3) zero or more *send* operations. The only exception is the first step in the execution, which does not include a receive operation. The internal computation of the sender can include the input of data items, while the internal computation of the receiver can include the output of data items.

We assume that the sender and receiver are not subject to crashes. Any other processor is called an *intermediate* processor. Intermediate processors are subject to *crashes*. Following a crash, a processor reenters its initial state and it may continue executing. The crash of an intermediate processor could occur in the middle of a step, modeled as a *partial* step, in which only a subset of the messages that should have been sent are actually sent.

We model the link between processors P and Q as two FIFO queues, one holding the messages in transit from P to Q and the other holding the messages in transit from Q to P. Links are subject to failures; a link failure causes one or more messages to be eliminated from the component queues.

A *configuration* of the system is the set of states of the processors and the contents of the messages in the links. A *run* is a sequence of configurations $c_0, c_1, c_2, \ldots$ such that c_0 is an initial configuration (each processor is in its initial state and all the links are empty), and for each i, in going from c_i to c_{i+1}, one of the following holds.

- Some processor P takes the next step (possibly a partial step) according to its program: the message received, if any, is at the head of the relevant queue in c_i and is dequeued in c_{i+1}, P changes state accordingly (enters its initial state if this is a partial step), and the messages sent are enqueued in c_{i+1}. Nothing else changes.

- A link fails: the only change is that one or more of the messages that are in the queues of a particular link in c_i are no longer there in c_{i+1}.

We are only going to be concerned with runs that satisfy certain basic conditions, as given now. A run is *admissible* if

- the sender takes an infinite number of steps,
- the receiver takes an infinite number of steps,
- there exists at least one *viable* path between the sender and the receiver.

It remains to define a *viable* path. Assume an infinite run satisfying the first two admissibility conditions. An intermediate processor P is *viable* provided whenever P receives a message m infinitely often, it succeeds in sending m infinitely often[3]. A link is *viable* provided whenever an infinite number of messages is sent on that link, then an infinite number of messages is received by the receiving processor. We assume this is true in both directions of the link. (Note that our definition of a viable link implies that the receiving processor must take an infinite number of steps or partial steps.) A path is *viable* if every intermediate processor and every link on the path is viable. Our definition of viability is weaker than that in [8] since the latter does not consider processor crashes. If there is no viable path in the network between the sender and receiver, then every path between them either has a nonviable processor or a nonviable link, and thus there is a sender-receiver cut.

Note that there are no restrictions concerning relative ordering of processor steps or the number of steps between the sending of a message and its receipt. Thus we have an asynchronous system.

We assume that whatever entity supplies the data items to the sender provides one when and only when the sender requests one.

3 Problem Statement

An algorithm solves the *end-to-end communication problem* if in every admissible run the following properties are satisfied:

Safety: In any prefix of the run, the sequence of data items output by the receiver is a prefix of the sequence of data items input by the sender.

Liveness: The receiver does an infinite number of outputs.

[3]This definition restricts the behavior of the intermediate processor in ways that are not compatible with some systems, such as those that are supposed to strip off part of the header or send m on more than one link. A more general definition of a viable processor would be: upon receiving a message m infinitely often P executes infinitely often a (complete) step that starts with receiving m. However, for our protocol the weaker condition of infinite execution of a *partial* step that succeeds in forwarding m is sufficient.

The time and message complexity are only discussed for ideal runs, which we now define. An admissible run is *ideal* if

1. Whenever a processor has taken k steps, at least T real time has elapsed.
2. If processor P receives message m at time t which is addressed to travel through a viable path Path through P's neighbor Q, then Q receives m at time t', $t < t' \leq t+1$.

We now elaborate on the above definition of an ideal run. Processors send messages either in response to receiving messages or based on some timeout parameter (in order to guard against message loss). We say that a message is *retransmitted* if it is sent based on a timeout expiring; otherwise it is *new*. The time between retransmissions is a function of the speed of the processor. Naturally, the frequency of retransmission influences the total time and number of messages needed to deliver a data item. At each processor there is a procedure responsible for sending messages. The Send procedure keeps track of the number of steps, k, that the processor has taken in order to estimate when T real time has elapsed, where T is some system-dependent value (that depends on, say, the probability of loss and the distribution of message delays on links). We want this estimate based on k to be a good one; this is the rationale for condition 1 of the definition of ideal. The goal of retransmitting every T units of time is to ensure that a message will be delivered within one time unit, assuming the link is viable. This is the rationale for condition 2 of the definition of ideal.

We are interested in these complexity measures:

- space: the maximum amount of space required by any node's program,
- message size: the maximum number of bits in any message.
- message number: the maximum number of messages sent to transfer a data item, in any *ideal* run, between two successive inputs.
- time: the maximum length of time between two successive inputs, in any *ideal* run.

4 The Protocol

The main problem that our protocol has to cope with is keeping track of the set of sequence numbers that label messages in transit in the network. The key observation is that the FIFO property of the links can be used to ensure FIFO delivery in every path from the sender to the receiver (and vice versa). If each message is sent with the description of the path it should take, then messages that are sent through the same path obey the FIFO order. Note that two different paths might have some shared links. Thus, the set of messages in transit in some link could be related to more than one path. The order of two messages within one link does not necessarily imply anything about the relative order of their sending or receipt, unless the two messages have the same path.

The sender and receiver use a set of $2\mathcal{P}+1$ sequence numbers (shortly it is explained why this number is sufficient). Both the sender and receiver remember the sequence number of the most recent message sent and received on each path. The sender uses the arrays Sent_{SR} and Recv_{RS}, with an entry for each possible path. We use the following convention: the index SR of, say, Sent_{SR} denotes that the array Sent_{SR} stores information concerning the messages sent from S to R. Whenever the sender sends a message with sequence number SeqNum on path Path, the sender assigns Sent_{SR}[Path]:=SeqNum. Similarly, whenever the sender receives a message with sequence number SeqNum the sender assigns Recv_{RS}[Path]:=SeqNum. The receiver uses the arrays Sent_{RS} and Recv_{SR} for its bookkeeping.

The sender uses only *clean paths* to send a new data item with a new sequence number. A path, Path, from the sender to the receiver is considered clean when Sent_{SR}[Path]=Recv_{SR}[Path]. That is to say, the sequence number that is currently being repeatedly sent through that path has arrived at its destination. The use of clean paths ensures that the set of sequence numbers on the messages in any particular sender-receiver path, Path, is contained in (Sent_{SR}[Path] $\cup$ Recv_{SR}[Path]). Moreover, in case there are two sequence numbers in some path, Path, then the messages are ordered in that path such that the ones with sequence Sent_{SR}[Path] are closer to the sender than any message with sequence number Recv_{SR}[Path].

Unfortunately, the array Recv_{SR} is updated by the receiver and thus the value of Recv_{SR}[Path] is not known to the sender. Since the sender does not know which of the paths is clean and can be used for transmission it tries to receive information concerning all the possible paths from the receiver. Consequently, the receiver sends the array Recv_{SR} to the sender. However, the sender has to be able to distinguish old values of Recv_{SR} from more current values. This is done by implementing independent alternating bit protocols [12], one for each entire path from S to R (*not* for individual physical links). The alternating bit protocol uses the sequence numbers 0 and 1 for delivering information using a single FIFO link (or path). Whenever the sender receives an acknowledgment with a sequence number that is identical to the sequence number that the sender is currently sending, the sender inputs a new data item, alternates the sequence number bit (from 0 to 1, or vice versa) and starts repeatedly sending the new data item with the new sequence number. Whenever the receiver receives a sequence number that is different from the previous arriving sequence number, the receiver outputs the data item and continues to acknowledge every message with an acknowledging message that contains the sequence number of the arriving message.

Using the alternating bit protocol on a given path, the protocol keeps track of the number of *alternating bit tokens* (i.e., the number of times the sender receives an acknowledgment for the same bit it is currently trying to send) that have arrived at the sender over that path since the sender input the last data item. The sender uses the information concerning Recv_{SR} that arrives through some path only if it comes after the second token has arrived through the same path since the last data item was input. As we show in the sequel in Lemma 4.1 and Figure 8, this rule ensures that the information received reflects the current state of the path. Whenever such information concerning Recv_{SR} arrives, the sender updates its view of Recv_{SR} in an array called VRecv_{SR}. The prefix V of VRecv_{SR} stands for "virtual," signifying that the source is Recv_{SR}. Note that Recv and the SR subscript of Recv_{SR} indicate that Recv_{SR} is maintained by R and thus VRecv_{SR} is maintained by S. The sender uses the information in Sent_{SR} and VRecv_{SR} to determine whether a sender-receiver path is clean or not. Similarly, the receiver uses Sent_{RS} and VRecv_{RS} to determine whether a receiver-sender path is clean or not.

We view a run of the protocol as a sequence of alternating *stings*. The intuition behind our choice of the term sting is that the receiver may receive stream of messages with many different sequence number, but non of these messages will cause the receiver to deliver a message to its host. Only a message with a particular sequence number (called sting-tag) will cause the delivery of the data. Similarly, the sender may receive stream of (non-up-to-date) acknowledgments. The sender will start dealing with the next data item only after a particular sequence number (sting-tag) will arrive to it.

Clearly, the clean paths concept ensures that at most $2\mathcal{P}$ different sequence numbers are in each direction (the sender-receiver direction, denoted SR, and receiver-sender direction, denoted RS). Thus, an outside observer can easily choose a sequence number for $\mathrm{StingTag}_{SR}$, out of the $2\mathcal{P} + 1$ possible sequence numbers, that does not exist in transit. Assume for a moment that the sender can successfully choose such a sequence number for $\mathrm{StingTag}_{RS}$. In such a case how does the receiver identify this sequence number as a "new sequence number" (a sting) upon its arrival? Our approach is to let the receiver choose a non-existing sequence number in the SR direction, and suggests this number to the sender as a value for $\mathrm{StingTag}_{SR}$. To do so the receiver, upon arrival of a sting, chooses a sequence number value that is not in Recv_{SR}, nor in the last received value of Sent_{SR}.

At the beginning the sender uses sequence number 1 to sting the receiver with the first data item. The sender repeatedly sends a message with sequence number 1 through all possible paths. Eventually such a message arrives at the receiver, which uses sequence number 1 to sting the Sender with an acknowledgment. At the same time the receiver sends the next sequence number, called the sting-tag, that will be used by the sender to sting the receiver with the second data item. The receiver chooses this sequence number to be the minimal sequence number that does not appear either in Recv_{SR} or in VSent_{SR}; thus a sequence number set of size $2\mathcal{P}+1$ is big enough. In this case, both Recv_{SR} and VSent_{SR} include only the sequence number 1, and thus the receiver chooses 2 to be the next sequence number to be used by the sender to sting the receiver. The sender eventually is stung with an acknowledgment from the receiver when the first message with sequence number 1 arrives at the sender. Now the sender calculates the next sting-tag that will be used to sting itself (with an acknowledgment) by similar arguments as above; this sequence number will be 2. A detailed example is described in Section 4.1.

When the sender is ready to send the i'th data item (after getting the acknowledgment for the $i-1$'st data item), the sequence numbers in the entries of Sent_{SR} could all be distinct. Roughly speaking, each entry contains the last sequence number that was sent and has not yet cleaned its path (i.e., arrived at the other side). However, since there is at least one viable path Path, this path is eventually cleaned (by the sequence number Sent_{SR}[Path]). The sender eventually gets two tokens through this path and updates its VRecv_{SR} so that VRecv_{SR}[Path]=Sent_{SR}[Path]. Then the sender uses Path for sending the i'th data item. The i'th data item is sent with SeqNum_{SR} that was chosen by the receiver during the $i-1$'st acknowledgment.

Until the i'th acknowledgment arrives, the sender continues to update the clean/dirty status of the sender-receiver paths by the use of the information concerning Recv_{SR} that arrives with the messages from the receiver. When a sender-receiver path changes status to clean before the i'th acknowledgment arrives, the sender uses this path also to send the i'th data item with SeqNum_{SR}. The receiver uses a similar scheme to deliver the i'th acknowledgment.

The formal description of the protocol appears in Figures 3 through 5. The variables used by the sender and their initial values appear in Figure 2. The variables of the receiver are similar except the order of the subscripts S and R is reversed. When no confusion is possible we use the name of an array, e.g., Sent_{SR}, to represent the set of sequence numbers yielded from its entries. An intermediate processor has a data structure Pending which is an array of messages, with one entry for each *directed* path from S to R and from R to S; initially each entry is nil.

Description of the code of the sender (S), refer to Figure 3:

Line 02 — S inputs a data item from its host.

Line 03 — S computes the set of clean paths from S to R. Every path for which there is an evidence in VRecv_{SR} that the sequence number currently sent by S arrives to R is included in the set of the clean paths.

Line 04 — The StingTag_{RS} that will be sent to R (in order to sting S) is the minimal sequence number not in transit from R to S.

Lines 05 to 21 — This operation in this loop repeats until the StingTag_{RS} (computed in line 04) arrives in a message from R.

Lines 06 to 10 — S checks every path Path. If Path is a clean path then S starts sending a message with the current sequence number SeqNum_{SR} through it. Once S sends such a message the path is not clean anymore, yet S has to resend the message with SeqNum_{SR} through it. Therefore, a message with SeqNum_{SR} is sent through every clean path or a path through which the last message sent used SeqNum_{SR}. Otherwise, S executes line 10 in which the last sequence number sent on Path is re-sent in order to clean the Path. Note that the sequence number stored in Sent_{SR}[Path] does not sting the receiver.

Line 11 — S examines its incoming buffers for arriving messages. The result of such an examination is either, nil, if no message arrived, or a message Msg $\neq$ nil if a message is present in the incoming messages buffers.

Lines 12 to 20 — If a message did arrive then store its SeqNum in Recv_{RS}[Msg.Path]. Then decide on the bit of the alternating bit protocol to be used (lines 14,15) and count the number of tokens that arrived on this path (line 16) since, t_0, the time of the last sting arrival. If the number of arriving tokens, since the last arrival of a sting, is 2 then Msg.Recv_{SR} is a value reflecting a value of Recv_{SR} following t_0. In such a case, S updates VRecv_{SR}[Path] of every Path, if there is an indication that Path is a clean path (lines 18,19). In line 20 the set of clean paths is recomputed according to the new update.

Line 22 to 23 — Some initialization following the arrival of a sting including the use of Msg.Sent_{RS} for VSent_{RS}, assigning the recommendation of R to a new SeqNum_{SR} (line 23) and initializing the array Tkns_{RS} to 0.

Note that every time the sender executes line 11 of its code, the sender starts a new step.

Description of the code of intermediate processor, refer to Figure 4:

Variable Name	Type	Initial Value
CleanPaths_{SR}	: set of paths	; empty
Sent_{SR}	: array[1..$\mathcal{P}$] of integer 1..$2\mathcal{P}+1$	; all entries nil
VRecv_{SR}	: array[1..$\mathcal{P}$] of integer 1..$2\mathcal{P}+1$	; all entries nil
VSent_{RS}	: array[1..$\mathcal{P}$] of integer 1..$2\mathcal{P}+1$	; all entries nil
Recv_{RS}	: array[1..$\mathcal{P}$] of integer 1..$2\mathcal{P}+1$	; all entries nil
Bit_{SR}	: array[1..$\mathcal{P}$] of integer 0..1	; all entries 0
Tkns_{RS}	: array[1..$\mathcal{P}$] of integer 0..2	; all entries 0
SeqNum_{SR}	: integer 1..$2\mathcal{P}+1$	; 1
StingTag_{RS}	: integer 1..$2\mathcal{P}+1$	; arbitrary

Figure 2: The Variables of the Sender

Lines 02 to 04 — Examining the incoming messages buffers for a message. If such a message exists (Msg $\neq$ nil) then the intermediate processor determines the direction of the message from the path of the message and the incoming buffer it arrived to. If the direction is from S to R then the message is assigned to Pending[Msg.Path(SR)], otherwise it is assigned to Pending[Msg.Path(RS)]. Note that the value of X (and Y) in line 04 of Figure 4 is S or R where $X \neq Y$.

Lines 05 to 06 — the intermediate processor forwards the messages stored in the Pending array to the appropriate neighbors.

Note that every time an intermediate processor executes line 02 of its code, the intermediate processor starts a new step.

The description of the code of the receiver is similar to the one of the sender.

4.1 Execution Sample

In this section we will demonstrate the operation of the protocol on a network of six processors. The sender S the receiver R and four intermediate processors P_1 to P_4. The topology of the network is described in the left portion of Figure 6. Our protocol sends every message with the description of the path it should take. Thus, we can view the network as the right portion of Figure 6. There are three possible undirected paths: Path_1 is (S, P_1, P_2, R), Path_2 is (S, P_1, P_3, R) and Path_3 is (S, P_4, R).

```
do forever
  Input(DataItem)
  $\text{CleanPaths}_{SR}$ := { Path | $\text{Sent}_{SR}$[Path] = $\text{VRecv}_{SR}$[Path]}
  $\text{StingTag}_{RS}$ := min({Num | Num $\notin$ ($\text{VSent}_{RS} \cup \text{Recv}_{RS}$)})
  repeat
    $\forall$ Path
      if Path $\in$ $\text{CleanPaths}_{SR}$ or $\text{Sent}_{SR}$[Path] = $\text{SeqNum}_{SR}$ then
        $\text{Sent}_{SR}$[Path] := $\text{SeqNum}_{SR}$
        Send(Path,DataItem,$\text{SeqNum}_{SR}$,$\text{Bit}_{SR}$[Path], $\text{StingTag}_{RS}$,$\text{Sent}_{SR}$,$\text{Recv}_{RS}$)
      else Send(Path,nil,$\text{Sent}_{SR}$[Path],$\text{Bit}_{SR}$[Path],nil, $\text{Sent}_{SR}$,$\text{Recv}_{RS}$)
    Recv(Msg)
    if Msg $\neq$ nil then
      $\text{Recv}_{RS}$[Msg.Path] := Msg.SeqNum
      if $\text{Bit}_{SR}$[Msg.Path] = Msg.Bit then
        $\text{Bit}_{SR}$[Msg.Path] := not(Msg.Bit)
        if $\text{Tkns}_{RS}$[Msg.Path] $<$ 2 then $\text{Tkns}_{RS}$[Msg.Path] := $\text{Tkns}_{RS}$[Msg.Path] + 1
      if $\text{Tkns}_{RS}$[Msg.Path] = 2 or Msg.SeqNum = $\text{StingTag}_{RS}$ then
        $\forall$ Path s.t. $\text{Sent}_{SR}$[Path] = Msg.$\text{Recv}_{SR}$[Path]
          $\text{VRecv}_{SR}$[Path] := Msg.$\text{Recv}_{SR}$[Path]
        $\text{CleanPaths}_{SR}$ := { Path | $\text{Sent}_{SR}$[Path] = $\text{VRecv}_{SR}$[Path]}
  until Msg.SeqNum = $\text{StingTag}_{RS}$
  $\text{VSent}_{RS}$ := Msg.$\text{Sent}_{RS}$
  $\text{SeqNum}_{SR}$ := Msg.StingTag
  $\forall$ Path $\text{Tkns}_{RS}$[Path] := 0
od
```

Figure 3: The Program of the Sender

```
do forever
  Recv(Msg)
  if Msg $\neq$ nil then
    Pending[Msg.Path(XY)]:=Msg
  $\forall$ Neighbor
    Send to Neighbor all the messages in Pending s.t. Neighbor is their next hop
od
```

Figure 4: The Program of an Intermediate Processor

While describing the execution we choose a particular scenario, one of many possible scenarios.

We now follow Figure 7 to describe an execution sample. The description of the execution starts in the upper left corner portion, marked by (a), and follows the alphabetic order up to the portion denoted by (d).

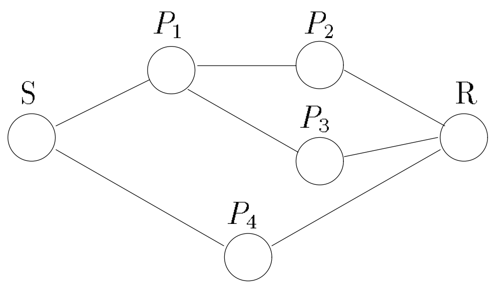
We start in the system configuration (a) in which R is stung by a message, Msg, with SeqNum=5, arriving through Path$_1$. In this configuration S maintains the following values: Sent$_{SR}$[Path$_1$]=5, Sent$_{SR}$[Path$_2$]=1, Sent$_{SR}$[Path$_3$]=7, Recv$_{RS}$[Path$_1$]=4, Recv$_{RS}$[Path$_2$]=3 and Recv$_{RS}$[Path$_3$]=5. The values R has in configuration (a) are: Sent$_{RS}$[Path$_1$]=2, Sent$_{RS}$[Path$_2$]=3, Sent$_{RS}$[Path$_3$]=6, Recv$_{SR}$[Path$_1$]=5, Recv$_{SR}$[Path$_2$]=6, and Recv$_{SR}$[Path$_3$]=2.

Upon the arrival of Msg to R (in the execution of line 13 of Figure 5) R updated Recv$_{SR}$[Path$_1$]=5. R outputs Msg.DataItem (line 24 of its code) and assigns VSent$_{SR}$ by Sent$_{SR}$ i.e. (5,1,7) (line 25 of its code). Then R uses Msg.StingTag as its next sequence number (used to sting the sender). A possible value (chosen by S) for Msg.StingTag is a value that is not in transit from R to S, say 1 (see the right side of configuration (a)). The set of CleanPath$_{RS}$ computed in line 05 (of Figure 5) is a function of the value of VRecv$_{RS}$. We choose to start with VRecv$_{RS}$=Recv$_{RS}$=(4,3,5). Thus, CleanPath$_{RS}$ includes only Path$_2$, for which Sent$_{RS}$[Path$_2$]= VRecv$_{RS}$=3. Then (in line 06) R computes StingTag$_{SR}$ to be 3 (VSent$_{SR}$ is equal to Sent$_{SR}$ at this stage).

Now R repeatedly sends a message with sequence number SeqNum$_{RS}$ = 1 over Path$_2$. If this path stops operating R will not be able to sting S through Path$_2$. If Path$_3$ is operating then it will be cleaned, i.e. all messages in transit from R to S will carry sequence number SeqNum$_{RS}$=6. This is a consequence of the repeated transmission of messages with SeqNum$_{RS}$=6 on Path$_3$ (in line 12 of Figure 5). This stage is depicted by configuration (b).

```
Wait until Recv(Msg $\neq$ nil)
  Output(Msg.DataItem)
  $\text{Recv}_{SR}$[Msg.Path] := Msg.SeqNum
  do forever
    $\text{CleanPaths}_{RS}$ := { Path | $\text{Sent}_{RS}$[Path] = $\text{VRecv}_{RS}$[Path]}
    $\text{StingTag}_{SR}$ := min({Num | Num $\notin$ ($\text{VSent}_{SR} \cup \text{Recv}_{SR}$)})
    repeat
      $\forall$ Path
        if Path $\in$ $\text{CleanPaths}_{RS}$ or $\text{Sent}_{RS}$[Path] = $\text{SeqNum}_{RS}$ then
          $\text{Sent}_{RS}$[Path] := $\text{SeqNum}_{RS}$
          Send(Path,Ack,$\text{SeqNum}_{RS}$,$\text{Bit}_{RS}$[Path],$\text{StingTag}_{SR}$,$\text{Sent}_{RS}$,$\text{Recv}_{SR}$)
        else Send(Path,nil,$\text{Sent}_{RS}$[Path],$\text{Bit}_{RS}$[Path],nil, $\text{Sent}_{RS}$,$\text{Recv}_{SR}$)
      Recv(Msg)
      if Msg $\neq$ nil then
        $\text{Recv}_{SR}$[Msg.Path] := Msg.SeqNum
        if $\text{Bit}_{RS}$[Msg.Path] $\neq$ Msg.Bit then
          $\text{Bit}_{RS}$[Msg.Path] := Msg.Bit
          if $\text{Tkns}_{SR}$[Msg.Path] $< 2$ then $\text{Tkns}_{SR}$[Msg.Path] := $\text{Tkns}_{SR}$[Msg.Path] + 1
        if $\text{Tkns}_{SR}$[Msg.Path] = 2 or Msg.SeqNum = $\text{StingTag}_{SR}$ then
          $\forall$ Path s.t. $\text{Sent}_{RS}$[Path] = Msg.$\text{Recv}_{RS}$[Path]
            $\text{VRecv}_{RS}$[Path] := Msg.$\text{Recv}_{RS}$[Path]
          $\text{CleanPaths}_{RS}$ := { Path | $\text{Sent}_{RS}$[Path] = $\text{VRecv}_{RS}$[Path]}
    until Msg.SeqNum = $\text{StingTag}_{SR}$
    Output(Msg.DataItem)
    $\text{VSent}_{SR}$ := Msg.$\text{Sent}_{SR}$
    $\text{SeqNum}_{RS}$ := Msg.StingTag
    $\forall$ Path $\text{Tkns}_{SR}$[Path] := 0
  od
```

Figure 5: The Program of the Receiver



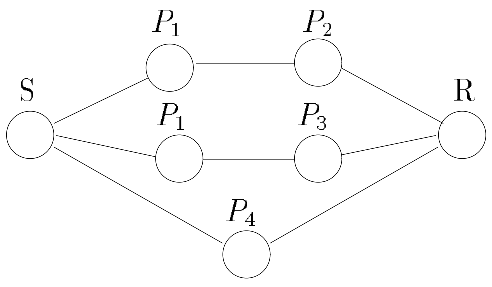


Figure 6: Equivalent network

R will discover that Path$_3$ is cleaned in the direction from R to S by receiving an updated Recv$_{RS}$: in line 19 of Figure 5 it is checked that Msg.Recv$_{RS}$ reflects the contents of Recv$_{RS}$ following the configuration (a) (See Lemma 4.1). Thus, when R receives the second (or later) token, say on Path$_1$, in the form of Msg, R uses Msg.Recv$_{RS}$ to update VRecv$_{RS}$[Path$_3$] by the sequence number 6 (line 21 of Figure 5). R includes Path$_3$ in CleanPath$_{RS}$ (line 22 of Figure 5) and starts repeatedly sending the message with sequence number SeqNum$_{RS} = 1$ through Path$_3$ as well (line 11) [4]. The message eventually reaches S and S is stung.

Configuration (c) immediately follows the arrival of the sting to S (line 11 of Figure 3). Note that during the execution that started in configuration (a), Path$_3$ has been cleaned in the direction from S to R as well, was identified as a clean path and messages with sequence number 5 were repeatedly sent through it. In lines 17 to 19 of Figure 3 VRecv$_{RS}$ is updated to (5,6,5). In line 22, VSent$_{RS}$ is assigned by (1,1,1) (assuming that Path$_1$ has been identified as a clean path by R in addition to the identification of Path$_3$). The sequence number used by S to sting R is Msg.StingTag=3 (line 23). S receives a new data item (line 02), compute CleanPaths$_{SR}$ to be Path$_1$ and Path$_3$. S Chooses StingTag$_{RS}$ to be 4 (i.e, the minimal number not in (1,1,1) (2,3,1)). Then S starts sending messages with sequence number 3 on Path$_1$ and Path$_3$ as depicted in configuration (d).

4.2 Correctness Proof

Throughout this section we assume an admissible run $R = c_0, c_1, c_2, \ldots$. Step i is the step that causes the transition from c_{i-1} to c_i in R. Recall that each step consists of (1) one *receive* operation, during which zero or one message is received, (2) internal

[4]Note that although Path$_3$ is not clean after the first such transmission, the condition in line 09 continues to hold since Sent$_{RS}$[Path$_3$]=SeqNum$_{RS}$.

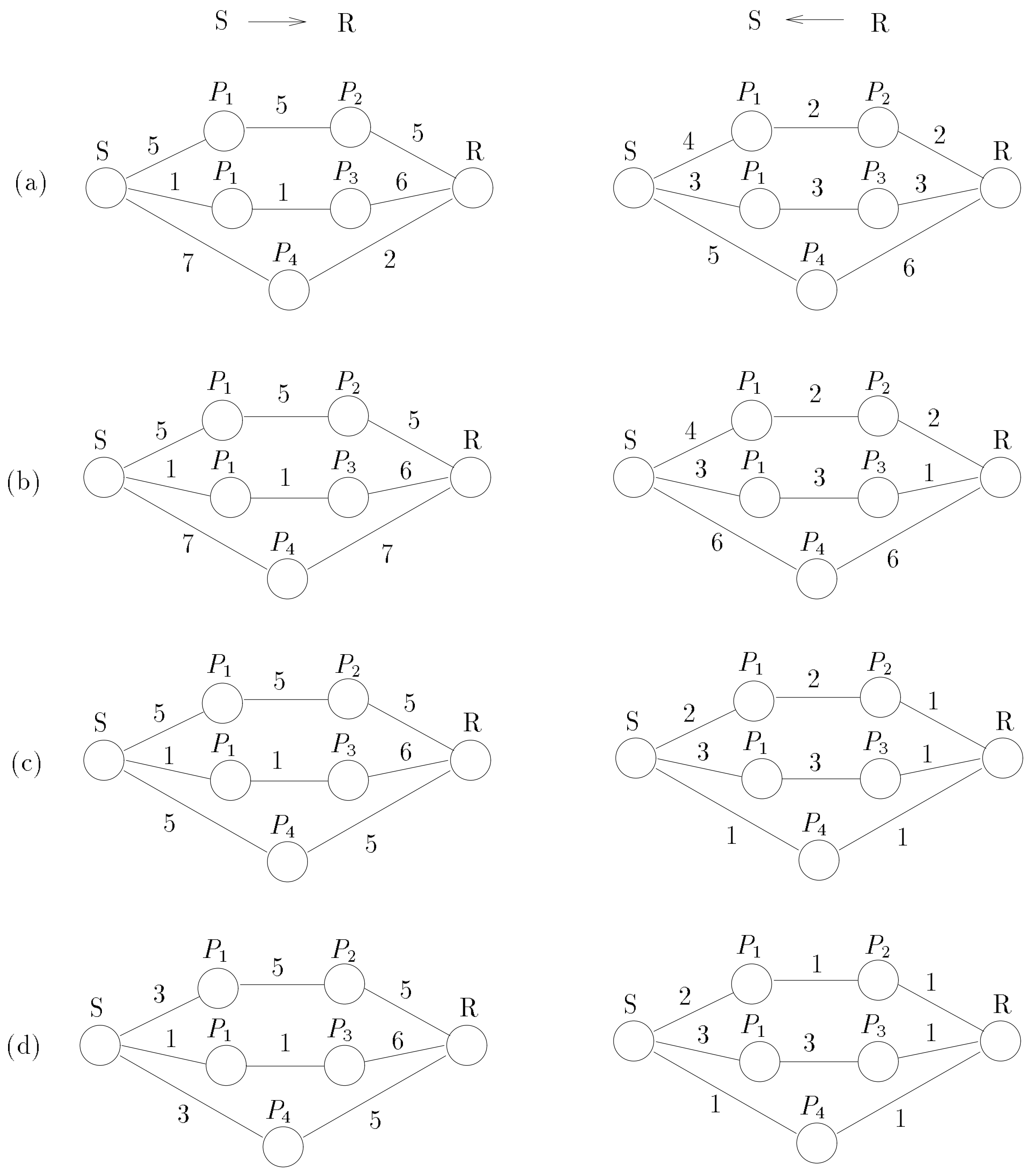


Figure 7: An execution sample.

computations, and (3) zero or more *send* operations. The internal computation of the sender can include the input of data items, while the internal computation of the receiver can include the output of data items.

We denote by Var(i) the value of variable Var in configuration c_i.

The correctness proof is by induction on the number of *stings* in R. The sender is stung whenever the sender executes a step during which it executes line 02 in Figure 3. The receiver is stung whenever the sender executes a step during which it executes line 02 or line 24 in Figure 5.

Each copy of the alternating bit protocol is implemented for a distinct undirected path. Consider a single such copy of the alternating bit protocol for undirected path Path. The *token* of the alternating bit protocol arrives at the sender when the sender receives a message with the same bit as the sender is currently sending. The token arrives at the receiver when the receiver receives a message with the opposite bit to the last bit the receiver sent. Given a configuration we denote for a path Path the sequence of messages from the sender to the receiver ($i \geq 0$ messages) and the sequence of messages from the receiver to the sender ($j \geq 0$ messages) together with the current values of $\text{Bit}_{SR}[Path]$ and $\text{Bit}_{RS}[Path]$ in the following form :
BitSeq $\equiv$ ($\text{Bit}_{SR}[Path]$, msg^1_{SR}, msg^2_{SR}, $\ldots$, msg^i_{SR}, $\text{Bit}_{RS}[Path]$, msg^1_{RS}, msg^2_{RS}, $\ldots$, msg^j_{RS})

It is well known (e.g., [12]) that if messages are only lost or duplicated then in any configuration of the alternating bit protocol if BitSeq contains more than a single value (either 0 or 1) then there exists a single border in BitSeq between those values. Furthermore, if there is no such border then the sender eventually changes the value of Bit_{SR} so that a border is produced. If there is a border and we look at the sequence BitSeq in successive configurations of the run, then this border "travels" towards the end of BitSeq.

The important property of the alternating bit, used by the protocol, is that between any two successive token arrivals at the sender there is one token arrival at the receiver. Figure 8 demonstrates the behavior of the alternating bit protocol and the importance of the second token arrival. The figure consists of the sender (marked S) and the receiver (marked R). The two directed lines represent an undirected Path. The circles represents messages where white (black) circles represent messages with value 0 (1, respectively) in the Bit field. Similarly, the circles inside S and R represent the values of Bit_{SR} (maintained by S) and Bit_{RS} (maintained by R). The figure contains a description for seven configurations (a) to (g). It starts at time t_0, immediately after S was stung and assigned 0 to Tkns_{RS}. An important property of the protocol is that S updates VRecv_{SR} only due to a value of Msg.Recv_{SR} that *follows* t_0. In the configuration marked (a), $\text{Bit}_{SR} = 0$ and hence S is waiting for a message with Bit=0 to arrive before changing the value of Bit$_{SR}$. The messages in transit from S to R are all with Bit=0 while two messages (that will arrive first to S) are with Bit=1. Configuration (b) follows the acceptance of a message with Bit=1, by S. Configuration (c) is derived from (b) by performing message send operation by S, message send and receive by R, and two message receives by S. The last message receive of S contains Bit=0 and hence is a token arrival — i.e. a message that causes a change in Bit$_{SR}$ (or Bit$_{RS}$). Note that this token passed R before t_0 and thus the Msg.Recv$_{SR}$ of the message might not be up to date. Configurations (d) to (g) demonstrate how the token (pointed by an arrow) travels through R (collecting an updated Recv$_{SR}$ and arriving at S).

We conclude the following lemma.

Lemma 4.1 *Suppose i and j are such that a message Msg arrives at the sender over path P in step $j > i$ with the token for P, and this is not the first token for P that the sender has received since step i. Then there exists k, $i < k < j$, such that Msg.Sent$_{RS}$ = Sent$_{RS}(k)$ and Msg.Recv$_{SR}$ = Recv$_{SR}(k)$. The analogous property is true for the receiver as well.*

The next lemma states that between consecutive stings at the sender (resp., receiver), the set of values in {SeqNum$_{SR}$}$\cup$ Sent$_{SR}$ (resp., {SeqNum$_{RS}$}$\cup$ Sent$_{RS}$) either remains the same or decreases. It can be seen to be true by inspecting the code, since the sender only changes an entry in Sent$_{SR}$ by sending a message with sequence number equal to SeqNum$_{SR}$.

Lemma 4.2 *For any i such that step i is not a sting at the sender, $\{SeqNum_{SR}(i)\} \cup Sent_{SR}(i) \supseteq \{SeqNum_{SR}(i+1)\} \cup Sent_{SR}(i+1)$. The analogous property is true for the receiver as well.*

Both the safety and liveness properties are proved by induction on the number of stings in R.

Lemma 4.3 *In any configuration for every sender-receiver path P, the sequence of sequence numbers in the messages in transit along P, in order starting with those closest to the sender, has the form $x^j y^k$ for some $j, k \geq 0$, where x = Sent$_{SR}$[P] and y = Recv$_{SR}$[P].*

Proof: The proof is by induction on the sting index i. We assume that the lemma holds till the i'th sting and prove for the i'th $+2$.

Basis: ($i = 1$). The sender is the first to be stung (executing line 02 of its code). At this time the values of Sent$_{SR}[P]$ and Recv$_{SR}[P]$ are *nil* and there is no message in transit. Thus, the claim of the Lemma holds for $i = 1$.

Induction: Assume for all $j \leq i$ and show for $i + 2$. We'll show it for i odd. Then $i + 1$ is even. Let t_i be the index of the step when the i-th sting in S occurs.

Consider any step t of the sender after step t_i. Assume the statement is true in all preceding configurations. We'll show it's true in configuration c_t. Pick any path P. If the message sent on P during step t has the same sequence number as in Sent$_{SR}[P](t-1)$, then clearly the induction assumption holds in c_t.

Suppose the message sent on P has a sequence number that is different from Sent$_{SR}[P](t-1)$. Then, during the step t_i line 08 of the sender's code has been executed. This is only possible when P has been a member in the CleanPaths$_{SR}(t)$ set. Moreover, once line 08 is executed it is not executed until the sender starts using a new SeqNum$_{SR}$ i.e. the sender is stung. Note that S is not stung during t_i and hence SeqNum$_{SR}[P](t-1)$= SeqNum$_{SR}[P](t)$, and both are not equal to Sent$_{SR}[P](t-1)$. Let x be the value of Sent$_{SR}[P](t-1)$. Since P is in CleanPaths$_{SR}$, Sent$_{SR}[P]$ = VRecv$_{SR}[P]$. Now look at the step between steps t_i and t when VRecv$_{SR}[P]$ was updated, causing P to be put in CleanPaths$_{SR}$. This was when for some path the second token was received since t_i. By Lemma 4.1, this information reflects the value of Recv$_{SR}[P]$ at the receiver in some configuration $c_{t'}$ with $t' > t_i$. Since $t' < t$, the inductive hypothesis holds and all the messages in path P have the same sequence number, namely x. As long as that is the only sequence number in P, Recv$_{SR}[P]$ will continue to be x. Thus in configuration c_{t-1}, Sent$_{SR}[P]$ = S = Recv$_{SR}[P]$ and all messages on P have sequence number x. So the induction hypothesis holds in configuration c_t. ■

Similarly, exchanging for and receiver-sender path and proving for the even number stings.

Lemma 4.4 *In any configuration, for every receiver-sender path P, the sequence of sequence numbers in the messages in transit along P, in order starting with those closest to the receiver, has the form $x^j y^k$ for some $j, k \geq 0$, where x = Sent$_{RS}[P]$ and y = Recv$_{RS}[P]$.*

Now we prove the correctness of the protocol.

Theorem 4.5 *The above protocol is a crash resilient end-to-end protocol.*

Proof: Both the safety and liveness properties are proved by induction on the number of stings in R. We will prove the following.

For all $i \geq 1$:

1. There are i stings.
2. If i is even then
   a. The i-th sting occurs at the receiver and causes the $i/2$-th data item to be output; the data of this output is equal to the data of the previous input.
   b. In every configuration between the $(i-1)$-st and i-th stings, if message Msg is in transit from the sender to the receiver and Msg.SeqNum = Sting_{SR}, then $\text{Msg.Sent}_{SR} \supseteq \text{Sent}_{SR}$.
3. Similarly, if i is odd then
   a. The i-th sting occurs at the sender and causes the $((i-1)/2+1)$-st data item to be input.
   b. If $i > 1$ then in every configuration between the $(i-1)$-st and i-th stings, if message Msg is in transit from the receiver to the sender and Msg.SeqNum = Sting_{RS}, then $\text{Msg.Sent}_{RS} \supseteq \text{Sent}_{RS}$.

We now prove this statement.

Basis: $(i = 1)$. Obvious from the code or initialization, or else vacuously true.

Induction: Assume for all $j \leq i$ and show for $i+1$. We'll show it for i odd (the case for i even is similar and left to the reader). Then $i+1$ is even. Let t_i be the index of the step when the i-th sting in R occurs.

To show 2a: We must show that the sender is never stung after step t_i as long as the receiver is not stung. I.e., we must show that every message received by the sender after step t_i has sequence number not equal to $s = \text{Sting}_{RS}(t_i)$. (Note that Sting_{RS} is only changed when the sender is stung.) s is chosen to be not in $\text{VSent}_{RS}(t_i)$ and not in $\text{Recv}_{RS}(t_i)$. Note that when the i-th sting arrives, the sender sets VSent_{RS} according to the information in the sting message M. By the inductive hypothesis (3b), since M is in transit just before the sting, $M.\text{Sent}_{RS}$ is a superset of $\text{Sent}_{RS}(t_i - 1)$, and thus $\text{VSent}_{RS}(t_i)$ is a superset of $\text{Sent}_{RS}(t_i - 1) = \text{Sent}_{RS}(t_i)$. By Lemma 4.4 any message in transit from R to S in configuration $c_{t_i - 1}$ has a sequence number that is in either $\text{Sent}_{RS}(t_i - 1)$ or $\text{Recv}_{RS}(t_i - 1)$. Even though Recv_{RS} is changed for M's path during the sting, any message on that path following M has a sequence number that is already included in $\text{Sent}_{RS}(t_i - 1)$ or has the same sequence number as M and thus is included in $\text{Recv}_{RS}(t_i)$. So no message that is in transit in configuration c_{t_i} will sting the sender.

Now we must show that no message that is sent by the receiver after step t_i can sting the sender. This follows from Lemma 4.2 and Lemma 4.4.

So the $(i+1)$-st sting, if it occurs, occurs at the receiver. By the inductive hypothesis (2a) for $i-1$, this would be the $(i-1)/2+1 = (i+1)/2$-th output. It is straightforward to check that the data is correct.

To show 2b: Immediately before the i-th sting, no message in transit from the sender to the receiver has SeqNum = $\text{Sting}_{SR} = s$, because when the receiver chose s as its next sting-tag back at sting $i-1$, no message with SeqNum s was in transit from the sender to the receiver.

Any message sent after sting i with SeqNum s has the current value of Sent_{SR} attached to it and by Lemma 4.2, the current set of elements in Sent_{SR} can only shrink relative to what was sent in the message.

To show 3: Vacuous since $i+1$ is even.

To show 1: We must show that eventually the receiver is stung after t_i. I.e., the receiver receives a message with sequence number equal to Sting_{SR}. Suppose this is not true. First, note that Sting_{SR} is only changed when the receiver is stung. After t_i, the sender gets at least two tokens over a viable path P, then puts P in CleanPaths_{SR}, and sends the current message on P (by Lemma 4.1). This current message has SeqNum = Sting_{SR}. Eventually a copy of this message will get through to the receiver, a contradiction. ■

4.2.1 Complexity Measures:

The message length is $O(\mathcal{P} \log \mathcal{P})$ since each message consists of a constant number of components, the largest of which are the Sent and Recv arrays, each of which consists of $\mathcal{P}$ entries of size $\log \mathcal{P}$ (assuming the data items are no bigger than this). The space complexity, due to the intermediate processors' storing a message for each path, is $O(\mathcal{P}^2 \log \mathcal{P})$.

Recall that the time and message complexity are defined for ideal runs. First we explain in more detail when messages are retransmitted by the Send procedure. The Send procedure keeps track, in Pending, of the most recent message that it has sent for each path. Whenever the Send procedure is executed, it does the following. First, consider the message that is the input to the procedure. The message is *fresh* if the last message sent on that path, which is stored in the Pending array, differs from this one in a component other than the Sent and Recv arrays. The message is stored in Pending (actually this has already been done in the intermediate nodes, but needs to be done here for the sender and receiver). If the message is fresh, then it is immediately sent onwards on its path. Otherwise, it is only sent if at least k steps have elapsed since it was last sent. (This can be determined by counting steps modulo k.)

Consider viable path P of length L_P and what can happen in between the input of two successive data items. In the worst case, the path needs to be cleaned, costing $O(L_P)$ new messages and $O(L_P)$ time; then the sender has to receive two alternating bit tokens, costing $O(L_P)$ new messages and $O(L_P)$ time; and finally the new sting-tag must sting the receiver, costing $O(L_P)$ new messages and $O(L_P)$ time. A similar analysis holds for the acknowledgment to come back to the sender.

Thus the time complexity is $O(L)$, where L is the length of the shortest viable path.

We now discuss the message complexity. The number of retransmitted messages sent between two successive data item inputs is $O(n\mathcal{P}L/T)$. (Recall that T is the retransmission parameter.) The reason is that during the $O(L)$ time between the inputs, each of the n processors retransmits for each of the $\mathcal{P}$ paths $O(L/T)$ times. Now we consider the new messages. For each path P with length L_P, the discussion above shows that $O(L_P)$ new messages are used. Since L_P is at most n and there are $\mathcal{P}$ paths, the number of new messages is $O(n\mathcal{P})$. Thus the message complexity is $O(n\mathcal{P} + n\mathcal{P}L/T))$. (Note that some cost due to retransmissions is implicitly, but not explicitly, there for protocols that assume a reliable data link layer.)

5 Self-Stabilizing Protocol

The *self-stabilizing* property [13] is important when a protocol should recover following any number and any type of faults. Regardless of the initial state of the system, once the assumptions made for the normal operation of the protocol start to hold (e.g., the sender and the receiver do not crash, every corrupted message is identified) the system eventually begins to function correctly. On the event of the receiver or the sender crashing as well as the acceptance of undetected corrupted message our protocol (and other cited protocols) may stop functioning as desired and even deadlock. To avoid such a possibility we introduce in this section a self-stabilizing version of our protocol.

A self-stabilizing end-to-end protocol is presented in [10]. This protocol is designed for a *fail-stop network*, in which a crashed link never recovers. Our self-stabilizing protocol improves upon [10] by tolerating repeated link and processor crashes and recoveries.

The correctness condition desired of a self-stabilizing protocol is that every admissible execution starting in *any* configuration (not necessarily an initial configuration) must satisfy the safety and liveness properties from Section 3, with one change: there is a configuration after which the sequence of data items output by the receiver is a prefix of the sequence of data items input by the sender.

For the self-stabilizing version we assume that the number of messages in transit in a single path is bounded and known. We also assume that the number of processors in a single path is bounded and known. Our self-stabilizing version uses the following parameters.

- k_1 : upper bound on number of nodes in each path
- k_2 : upper bound on number of messages simultaneously in transit in one direction of an undirected path.
- k_3 : some constant $> 2k_1 + 2k_2$.
- k_4 : some constant $> k_3 + 1$.
- k_5 : some constant $> k_4$.

To make the previous version of the algorithm self-stabilizing, S and R need to *explicitly clean* the paths periodically. Informally, S and R explicitly clean a path by exchanging enough messages on a path so that they can be sure that there are no old leftover messages in the path. In more detail, the sender and the receiver check the paths by sending $2k_1 + 2k_2 + 1$ *control messages* as follows. We now describe how the sender makes sure that a path Path is clean (the receiver uses a similar scheme). The sender repeatedly sends the message $(Path, Cln, 1)$ till the sender receives the message $(Path, ClnAck, 1)$. Then the sender repeatedly sends $(Path, Cln, 2)$ and waits for $(Path, ClnAck, 2)$ and so on till the sender sends $(Path, Cln, 2k_1 + 2k_2 + 1)$ and receives $(Path, ClnAck, 2k_1 + 2k_2 + 1)$. At that instant the sender is through checking the path Path. Whenever the receiver receives $(Path, Cln, i)$ the receiver sends $(Path, ClnAck, i)$.

The reasoning behind this explicit clean is the fact that when S started cleaning the path there were at most $2k_1 + 2k_2$ distinct sequence numbers in Path: $2k_1$ in Pending arrays and $2k_2$ in messages. Thus, during the explicit clean S starts sending at least once a sequence number that does not exist in Path. Since the path is FIFO it holds that when S receives an acknowledgment for this sequence number the path is (explicitly) cleaned, i.e., all messages in transit have the same sequence number.

The protocol is asynchronous, thus the period of time after which S and R should start the explicit clean cannot be measured in real time. Instead we use the alternating bit protocol for every path to make sure that some data items are transferred between any two successive explicit cleans. Namely S (R) keeps a counter of alternating bit tokens arriving at S (R) through each path. When the counter of some path reaches a predefined bound then S (R) starts an explicit clean procedure to clean all the paths.

The explicit cleans of S and R should also be coordinated in a way that ensures the clean property from Section 3 of explicitly cleaned paths. Ideally, S will communicate the fact that it started the cleaning procedure to R and both S and R will clean paths and initialize their variables in coordination. Unfortunately, S cannot be sure whether a certain Path is clean or not (this is the cause for the explicit clean). Thus, S uses the fact that R starts executing its cleaning when k_4 tokens are received on some path. S waits until k_4 tokens are received by R and only then initializes its variables.

In more detail, the coordination of the explicit cleans is controlled by S. When S starts an explicit clean of all the paths S does not transmit data until R starts an explicit clean too. S verifies that R started the explicit clean procedure by discovering that the counter of tokens received by R through some explicitly clean path had been set to zero. Then S sets all of its token counters to zero. Note that following this operation it holds for each path that the counters of tokens maintains by S and R for this path differ by at most 1. We choose k_3 and k_4 as the predefined bound on the number of tokens that cause S and R, respectively, to start explicit cleaning. This choice guarantees that following the first coordination of the explicit cleaning S starts cleaning first when R is just about to start explicit cleaning too – then S causes R to enter the cleaning mode by exchanging additional tokens with R.

In more detail, S keeps an array of counters $\mathrm{ClnTkns}_{RS}[1..\mathcal{P}]$ and R keeps an array of counters $\mathrm{ClnTkns}_{SR}[1..\mathcal{P}]$; they are used to count the number of alternating bit tokens received over each path. S has three modes of operation—cleaning, inquiring and data transfer. In the cleaning mode, no data is transferred; the purpose is to initiate the explicit cleaning of the paths. Once at least one path has been explicitly cleaned, S enters the inquiring mode during which S repeatedly asks R, over every explicitly cleaned path, to send the value of $\mathrm{ClnTkns}_{SR}$ for this path. Once S observes that for some path, the $\mathrm{ClnTkns}_{SR}$ entry at R has been initialized, S sets the entries of its $\mathrm{ClnTkns}_{RS}$ array to zero, and enters the data transfer mode. In this mode data transfer can start (or resume) over the explicitly cleaned path(s); however, explicit cleaning continues in parallel on the dirty paths (i.e., paths that are not explicitly cleaned). S exits the data transfer mode and enters the cleaning mode (again) when some entry of $\mathrm{ClnTkns}_{RS}$ reaches k_3. Upon entering the cleaning mode S sets $\mathrm{ClnTkns}_{RS}$ to zero. R just has two modes of operation, cleaning and data transfer. R exits the the data transfer mode and enters the cleaning mode again when some entry of $\mathrm{ClnTkns}_{SR}$ is greater than k_4. Upon entering the cleaning mode R sets $\mathrm{ClnTkns}_{SR}$ to zero. Once at least one path has been explicitly cleaned, R enters the data transfer mode.

The correctness of the protocol hinges on S repeatedly entering the cleaning mode. By the nature of the alternating bit protocol and the existence of a viable path it is clear that both the data transfer mode and the cleaning mode terminate. The termination of the inquiring mode depends on S finding out that R starts the explicit clean, i.e., R sets ClnTkns_{SR} to zero. We show that S definitely finds out this fact through a clean path. However, in case S starts in inquiring mode "knowing" that some set of paths were explicitly cleaned S does not attempt to explicitly clean those paths. In such a case S might not discover that R starts the explicit clean since the set of "explicitly clean paths" contains only non cleaned paths that lose the information concerning R assigning zero to ClnTkns_{SR}. To eliminate such a possibility S counts tokens during the inquiring mode and when k_5 tokens are counted for some path, S starts the cleaning mode. Note that $k_5 > k_4$; thus normally (i.e., following the first time S starts the explicit cleaning) S succeeds in triggering R to enter the cleaning mode before k_5 tokens are counted by S.

When S enters the cleaning mode: S initializes CleanPaths_{SR}, Sent_{SR}, VRecv_{SR}, VSent_{RS}, Recv_{RS}, Bit_{SR} and Tkns_{RS} to the initial values presented in Figure 2. Similarly R initializes CleanPaths_{RS}, Sent_{RS}, VRecv_{RS}, VSent_{SR}, Recv_{SR} and Tkns_{SR} to the initial values upon entering the cleaning mode. Note that in order to ensure proper "synchronization" of the alternating bit protocol through the paths R does not initialize Bit_{RS}. R initializes $\text{Bit}_{RS}[\text{Path}]$ to nil whenever R receives (Path,Cln,$2k_1 + 2k_2 + 1$). We now explain why R sets Bit_{RS} to nil when it receives the last Cln message from S. When S receives the last ClnAck, all the messages in transit from S to R are either Cln or ClnAck and thus do not use the alternating bit protocol. Consequently, R doesn't change its bit. When S receives the last ClnAck, all the messages in transit from R to S either do not use the alternating bit protocol (because they are Cln or ClnAck messages) or have nil as the value of the bit. Thus they will be ignored by S, since its bit never equals nil. After receiving the last ClnAck, S starts using 0 for the alternating bit protocol; this value will be accepted by R as being new.

Eventually, while in the inquiring mode S finds that an entry in ClnTkns_{SR} has changed from a nonzero number to zero, i.e., the receiver has initialized its variables too. Then S uses the alternating bit protocol to receive the values of SeqNum_{RS} and StingTag_{SR} from R. Using those values, S decides whether a new data item should be sent (if StingTag_{RS} equals SeqNum_{RS}) or the old data item should be sent (if StingTag_{RS} does not equal SeqNum_{RS}). S assigns $\text{SeqNum}_{SR} := \text{StingTag}_{SR}$ and $\text{StingTag}_{RS} := \text{SeqNum}_{RS} - 1$ (to make sure that R is stung first), initializes ClnTkns_{RS}, and enters the data transfer mode, sending the correct data item.

The coupling between the previous algorithm and this new part is that the set of "all" paths considered in the previous version, in order to assign CleanPaths for instance, is now the set of paths that have been explicitly cleaned since the last time the explicit cleaning was initiated. S and R *ignore* any message that arrives on a dirty path except for Cln or ClnAck.

Lemma 5.1 *S eventually enters its cleaning mode.*

Proof: Assume towards contradiction that S never enters its cleaning mode. Since S only enters the inquiring mode upon leaving the cleaning mode, there exists some time after which S is forever in the same mode.

Consider first the case in which S is stuck in the cleaning mode. In this case S is stuck sending Cln messages over every path. Since there exists a viable path Path, eventually those messages reach R through Path. Whenever R receives a Cln message, it responds with a ClnAck message. Repeated application of this argument yields that eventually S finishes sending Cln messages over Path and enters the inquiring mode.

Now consider the case in which S is stuck in the inquiring mode. Let t be the latest time S enters the inquiring mode. Obviously it never enters the data transfer mode after time t. We will show that S must eventually enter the cleaning mode, because for some path Path, $\text{ClnTkns}_{RS}[\text{Path}]$ is incremented enough times. Since S tries to clean every dirty path, eventually S considers every viable path as explicitly cleaned. Once this occurs, S counts the inquiry tokens on each viable path. Suppose R never enters the cleaning mode after time t. Then R eventually consider each viable path as an explicitly cleaned path and sends back the alternating bit tokens that arrive through this path. Thus, an entry in ClnTkns_{RS} is incremented. Now suppose R does enter the cleaning mode after time t. Then R has to receive k_3 tokens from S between any two successive cleaning of R. Thus, S must send tokens (and hence receive tokens), incrementing an entry of ClnTkns_{RS}. Repeated application of this argument yields that some entry of ClnTkns_{RS} eventually reaches k_5.

The last case is when S is stuck in the data transfer mode. A similar argument to the second case shows that some entry of ClnTkns_{RS} eventually reaches k_3. ∎

Lemma 5.2 *After S enters the cleaning mode for the first time and before S leaves the inquiring mode thereafter, R enters the cleaning mode.*

Proof: By Lemma 5.1, S enters the cleaning mode. By the existence of a viable path, after S starts the explicit cleaning S eventually succeeds in sending all the Cln messages and receives ClnAck through at least one path. Then S enters the inquiring mode.

Before S leaves the inquiring mode either S finds through an explicitly cleaned path that R entered the cleaning mode (and thus we are done) or S counts k_5 tokens arriving through an explicitly cleaned path. In the latter case R answered S with $k_5 > k_3$ tokens, thus R must have entered the cleaning mode too. ∎

Theorem 5.3 *The above protocol is a self-stabilizing crash resilient end-to-end protocol.*

Proof: We prove the following facts.

1. Let t be the time of the first entrance of R to the cleaning mode following the first entrance of S to the cleaning mode. Let $I_t = i_1, i_2, \cdots$ be the sequence of data items input by S following t and let $O_t = o_1, o_2, o_3, \cdots$ be the of data items output by R following t. Then I_t is either a prefix of O_t or a prefix of $o_2, o_3, \cdots$.

2. R does an infinite number of outputs.

To prove these two facts, we need to show that after R entered the cleaning mode, as guaranteed by Lemma 5.2, our non stabilizing protocol is initialized correctly. We also have to show that the next explicit cleaning will not cause loss or duplication of the data items.

When S starts the cleaning on Path there is at least one sequence number j such that $1 \leq j \leq 2k_1 + 2k_2 + 1$ and no Cln or ClnAck message with j is in transit in Path. When S sends (Path,Cln,j) and receives (Path,ClnAck,j), all the messages in transit from S to R in Path are (Path,Cln,j) and all the ClnAck messages from R to S have j as sequence number. This ensures that when S sends any message (Path,Cln,i) with $i \geq j$, R receives this message, and then S receives (Path,ClnAck,i). Thus, when S sends (Path,Cln,$2k_1 + 2k_2 + 1$) this message arrives at R, causing R to assign $\text{Bit}_{RS} :=$ nil and then R starts to send (Path,ClnAck,$2k_1 + 2k_2 + 1$). Therefore, when S receives (Path,ClnAck,$2k_1 + 2k_1 + 1$) all the messages in transit either do not have a bit field or have nil value in their bit field. Thus, the alternating bit protocol is initialized for Path: when S sends a message with Bit$= 0$ for the first time after the explicit cleaning, S receives a message with Bit$= 0$ only after R does.

By the fact that the alternating bit protocol on each clean path of S is working correctly, S receives the correct information about Sting_{RS} and SeqNum_{SR}, and thus stings R with the correct data. This fact ensures that no data is lost or duplicated. ∎

6 Concluding Remarks

We have presented a crash resilient end-to-end protocol for dynamic networks in which links and processors can crash and recover spontaneously. The protocol ensures reliable data transfer as long as there is at least one viable path between the sender and the receiver. We have further presented a self-stabilizing version that can cope with the sender and the receiver entering arbitrary states (due to failures) and undetected corrupted messages.

The partition of a communication protocol into separate layers is a common procedure of implementation. Although our protocols bypassed the data-link layer, they will still work if a data-link protocol is executed on each link, even in the presence of crashes. A crash can affect the correct functioning of a data-link protocol in one of two ways: either a message is lost, which the link could do even without the data-link protocol, or a message is duplicated. However, this duplication is tolerable by our protocol, because it could have happened at the same point with our intermediate node protocol as well — the duplicate immediately follows what it is a duplicate of, and is not inserted later in some malicious way.

Our protocol can be easily modified to work in the case that viability in one direction does not imply viability in the other direction. Let $\mathcal{P}_{SR}$ be the number of paths from the sender to the receiver and $\mathcal{P}_{RS}$ be the number of paths from the receiver to the sender. One possibility is to virtually implement $\mathcal{P}_{SR} \times \mathcal{P}_{RS}$ alternating bit protocols, one for each possible combination of sender-receiver path with receiver-sender path. Since there is at least one combination that is viable in both directions, current information about the Recv_{SR} array will reach the sender and current information about the Recv_{RS} array will reach the receiver.

Several optimizations are possible. For example, our protocol repeatedly sends a single message and waits for acknowledgment. Instead we can send several different messages with the same sequence number of the single message sent, each message is augmented with a (bounded) running sequence number too. The acknowledgment will include the last running sequence number received in a similar fashion to the sliding window protocol. For simplicity of presentation we have not incorporated the above optimizations.

Acknowledgment: We thank Reuven Cohen and Amir Herzberg for helpful discussions. Amir also gave us useful comments on a written draft.

References

[1] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck. Reliable Communication Over Unreliable Channels. Technical Report YALEU/DCS/TR-853, Department of Computer Science, Yale University, October 1992.

[2] Y. Afek, B. Awerbuch, and E. Gafni. Applying Static Network Protocols to Dynamic Networks. In *Proc. of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pp. 358–370, 1987.

[3] H. Attiya, S. Dolev, and J. Welch. Connection Management Without Retaining Information. In *Proc. of the 28th Hawaii International Conference on System Sciences*, pp. 622–631, 1995.

[4] B. Awerbuch and S. Even. Reliable Broadcast Protocols in Unreliable Networks. *Networks*, 16:381–396, 1986.

[5] Y. Afek and E. Gafni. End-to-End Communication in Unreliable Networks. In *Proc. of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pp. 131–148, 1988.

[6] Y. Afek and E. Gafni. Bootstrap Network Resynchronization: An Efficient Technique for End-to-End Communication. In *Proc. of the 10th Annual ACM Symposium on Principles of Distributed Computing,* pp. 295–307, 1991.

[7] B. Awerbuch, O. Goldreich, and A. Herzberg. A Quantitative Approach to Dynamic Networks. In *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pp. 189–203, 1990.

[8] Y. Afek, E. Gafni and A. Rosen. The Slide Mechanism with Applications in Dynamic Networks. In *Proc. of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pp. 35–46, 1992.

[9] B. Awerbuch, Y. Mansour, and N. Shavit. Polynomial End to End Communication. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pp. 358–363, 1989.

[10] B. Awerbuch, B. Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. of the 32nd Annual IEEE Symp. on Foundations of Computer Science*, pp. 268-277, 1991.

[11] B. Awerbuch and M. Sipser. Dynamic Networks are as Fast as Static Networks. In *Proc. of the 29th Annual IEEE Symposium on Foundations of Computer Science,* pp. 206–220, 1988.

[12] K. Bartlett, R. Scantlebury, and P. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM,* 12(5):260–261, May 1969.

[13] E.W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control, *Communications of the ACM,* 17(11):643–644, 1974.

[14] S. Dolev, and J. Welch. Crash Resilient Communication in Dynamic Networks, Technical Report 93-032, Department of Computer Science, Texas A&M University, June 1993.

[15] S. G. Finn. Resynch Procedures and a Fail-Safe Network Protocol. *IEEE Trans. on Communication*, COM-27:840–845, June 1979.

[16] A. Fekete, N. Lynch, Y. Mansour and J. Spinelli, The Impossibility of Implementing Reliable Communication in the Face of Crashes, to appear in *Journal of the ACM.* Also: Technical Memo MIT/LCS/355.c, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1991.

[17] A. Herzberg. Connection-Based Communication in Dynamic Networks. In *Proc. of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pp. 13–24, 1992.

[18] D. Wang and L. Zuck, Tight Bounds for the Sequence Transmission Problem. *Proc. 8th ACM Symposium on Principles of Distributed Computing,* pp. 73–83, 1989.

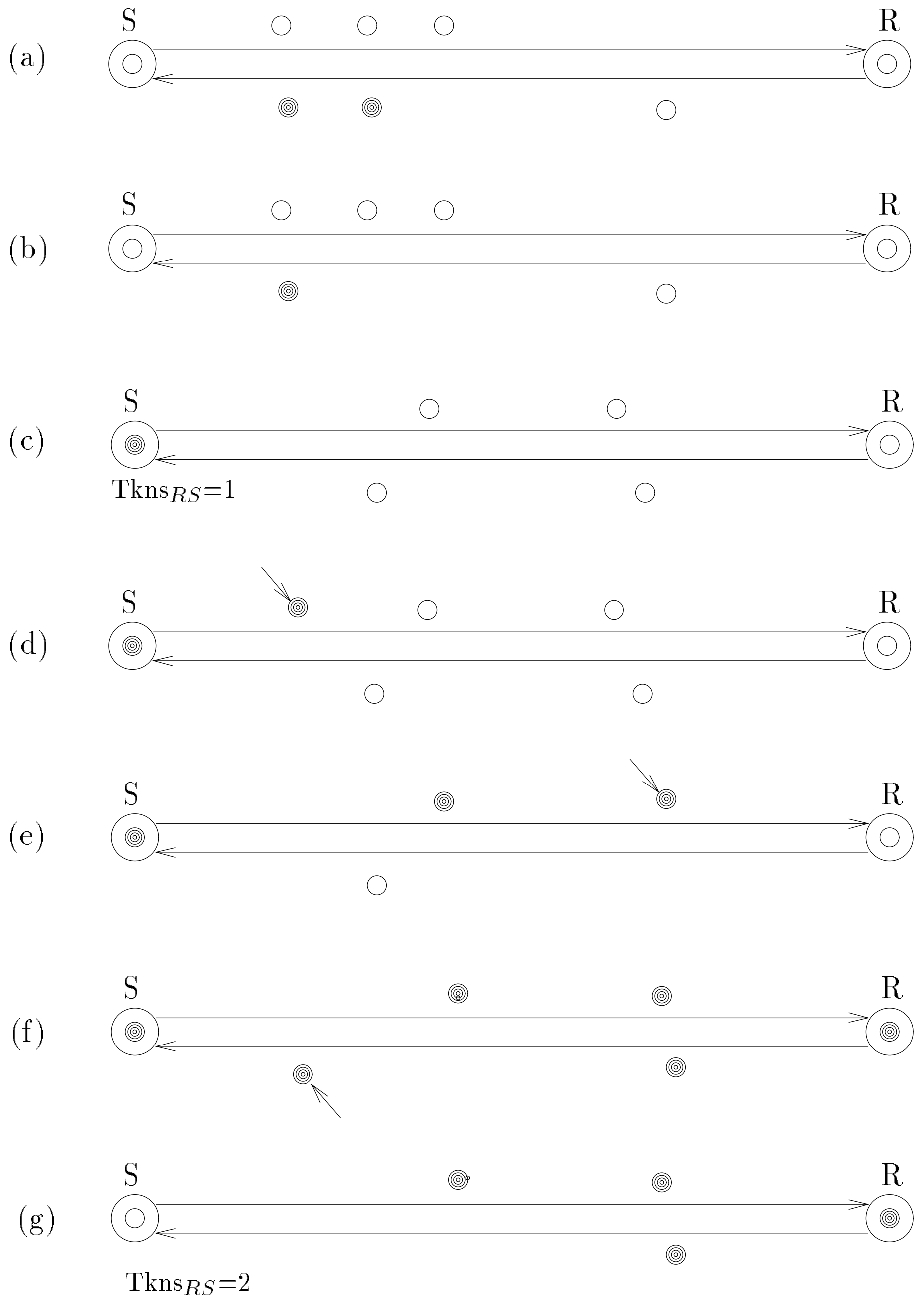


Figure 8: The second token of the alternating bit protocol