LOG-SPACE POLYNOMIAL END-TO-END COMMUNICATION*

EYAL KUSHILEVITZ†, RAFAIL OSTROVSKY‡, AND ADI ROSÉN§

Abstract. Communication between processors is the essence of distributed computing: clearly, without communication distributed computation is impossible. However, as networks become larger and larger, the frequency of link failures increases. The end-to-end communication problem asks how to efficiently carry out fault-free communication between two processors over a network, in spite of such *frequent* link failures. The sole minimum assumption is that the two processors that are trying to communicate are not permanently disconnected (i.e., the communication should proceed even in the case that there does not (ever) simultaneously exist an operational path between the two processors that are trying to communicate).

We present a protocol to solve the end-to-end problem with logarithmic-space and polynomial-communication at the same time. This is an exponential memory improvement to all previous polynomial-communication solutions. That is, all previous polynomial-communication solutions needed at least *linear* (in n, the size of the network) amount of memory per link.

Our protocol transfers packets over the network, maintains a simple-to-compute $O(\log n)$-bits potential function at each link in order to perform routing, and uses a novel technique of packet canceling which allows us to keep only *one* packet per link. The computations of both our potential function and our packet-canceling policy are totally local in nature.

1. Introduction. In this paper we address the problem of *communication* in distributed systems: How can two processors (a sender and a receiver) communicate over an unreliable communication network? This question was considered in many different settings, which can be divided into two groups depending on the *frequency* of link failures.

Communication During Infrequent Faults. If link failures are "infrequent", then after each failure a new communication path between the two communicating processors can be computed, which will be operational until the next fault. That is, in case failures occur *rarely*, it is possible, upon a failure, to "reset" the network and compute a new path between the communicating processors (e.g., [Fin79, AAG87, AS88, AAM89, AGH90]). In a related model, a communication path can also be computed in a "self-stabilizing" manner (e.g., [Dij74, AKY90, KP90, APV91, AV91, AKM+93, AO94, IL94]), which essentially means that after faults stop *for a sufficiently long period of time*, the protocol "stabilizes" to its correct behavior (i.e., establishes a new path). We emphasize that both the above reset and self-stabilizing solutions work *only if faults*

* An early version of this paper appeared in *Proc. of the 27th ACM Symp. on the Theory of Computing (STOC)*, pp. 559–568, May 1995.

† Dept. of Computer Science, Technion, Haifa 32000, Israel. E-mail: eyalk@cs.technion.ac.il. http://www.cs.technion.ac.il/~eyalk . Part of this research was done while visiting ICSI, Berkeley.

‡ Computer Science Division, University of California at Berkeley, and International Computer Science Institute, Berkeley, CA 94720. E-mail: rafail@cs.berkeley.edu.

§ Dept. of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel. E-mail: adiro@math.tau.ac.il. Part of this research was done while visiting ICSI, Berkeley. Current address: Dept. of Computer Science, University of Toronto, Canada.

are not too frequent, as they require that a message is transmitted over the computed path.

Communication During Frequent Faults. We consider a setting where failures occur frequently. The so-called end-to-end communication problem [AG88, AMS89, AGR92, AAG+97] is to deliver, in finite time, data-items from a sender processor to a receiver processor, where data-items are given in an on-line fashion to the sender, and must be output in the same order, without duplication or omission at the receiver processor. This should be done even if there does not exist, at any time, a path of simultaneously active links that connects the two communicating processors. The sole assumption is that the two communicating processors are not separated by a cut of permanently failed links[1]. Solutions to the problem are evaluated according to their *Communication Complexity* and *Space Complexity*.

1.1. Frequent faults model.

Communication Complexity. One possible solution to the above problem is to give data-items "unbounded sequence numbers" and to "flood" the network with each item [Vis83, AE86]. However, this solution has the drawback that the message size increases unboundedly with the number of items sent; hence, the amount of communication needed per data-item grows unboundedly with the number of data items. Recently, the study of end-to-end protocols with *bounded* communication complexity received a lot of attention [AG88, AMS89, APV91, AG91, AGR92, AAG+97]. In this paper, we concentrate on bounded (in fact, polynomial) communication-complexity protocols.

Space Complexity. Another important complexity measure is the *space complexity* — the amount of space needed at the processors, per incident link. Notice that the "unbounded sequence numbers" solution requires unbounded memory as well. The question of reducing memory requirements, while maintaining efficiency, received a lot of attention in the "self-stabilizing" setting (e.g., [MOOY92, AO94, IL94]), where it was shown that, in the case of *infrequent* memory faults, small memory and communication efficiency are simultaneously attainable. In contrast, all end-to-end protocols that were efficient in terms of their communication complexity, were not efficient in their space complexity: more precisely, all end-to-end communication protocols that had polynomial communication complexity, required at least *linear* (in the number of processors of the network) amount of space, at each processor, per incident link. Protocols with smaller space complexity were only presented at the cost of (at least) exponential communication complexity: Afek and Gafni [AG88] give a protocol which uses logarithmic amount of space, but has exponential communication complexity, and another protocol which uses constant amount of space but has unbounded communication complexity.

1.2. Our Result. The question whether there exists an end-to-end communication protocol with sub-linear space complexity and at the same time polynomial commu-

[1] See Section 2 for a formal description of the model.

nication complexity remained open. In this paper, we give an affirmative answer to this question: we give a protocol that has logarithmic ($O(\log n + D)$) space complexity and polynomial ($O(n^2 mD)$) communication complexity, where n and m are the number of processors and links in the network respectively, and D is the data-item size. This is an exponential space complexity improvement over all known polynomial-communication protocols. Notice that this is achieved at a slight increase of the communication complexity comparing to the best known solution [AG91]. We compare our result to the previous work in the table below:

Paper reference	Communication Complexity (total number of bits)	Space Complexity (bits per incident link)
[Vis83, AE86]	**unbounded:** ∞	**unbounded:** ∞
[AG88], Alg. 1.	**unbounded:** ∞	**constant:** $O(D)$
[AG88], Alg. 2.	**exponential:** $O(D \cdot \text{exponential}(n))$	**logarithmic:** $O(\log n + D)$
[AMS89]	**polynomial:** $O(n^9 + mD)$	**polynomial:** $O(n^5 + D)$
[AGR92]	**polynomial:** $O(n^2 mD)$	**linear** : $O(nD)$
[AG91]	**polynomial:** $O(nm \log n + mD)$	**linear:** $O(n + D)$
present work	**polynomial:** $O(n^2 mD)$	**logarithmic:** $O(\log n + D)$

Our Techniques and Previous Work. The starting point of our investigation is the Slide protocol of [AGR92][2] and the Majority algorithm of [AAF+90, AGR92]. The Slide protocol requires storing n packets per incident link, and the decision on the recency of the received item is taken only at the receiver processor (using the technique of [AAF+90]). If we wish to reduce the space per link, we can no longer afford storing n different packets, but must keep far fewer packets per link, and we can no longer afford the Majority algorithm of [AAF+90, AGR92], which collects a large number of packets arriving at the receiver and then calculates the value representing the majority among the received values. Thus, we must somehow "drop" all but several packets, and decide which to keep at each of the processors. Moreover, we must design an "on-line" analogue of the majority calculation, where we can "discard" packets as soon as they arrive at the receiver processor, yet, manage to compute the majority value. However, if we begin to "drop" packets everywhere, is it no longer the case that the technique of deciding which packet is the correct one to output (at the receiver), still works. To overcome both difficulties, we combine two ingredients:

[2] A journal version of this paper, jointly with [AMS89], appears in [AAG+97].

- A potential function that controls the flow of data-items in the network, which is an analogue to the potential function of the Slide protocol; and
- A novel data-item cancelling policy which makes sure that in any processor there will be at most two distinct values of data-items per link. The same policy is used both in the intermediate processors and in the receiver processor.

We show that a combination of these two techniques yields the desired result. Hence, even in the presence of frequent link failures, it is possible to achieve *both* polynomial communication and logarithmic space. As in [AMS89, AGR92], our solution has the additional benefit that it is totally local in nature. For example, the locality of Slide was used for establishing its self-stabilizing extension in [APV91]; similar local techniques were also used for various multi-commodity flow problems in dynamic graphs [AL93, AL94].

1.3. Organization. Section 2 contains all the necessary background including the formal definitions of the model and the problem. Section 3 contains the description of the protocol; we start with an informal description (Section 3.1), and then give a formal description (Sections 3.2 and 3.3). Then, we show some of the properties of the protocol (Section 3.4 with some proofs deferred to Appendix A). We conclude with its proof of correctness and complexity (Section 4).

2. Model and Problem Statement.

2.1. The Network Model. A *communication network* is associated with an undirected graph $G(V, E)$, $|V| = n$, $|E| = m$, where nodes correspond to processors and edges correspond to links of communication. We denote by $E(v)$, for $v \in V$, the set of edges which are adjacent to v. Processors are modeled as identical (except the sender and the receiver) deterministic finite state transducers with $O(\log n)$ memory per edge (We do not require processors to have unique identifiers). We model each *undirected* communication link as consisting of two *directed* links, delivering messages in opposite directions. Each transmission of a message is associated with a *send* event and a *receive* event; each event has its time of occurrence according to a global time, unknown to the processors. Without loss of generality, we assume that no two events occur exactly at the same time. A message is said to be *in transit* in any time after its send event and before its receive event. As discussed in the introduction we would like to deal with networks whose links may fail and recover frequently. Each failure and recovery is eventually reported to both end-points of the link. When a link fails, messages that are in transit over it are lost. In the following we give a formal definition of a somewhat simpler-to-discuss model that we use in the sequel. Intuitively, the main difference between the models is that in the model we define below links never recover if they fail, and that such a failure is *not* reported to the processors. It is known that this simpler model, which was also used in [AG88, AMS89, AGR92], does not cause any reduction in power (for completeness we prove this fact in Section 2.4). In our model each *directed* link satisfies the following properties:

- Each link has *constant* capacity; that is, the protocol must obey the rule that only a constant number of messages are in transit on a given link at any given time.
- Communication over links obeys the *FIFO* rule; that is, at any time, the sequence of messages *received* over the link is a prefix of the sequence of messages sent over the link.
- Communication is *asynchronous*; that is, there is no a-priori bound on message transmission delays over the links.

A directed link is called *non-viable* if starting from some message and on it does not deliver any message; the transmission delay of this message and any subsequent message sent on this link is considered to be infinite. The sequence of messages received over the link is in this case a *proper* prefix of the sequence of messages sent. Otherwise, the link is *viable*. An undirected link is *viable* if both directed links that it consists of are viable.

We say that the sender is *eventually connected* to the receiver if there exists a (simple) path from the sender to the receiver consisting entirely of viable (undirected) links. Note that if there is a cut of the network, disconnecting the sender from the receiver, such that all the directed links crossing the cut are non-viable links, then eventually it becomes impossible to deliver messages from the sender to the receiver.

Remark. Notice that we model an undirected graph as a bi-connected directed graph. Hence we assume that either both directed links are viable or both are not viable. In this case, the above assumption about the eventual connectivity of the sender and the receiver is in fact the minimal possible for communication between them. On the other hand, in the model of *directed* graphs, it could be the case that there is a *directed* path from the sender to the receiver, and maybe (another) directed path from receiver to sender, yet, all undirected edges are non-viable. We do not consider such (more difficult) case, and are in fact dealing only with undirected graphs.

2.2. The End-to-End Communication Problem. The purpose of an end-to-end communication protocol is to establish a (directed) "virtual link" to be used for the delivery of data-items from one special processor, called the *sender*, to another special processor, called the *receiver*. Each data-item is a character of some alphabet of size 2^D; that is, each data-item can be encoded by D bits. It is required that if the sender is eventually connected to the receiver then the "virtual link" established by the protocol will be viable. This virtual link should have the same properties as a "regular" network link; namely, it should satisfy:

Safety: The sequence of data-items output by the receiver, at any time, is a *prefix* of the sequence of data-items input by the sender.

Liveness: If the sender is eventually connected to the receiver, then each data-item input by the sender is eventually output by the receiver.

A protocol for the end-to-end communication problem is given, *in an on-line fashion*, a sequence of data-items at the sender (i.e., every data-item must be delivered

without the knowledge of the next data-item to be transmitted) and generates a sequence of data-items at the receiver, that obey the safety and liveness properties.

2.3. The Complexity Measures. We consider the following complexity measures:

Communication: The number of bits transferred in the network in the worst case, per data-item delivered. More precisely, the total number of bits sent in the worst case in the period of time between two successive data-item output events at the receiver (measured in terms of n, m and D).

Space: The maximum amount of space per incident link required by a processor's program throughout the protocol (measured in terms of n, m and D).

In addition, we require that the local computation of the processors be polynomial for each send/receive event at each processor (in fact, our protocol uses a constant number of computational steps per event).

2.4. Relations to other Models. The model described above is called the "∞-delay model" in [AG88], and the "fail-stop model" in [AM88]. As mentioned, we would like to deal with networks where links fail and recover frequently; in such *dynamic* networks, links may fail and recover many times (yet processors never fail) (see [AAG87]), and each failure or recovery of a network link is eventually reported at both its endpoints by some underlying link protocol. This model should be contrasted with the self-stabilizing model (e.g., [Dij74]), where both processors and links can start in an inconsistent state, but it is assumed that they never fail after the computation begins. As was discussed in the introduction, the self-stabilizing model corresponds to *infrequent memory faults*, and is incomparable to the model of *frequent link failures* addressed here. The question how, in addition to frequent failures of links, one can allow inconsistent initial memory states was addressed in the end-to-end setting in [APV91, DW93].

As pointed out in [AG88], one can design protocols in the *fail-stop* model and convert them to the dynamic model. For this, a message to be forwarded on a link is stored in a buffer until the link recovers and the previously sent message has been delivered. A protocol similar to the data-link initialization protocol of [BS88] is used to guarantee that no message is lost or duplicated. Each link in the dynamic network, that fails and never recovers for a long enough period to allow the delivery of a message, is represented by a non-viable link. Note that the only space used by the above transformation is for storing the buffers (i.e., per each link it is the capacity of the link, times the size of the longest message)[3].

3. The Protocol.

[3] The communication is increased by a multiplicative factor of the number of failures of the link.

3.1. High-level Description. Our starting point is a linear space, yet simple, solution of [AGR92]. Their solution combines (as black-boxes) two components: the Slide protocol and the Majority algorithm. Before explaining our protocol, we give a short overview of the approach taken in [AGR92]. The Slide is used to transfer tokens (packets containing a data-item) in the network. This is done by letting each processor maintain a stack of tokens for each of its links. On each link, if active, tokens move from a larger stack of tokens to a smaller stack of tokens. The sender always has a large stack of tokens so it only sends tokens out, and the receiver always has an empty stack hence it only receives tokens. The Majority algorithm enables the receiver to decide, by collecting a sufficiently large number of tokens (containing data-items) and taking the majority value, what is the sequence of data-items sent by the sender. It is proved in [AGR92] that the combination of these two components yields a polynomial-communication solution to the end-to-end problem.

In order to make the space requirements of the protocol logarithmic per edge, we can no longer use the Slide as a method to establish the virtual link between the sender and the receiver and we can no longer use the Majority algorithm. The reason is that the Slide needs a lot of space to store the stack of tokens (for each link), and the Majority algorithm needs even more space to collect the tokens in order to decide on the correct value to output. To overcome this, we introduce the following procedure:

- While transferring packets from the sender to the receiver, our protocol "cancels" some of these packets *both* en route and upon their arrival at the receiver processor. More precisely, it replaces some packets by "nil" packets.
- The cancelling policy is designed so as to guarantee that at most two different packet-values are to be stored at each processor (including the receiver) at any given time – a "real" packet, and a "nil" packet. We use a potential function that only *counts* the number of packets of both types, rather than storing all of them, and use this function to control the flow of tokens. We prove that two counters (of $\lceil \log n \rceil$ bits each) per link are sufficient for this. Altogether, we use only two counters per link and store only one data-item per processor.

Note that, usually, protocols that change the values of packets are undesirable. However, our protocol changes these values in a very restricted way – it may only replace a "real" packet by a "nil" packet. Our "canceling" policy does not effect the routing properties of the Slide protocol: it guarantees that if the sender and the receiver are eventually connected, then tokens will be transferred from the sender to the receiver. Moreover, our protocol guarantees an upper bound on the number of tokens that are in the network at any given time. Denote this upper bound by $\mathcal{C}$. For each data-item to be sent, the sender transmits to the receiver, $2\mathcal{C}+1$ packets (tokens) that contain that data-item. The receiver *in an on-line, space-efficient fashion*, "collects" the same number (i.e., $2\mathcal{C}+1$) of tokens (some of which, but no more than $\mathcal{C}$, may be old tokens remained in the network from previous transmissions), and outputs the data-item that represents the majority amongst the tokens received, *ignoring the nil tokens*[4]. We emphasize that

[4] for the first data-item only $\mathcal{C}+1$ tokens are collected.

the receiver computes this majority *without storing the tokens*; rather, it does so by using the same "canceling policy" as in the intermediate processors.

As it is clear from the above description, the heart of our protocol is the new (local) "cancelling policy", described below. Whenever a token arrives into a processor on a given edge e we do the following:

- If it is a nil token, then the counter of nil tokens of edge e is augmented by one.
- If it is a "real" data token and the data-item is identical to the data-item currently stored in the processor, then the data tokens counter of edge e is augmented by one.
- If it is a "real" data token and the data-item is different from the data-item currently stored in the processor, then the arriving token, and one data token already accounted for in the processor are both "cancelled" and become two nil tokens (that is, the counter of nil tokens of edge e is augmented by one, and for some edge e', whose data tokens counter is greater than 0, the counter of nil tokens is augmented by one, and the counter of data tokens is decreased by one). If as a result there are no more "real" data tokens accounted for in the processor (over all edges), we erase the current data-item stored in the processor.
- If it is a "real" data token and there is no data-item currently stored in the processor, we store the arriving data-item as the current one, and set the counter of data tokens of e to 1.

The essential idea of the above "canceling" policy is that from the point of view of the majority calculation done by the receiver, the above cancelation of two data-items into nil tokens has only a minor effect; the receiver only needs to ignore the nil tokens. Intuitively, since one of the properties of the old Majority algorithm is that, without these cancelations, the "correct" data-item would have more than half the tokens in any block of $2\mathcal{C}+1$ tokens then, in worst, if any of the cancelled data-items is the correct one then the other canceled data-item is an old one. Therefore, the majority of the correct data-item is maintained, while storing only one data-item per processor, and two counters per link.

3.2. A Formal Description of the Protocol. We begin by describing the data-structures and messages used by our protocol. The protocol uses the following types of messages:

TOKEN **messages:** to carry data-items (either "real" data-items or the "nil" data-item).

TOKEN_LEFT **messages:** to announce over a link e that a token, accounted for in the counters of link e, has been sent away.

ACK_TOKEN **and** ACK_TOKEN_LEFT **messages:** are used to acknowledge the arrival of a TOKEN message or a TOKEN_LEFT message, respectively.

The following data is stored at each processor v:

- A variable `current_message` that stores a single data-item to be duplicated and sent in TOKEN messages.
- For each incident link e, there are two counters `message_tokens[e]` and `nil_tokens[e]`. The variable `message_tokens[e]` records a value associated with the number of tokens that arrived on e carrying the value `current_message`; however, the value stored is not exactly this number as sometimes a stored token is converted from a message token into a nil token, at which event the value of `message_tokens[e]` is decreased by one and the value of `nil_tokens[e]` is increased by one.
- For each incident link e, a variable `bound[e]` that stores an estimate on the sum of the above counters on the other side of the link. This bound is initialized to 1, incremented by one every time a token is sent over the outgoing link, and decremented by one every time a TOKEN_LEFT message is received over the corresponding incoming link.
- For each incident link $e = (v, u)$, a counter called `tokens_left_pending[e]`, which counts the number of TOKEN messages that were sent from v (possibly on other links) on "the account" of e (i.e., caused the counters of e to decrease), but have not yet been reported as such to u. This counter is initialized as 0, incremented by one when a token leaves the counters of e, and decremented by one when a TOKEN_LEFT message is sent over e.
- For each incident link e, a flag `free_for_token` indicating if an ACK_TOKEN message has already been received for the previous TOKEN message sent on e, and a flag `free_for_token_left`, indicating if an ACK_TOKEN_LEFT message has already been received for the previous TOKEN_LEFT message sent on e.

The sender and the receiver store some additional data. The sender stores the last message it has received to transmit (`current_input`), together with a count of how many such packets are still to be sent (`left_to_send`). In addition, it stores part of the above stated data relative to an additional "virtual link" through which it introduces new tokens into the network. Although we call it a virtual *link*, in what follows it is not included in the set E. The receiver maintains a counter storing the number of packets it received since the time of the last output event (`count`), and another counter storing the number of packets that contained the message stored in `current_message` (`current_message_count`). It also maintains a single bit flag `first_item`.

Throughout the proofs we assume a global time, unknown to the processors, and we denote the value of variables in a processor at a given time by a subscript of the processor and a superscript of the time (e.g., $\mathcal{X}_v^t$). We also use the following notation to count the number of different messages on a given link at a given time: Let $tokens^t_{u\to v}$ be the number of TOKEN messages in transit from u to v at time t. Let $tokens_left^t_{u\to v}$ be the number of TOKEN_LEFT messages in transit from u to v at time t.

In our protocol the nodes do not store tokens, but merely store a single message-value and several counters. However, for our proofs we sometime use the terminology that tokens are stored, or present, in the nodes. The analogy is straightforward: when a counter counts k nil-tokens, we sometimes refer to it as if the node stores k nil-tokens,

etc. This enables us to talk about the "number of tokens in the network", rather than about the "sum of values of the counters in the network", and make our arguments more readable.

3.3. The Code. In this section, we present the code of our protocol. Following [AMS89, AGR92], the presentation of the code is based on the language of *guarded commands* of Dijkstra [DF88] where the code of each process is of the form

Select $G_1 \rightarrow A_1 \,\square\, G_2 \rightarrow A_2 \,\square\, \ldots \,\square\, G_l \rightarrow A_l$ **End Select**.

The code is executed by repeatedly selecting an arbitrary i from all guards G_i which are true, and executing A_i. Each guard G_i is a conjunction of predicates.

The predicate **Receive** M is true when a message M is available to be received. If the statements associated with this predicate are executed, then prior to this execution the message M is actually received. The message may contain some values that are assigned, upon its receipt, to variables stated in the **Receive** predicate (e.g., **Receive** TOKEN(data)). The command **Send** M on e sends the message M on the link e.

We present the code of a regular processor (Figure 1), which is every processor except the sender and the receiver. The code of the receiver is presented separately (Figure 3). The sender has an additional "virtual link" (which does not belong to the set E) through which new tokens are introduced into the network. This virtual link is denoted in the code by I. The code of the sender is composed of two parts: the code of a regular processor, and additional special code (Figure 2). Upon initialization, the sender executes the two initialization procedures (of a regular processor, and the additional one), and then regards the two select commands as a unified one.

3.4. Properties of the Protocol. In this section we present properties of the protocol which later allow us to prove its correctness and complexity. The structure of the proof follows the structure of the proofs in [AGR92], and some of the proofs are analogous to those of [AMS89, AGR92]. We first state the following technical lemmas (we postpone their proofs to Appendix A).

Consider an edge $e = (u, v) \in E$. The following lemma relates the estimate $bound[e]_u^t$ which u has on the number of tokens that are stored in v (in the counters corresponding to the same edge e). It shows that this estimate essentially equals to the actual number of tokens stored (i.e., $message_tokens[e]_v^t + nil_tokens[e]_v^t$) plus those that have left, but were not yet reported as such, and those which are still in transit (i.e., $tokens_{u\rightarrow v}^t + tokens_left_{v\rightarrow u}^t$).

LEMMA 3.1. *At any time t, and for any $e = (u, v) \in E$,*

$$\begin{aligned} bound[e]_u^t - 1 &= message_tokens[e]_v^t + nil_tokens[e]_v^t \\ &+ tokens_left_pending[e]_v^t + tokens_{u\rightarrow v}^t + tokens_left_{v\rightarrow u}^t \ . \end{aligned}$$

```
Select
Initialization ⟶
    C := n · (2m + 1) ;
    current_message=empty;
    for every incident link e ∈ E(v)
        bound[e]:=1; message_tokens[e]:=0; nil_tokens[e]:=0; tokens_left_pending[e]:=0;
        free_for_token[e]:=true; free_for_token_left[e]:=true;
□
Receive TOKEN_LEFT on e ⟶
    bound[e]:=bound[e]-1;
    Send ACK_TOKEN_LEFT on e;
□
Receive TOKEN(data) on e ⟶
    if (data=nil) then
        nil_tokens[e]:=nil_tokens[e]+1;
    else
        if (current_message=empty or current_message=data) then
            message_tokens[e]:=message_tokens[e]+1;
            current_message:=data;
        else
            nil_tokens[e]:=nil_tokens[e]+1;
            for some e' ∈ E(V) s.t. message_tokens[e'] >0
                nil_tokens[e']:=nil_tokens[e']+1;
                message_tokens[e']:=message_tokens[e'] -1;
            if (for every e message_tokens[e]=0) then current_message:=empty;
        endif
    endif
    Send ACK_TOKEN on e;
□
Receive ACK_TOKEN on e ⟶
    free_for_token[e]:=true;
□
Receive ACK_TOKEN_LEFT on e ⟶
    free_for_token_left[e]:=true;
□
∃e, e' ∈ E(v) s.t. nil_tokens[e']+message_tokens[e'] > bound[e] and
                   free_for_token[e]=true ⟶      /* possibly e' = e */
    if (nil_tokens[e'] > 0) then
        Send TOKEN(nil) on e
        nil_tokens[e']:=nil_tokens[e'] - 1;
    else
        Send TOKEN (current_message) on e;
        message_tokens[e']:=message_tokens[e'] - 1;
    endif
    free_for_token[e]:=false;
    bound[e]:=bound[e]+1;
    tokens_left_pending[e']:= tokens_left_pending[e'] + 1;
□
∃e ∈ E(v) s.t. tokens_left_pending[e] > 0 and free_for_token_left[e]=true ⟶
    send TOKEN_LEFT on e;
    free_for_token_left[e]:=false;
    tokens_left_pending[e]:=tokens_left_pending[e]- 1;
End Select
```

FIG. 1. *Code of ordinary processor v*

Select
Initialization $\longrightarrow$
    $\mathcal{C} := n \cdot (2m+1)$;
    current_input=`nil`;
    left_to_send:=0;
    message_tokens[I]:=0;
    nil_tokens[I]:=0;
    tokens_left_pending[I]:=0;
$\square$
left_to_send = 0 $\longrightarrow$
    current_input := `input data-item`;
    left_to_send:= $2\mathcal{C}+1$;
$\square$
message_tokens[I] < n **and** left_to_send > 0 $\longrightarrow$
    message_tokens[I]:=message_tokens[I] +1;
    left_to_send:=left_to_send - 1;
$\square$
$\exists e \in E(v)$ s.t. message_tokens[I] > bound[e] **and** free_for_token[e]=`true` $\longrightarrow$
    Send TOKEN (current_input) on e;
    message_tokens[I]:=message_tokens[I] - 1;
    free_for_token[e]:=`false`;
    bound[e]:=bound[e]+1;
End Select

FIG. 2. *Additional code for the sender*

The next lemma gives the main intuition for the progress in the protocol.

LEMMA 3.2. *Consider a* TOKEN *message sent from processor u to processor v, on link $e=(u,v)$. Let e' be the link whose counter at processor u decreased when the message was sent. If just before the message is sent $message_tokens[e']_u + nil_tokens[e']_u = i$ and just after it is received $message_tokens[e]_v + nil_tokens[e]_v = j$, then $j < i$.*

Since all the tokens in the network are either "stored" in the processors or in transit over links, the following lemma and corollary will allow us to give a bound on the capacity of the network (property ($\mathcal{P}1$) below).

LEMMA 3.3. *At any time t and for any $e=(u,v)$,*

$$bound[e]_u^t \leq n \ .$$

COROLLARY 3.4. *The following hold for any time t,*

1. *For any $e=(u,v) \in E$,*

$$message_tokens[e]_v^t + nil_tokens[e]_v^t + tokens_{u \to v}^t \leq n \ ,$$

$$tokens_left_pending[e]_v^t \leq n \ .$$

2. *for $e = I$,*

$$message_tokens[e]_v^t + nil_tokens[e]_v^t \leq n \ ,$$

```
Procedure check_and_output
    if first_item and count=C + 1 then /* first data-item */
        output(current_message);
        current_message := empty;
        count:= 0;
        current_message_count:= 0;
        first_item:=false;
    else
        if (not first_item) and count= 2·C + 1 then /* all other data-items */
            output(current_message);
            current_message := empty;
            count := 0;
            current_message_count:= 0;
        endif
    endif
End Procedure
Initialize ⟶
    C := n · (2m + 1) ;
    current_message := empty;
    current_message_count:=0;
    count:=0;
    first_item:=true;
    for every incident link e ∈ E(u)
        bound[e]:=0 ;
        message_tokens[e]:=0;
        nil_tokens[e]:=0;
        tokens_left_pending[e]:=0;
        free_for_token_left[e]:=true;
□
Receive TOKEN(data) on e ⟶
    count:=count + 1;
    if (data ≠ nil) then
        if (current_message=empty or current_message=data) then
            current_message_count:=current_message_count+1;
            current_message:=data;
        else
            current_message_count:=current_message_count -1;
            if (current_message_count=0) then current_message:=empty;
        endif
    endif
    tokens_left_pending[e]:= tokens_left_pending[e] + 1;
    Send ACK_TOKEN on e;
    call check_and_output;
□
Receive ACK_TOKEN_LEFT on e ⟶
    free_for_token_left[e]:=true;
□
∃e ∈ E(u) s.t. tokens_left_pending[e] > 0 and free_for_token_left[e]=true ⟶
    send TOKEN_LEFT on e;
    free_for_token_left[e]:=false;
    tokens_left_pending[e]:=tokens_left_pending[e]- 1;
End Select
```

FIG. 3. *Code of the receiver u*

$$tokens_left_pending[e]_v^t \leq n \ .$$

We use the above lemmas to prove the following theorem:

THEOREM 3.5. *The protocol has the following properties:*

- $\mathcal{P}1$. *At any time t, the number of tokens in the network is bounded by $\mathcal{C} = n \cdot (2m+1)$.*
- $\mathcal{P}2$. *Consider a time interval in which k_{new} new tokens are inserted into the network. During this time interval at most $O(n^2 m + k_{\text{new}} \cdot n)$* TOKEN *messages are sent in the network.*
- $\mathcal{P}3$. *If the sender and the receiver are eventually connected, the sender will eventually insert a new token into the network.*

Proof. Every token in the network is either in transit on a link, or accounted for in some counter at a processor. For every edge $e = (u, v) \in E$ we consider its two directions. For the direction from u to v (v to u) Corollary 3.4 states that at any given time the number of tokens in transit from u to v (v to u) plus the number of tokens accounted for at the counters of e at node v (node u) is at most n. Thus we get at most $n \cdot 2m$ tokens. At most n additional tokens can be accounted for at the counters of the "virtual link" I, altogether $n \cdot (2m + 1)$ tokens. This concludes the proof of property ($\mathcal{P}1$).

We can now also prove property ($\mathcal{P}2$). Define the following potential function. For any processor v and for any incident link $e = (v, u)$, and for $e = I$ if $v = s$, denote $J^t(v, e) = message_tokens[e]_v^t + nil_tokens[e]_v^t$ and let $H^t(v, e) = \sum_{k=1}^{J^t(v,e)} k = \binom{J^t(v,e)}{2}$. Let P^t be the set of TOKEN messages in transit at time t. For $p \in P^t$ let t' be the time just before the TOKEN message p was sent, say from v to u. If e' is the link whose counters accounted for the token sent, then we define $T^t(p) = message_tokens[e']_v^{t'} + nil_tokens[e']_v^{t'}$ (that is, the token "carries" the "number" of tokens in v, at link e', just before it left this processor). Denote the sender by S. The potential function is

$$\Phi^t = H^t(S, I) + \sum_{e=(u,v)\in E} [H^t(v, e) + H^t(u, e)] + \sum_{p \in P^t} T^t(p).$$

This potential function may change upon one of the following three events:

1. A new token enters the network – the potential function increases by at most n.
2. A token is sent – the potential function does not change, since the relevant H function decreases by exactly the value of the relevant new T function that is added to the sum.
3. A token is received – the potential function decreases by at least 1. This follows from Lemma 3.2, as the value of the function T that is extracted from the sum, is larger by at least 1 than the sum $message_tokens + nil_tokens$ at the accepting link, which is the exact increase in the corresponding function H.

Since by Corollary 3.4,

$$message_tokens[e]_v^t + nil_tokens[e]_v^t \leq n,$$

and since there is at most one TOKEN in transit on each link at any given time, the value of Φ is at most $2m \cdot (\binom{n}{2} + n) + \binom{n}{2}$; also Φ is clearly always non-negative. For each token received, Φ decreases by 1, and it can increase only upon the entry of a new token to the network and by at most n. Thus if in a given time interval k_{new} new tokens are introduced, then an upper bound on the number of token receipts in this time interval is $2m \cdot (\binom{n}{2} + n) + \binom{n}{2} + k_{\text{new}} \cdot n$, since otherwise the potential function Φ would become negative. Since by the code a TOKEN message is sent on link e only after an acknowledgment on the previous TOKEN message is received, the number of TOKEN messages sent in the time interval is at most $2m$ more than the number of TOKEN messages received in the time interval. Thus, we get property $(\mathcal{P}2)$.

We now turn to the proof of property $(\mathcal{P}3)$. By way of contradiction, assume that t is the last time at which a new token enters the network. As a result of property $(\mathcal{P}2)$ and as there is at most one TOKEN_LEFT message per TOKEN message, and one ACK message (of the corresponding type) per TOKEN and TOKEN_LEFT message, there is a time $t' \geq t$ after which no TOKEN, TOKEN_LEFT, ACK_TOKEN, or ACK_TOKEN_LEFT messages are sent. As the sender, S, and the receiver, R, are eventually connected, there is a path $R = v_0, v_1, \ldots, v_{k-1}, v_k = S$, $k < n$, such that for each $0 \leq i \leq k-1$, the edge $e = (v_i, v_{i+1})$ is viable, hence there is a time $t'' \geq t'$ by which all messages between v_i and v_{i+1}, in both directions, are delivered. Note that we enumerate the nodes on the path in the direction from the receiver to the sender.

Next, note that this implies that $tokens_left_pending[e]_{v_i} = 0$, for any i, and for any time after t''. We now prove by induction on the length of the viable path from v_i to R, that for any link e incident to v_i, after time t'', $message_tokens[e]_{v_i} + nil_tokens[e]_{v_i} \leq i$. The receiver, v_0, has no tokens stored at all, thus the induction basis holds. Let $e = (v_{i-1}, v_i)$ (for $i \geq 1$), and apply the induction hypothesis to the link e, in node v_{i-1}, i.e., $message_tokens[e]_{v_{i-1}} + nil_tokens[e]_{v_{i-1}} \leq i - 1$. Applying Lemma 3.1, and since after t'' there is no message in transit between v_{i-1} and v_i, and $tokens_left_pending[e]_{v_{i-1}} = 0$, we get $bound[e]_{v_i} \leq i$. As $t'' \geq t'$, no token is sent after t'', but according to the code this can happen only if for any time after t'', and any e incident to v_i, $message_tokens[e]_{v_i} + nil_tokens[e]_{v_i} \leq i$, proving the induction step.

Thus for the "virtual link" I at the sender S, $message_tokens[I]_S + nil_tokens[I]_S \leq k < n$, and by the code of the sender a new token is introduced into the network, contradicting the assumption. Property $(\mathcal{P}3)$ follows. □

4. Correctness Proof of the Protocol. In this section we prove the Safety and Liveness properties of the protocol.

THEOREM 4.1 (SAFETY). *At any time the output of the receiver is a prefix of the input of the sender.*

Proof. Denote by $IN = (I_1, I_2, \ldots)$ the input to the sender, and by $OUT = (O_1, O_2, \ldots)$ the output of the receiver. Denote by $t_i, i > 0$ the time at which O_i is output.

By the code the receiver outputs at t_i the value in *current_message* as the next output. We will therefore show that at t_i the value of *current_message* is I_i. For this,

we consider the tokens received by the receiver in the time interval between the time data item $i-1$ and data item i are output. We use the following definitions:

DEFINITION 4.2. *Let $in^{(t,t']}$ be the number of tokens introduced into the network by the sender in interval of time $(t,t']$. Let $out^{(t,t']}$ be the number of tokens received by the receiver in the interval of time $(t,t']$.*

First note that the total number of tokens (either nil-tokens or message-tokens) received by the receiver by time t_i is

$$out^{(t_0,t_i]} = \mathcal{C} + 1 + (i-1)(2\cdot\mathcal{C}+1).$$

By Theorem 3.5, the capacity of the network is $\mathcal{C}$, thus the total number of tokens sent by the sender by any time t is at most $\mathcal{C}$ more than the total number of tokens received by the receiver by the same time, t. Thus,

$$in^{(t_0,t_i]} \leq i(2\cdot\mathcal{C}+1).$$

Since the sender sends successively $2\mathcal{C}+1$ tokens for each data item, this implies that all tokens sent by time t_i carry the value I_j for some $j \leq i$.

As to the first data-item this guarantees that all TOKEN messages received by time t_1 carry the first data-item, and since at the beginning of the run *current_message* is initialized to `empty`, this guarantees that at time t_1 *current_message* is I_1.

To prove the claim for $i > 1$ we first show that more than half of the tokens received by the receiver in $(t_{i-1},t_i]$ that carry a data item when received (as opposed to nil tokens), carry the value I_i. Since $in^{(t_0,t_i]} \leq i(2\cdot\mathcal{C}+1)$ no token carrying I_k, for $k > i$ is present in the network by time t_i. Therefore, and since $out^{(t_0,t_i]} = (l-1)(2\mathcal{C}+1)+(\mathcal{C}+1)$, any token received in $(t_{i-1},t_i]$ is a token that is either:

- Sent with value I_i (note that no such token can be received by t_{i-1}).
- Sent with value I_k, $k < i$, and not received by time t_{i-1}. There can be at most $\mathcal{C}$ such tokens, as the total number of tokens ever sent with I_k $(k < i)$, is $(i-1)(2\mathcal{C}+1)$.

The set of tokens received in $(t_{i-1},t_i]$ is thus some subset of size $2\mathcal{C}+1$ of the above $2\mathcal{C}+1+\mathcal{C}$ tokens. We argue that in any subset of size $2\mathcal{C}+1$ of these tokens, more than half of the "real" tokens carry the value I_i. This is true along time, even after occurrence of "cancelation" events that cause two tokens to become nil. At t_{i-1} the claim is true since no one of the $2\mathcal{C}+1$ tokens to be sent with I_i is sent yet, thus they still "carry" I_i, while there are at most $\mathcal{C}$ other tokens. Consider any cancelation event, then if the number of tokens carrying I_i is reduced as a result of one such token becoming nil, then also a token carrying a value I_k, $k < i$, becomes nil.

We now show that indeed at time t_i the value of *current_message* is I_i. Consider the sequence of $2\mathcal{C}+1$ tokens received during the interval $(t_{i-1},t_i]$. For the purpose of the proof we give to each token that arrives at the receiver in the time interval $(t_{i-1},t_i]$, a number which is its number in the sequence of the tokens that arrive after time t_{i-1}. Let $1 \leq T_1 < T_2 < T_j \ldots < T_k \leq 2\mathcal{C}+1$ be the token numbers that upon their arrival

the value of *curret_message* changes from `empty` to another value. Let Val_j be the new value assigned to *curret_message*. We show that at time t_i *current_message* is not `empty`, and that the last "real" value assigned to it, Val_k, is I_i. Consider the set of tokens with numbers $T_j \leq l < T_{j+1}$, for $j < k$. If at T_{j+1} a new, non `empty`, value is assigned to *current_message*, then *current_message_count* was 0 at this time, which means that it has become such at the receipt of token number $T_{j+1} - p$, for some $p \geq 1$, and any token with number s, such that $T_{j+1} - p + 1 \leq s < T_{j+1}$ (if any) was a nil token. Thus, we conclude that among the tokens $T_j \leq l < T_{j+1}$ that carry a data-item (i.e., are not nil tokens) exactly half carry Val_j. Whatever the value Val_j is, *at most* half of the above tokens carry I_i. If upon the receipt of token $2\mathcal{C}+1$ *current_message* is `empty` (and is thus `empty` at time t_i), we have that over the whole sequence the number of tokens carrying I_i is at most half the number of tokens carrying any value. Thus, this cannot happen (i.e., *current_message* is not `empty` at time t_i). Next, assume that $Val_k \neq I_i$ (and this is the value at time t_i). Then *more* than half the tokens $T_k \leq l \leq 2\mathcal{C}+1$, that are not nil-tokens, carry value Val_k. Then, over the whole sequence the tokens carrying I_i are less than half the number of tokens carrying any value, a contradiction again. □

THEOREM 4.3 (LIVENESS). *If the sender and the receiver are eventually connected, then the receiver eventually outputs any data-item input to the sender.*

Proof. If the sender inputs the i'th data-item, then it tries to send $i(2\cdot\mathcal{C}+1)$ tokens (counted over the whole run). As the sender and the receiver are eventually connected, by Property ($\mathcal{P}3$) all these tokens are eventually input into the network. Since the network can delay at most $\mathcal{C}$ tokens, the receiver will eventually receive $i(2\cdot\mathcal{C}+1) - \mathcal{C}$ tokens, and thus outputs the i'th data-item. □

4.1. The Complexity of the Protocol. LEMMA 4.4. *The number of messages sent by the protocol in any time interval where k_{new} new tokens are added to the network is $O(n^2m + k_{\text{new}} \cdot n)$.*

Proof. Each message is a TOKEN, TOKEN_LEFT, ACK_TOKEN, or ACK_TOKEN_LEFT message. By Property ($\mathcal{P}2$), the number of TOKEN messages sent in the time interval is $O(n^2m + k_{\text{new}} \cdot n)$. The number of TOKEN_LEFT messages sent is at most the sum of the counters *token_left_pending* at the beginning of the interval, plus the number of TOKEN messages sent in the time interval. By Corollary 3.4 and the above, this sums to $O(2nm + n^2m + k_{\text{new}} \cdot n)$.

The number of ACK_TOKEN and ACK_TOKEN_LEFT messages sent, in the time interval under consideration, is at most the number of TOKEN and TOKEN_LEFT messages sent in this time interval, plus the number of messages that were in transit when the time interval started. Since the capacity of each link is constant the number of messages in transit at any given time is $O(m)$, and we have a bound on the number of ACK messages (of both types) of $O(n^2m + k_{\text{new}} \cdot n + 2mn + m)$. □

LEMMA 4.5. *The message complexity of the protocol is $O(n^2m)$ messages.*

Proof. For every time interval $(t_{i-1}, t_i]$ the receiver receives $2\cdot\mathcal{C}+1$ tokens during the interval. Since the capacity of the network is $\mathcal{C}$ tokens, at most $3\cdot\mathcal{C}+1$ tokens are

sent by the sender in $(t_{i-1}, t_i]$. As $\mathcal{C} = O(nm)$, the lemma follows from Lemma 4.4. □

Since messages have size at most the size of the data-item, we establish the following theorem:

THEOREM 4.6 (COMMUNICATION COMPLEXITY). *The communication complexity of the above protocol is $O(n^2 mD)$ bits.*

THEOREM 4.7 (SPACE COMPLEXITY). *The space required at each processor is $O(D + \log n)$ bits per incident link.*

Proof. The list of variables used by each processor (per incident link) is given at Section 3.2. By Corollary 3.4, the value of each of the counters *message_tokens*, *nil_tokens*, and *tokens_left_pending* is at most n, hence requires only $O(\log n)$ bits. By Lemma 3.3, the same is true for the counter *bound*. In addition there is a constant number of a single-bit flags, per link. Finally, the processors store a single variable, *current_message* of size D bits. □

5. Conclusions. In this work, we give an end-to-end communication protocol that has polynomial communication complexity and at the same time logarithmic space complexity. Thus we show that it is possible to attain polynomial communication complexity and sub-linear space complexity at the same time. It remains, however, open, whether *constant* space complexity (more precisely, space complexity $O(D)$) allows for polynomial, or even merely bounded, communication-complexity protocols in the above setting.

REFERENCES

[AAF+90] Y. Afek, H. Attiya, A. Fekete, M. J. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. D. Zuck. Reliable Communication over Unreliable Channel. *J. of the ACM* 40(5):1087-1107, 1993.

[AAG+97] Y. Afek, B. Awerbuch, E. Gafni, Y. Mansour, A. Rosén, and N. Shavit. Slide – the Key to Polynomial End-to-End Communication. *J. of Algorithms* 22(1):158-186, 1997.

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying Static Network Protocols to Dynamic Networks. In *Proc. of the 28th IEEE Ann. Symp. on Foundation of Computer Science*, pages 358–370, October 1987.

[AAM89] Y. Afek, B. Awerbuch, and H. Moriel. A Complexity Preserving Reset Procedure. Technical Report MIT/LCS/TM-389, MIT, May 1989.

[AE86] B. Awerbuch, and S. Even. Reliable Broadcast Protocols in Unreliable Networks. *Networks* 16(4):381-396, 1986.

[AG88] Y. Afek, and E. Gafni. End-to-End Communication in Unreliable Networks. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 131–148, August 1988.

[AG91] Y. Afek, and E. Gafni. Bootstrap Network Resynchronization: An Efficient Technique for End-to-End Communication. In *Proc. of the Tenth Ann. ACM Symp. on Principles of Distributed Computing*, pages 295–307, August 1991.

[AG90] A. Arora, and M. Gouda. Distributed Reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Springer-Verlag (LNCS 472), 1990.

[AGH90] B. Awerbuch, O. Goldreich, and A. Herzberg. A Quantitative Approach to Dynamic Networks. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 189–204, August 1990.

[AGR92] Y. Afek, E. Gafni, and A. Rosén. The Slide Mechanism with Applications in Dynamic Networks. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 35–46, August 1992.

[AKM+93] A. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time Optimal Self-Stabilizing Synchronization. In *Proc. 25th ACM Symposium on the Theory of Computing* , pages 652–661, 1992.

[AKY90] Y. Afek, S. Kutten, and M. Yung. Memory-Efficient Self-Stabilization on General Networks. In *Proc. 4th Workshop on Distributed Algorithms*, Italy, 1990.

[AL93] B. Awerbuch, and T. Leighton. A Simple Local-Control Approximation Algorithm for Multi-Commodity Flow. In *Proc. 34th IEEE Symposium on Foundations of Computer Science*, pages 459–469, 1993.

[AL94] B. Awerbuch, and T. Leighton. Improved Approximation Algorithms for the Multi-Commodity Flow Problem and Local Competitive Routing in Dynamic Networks. In *Proc. 26th ACM Symposium on the Theory of Computing*, pages 487–496, 1994.

[AM88] B. Awerbuch, and Y. Mansour. An Efficient Topology Update Protocol for Dynamic Networks. Unpublished manuscript, January 1988.

[AMS89] B. Awerbuch, Y. Mansour, and N. Shavit. Polynomial End to End Communication. In *Proc. of the 30th IEEE Ann. Symp. on Foundation of Computer Science*, pages 358–363, October 1989.

[AO94] B. Awerbuch, and R. Ostrovsky. Memory-Efficient and Self-Stabilizing Network RESET. In *Proc. of the 13th ACM Symp. on Principles of Distributed Computing*, pages 254–263, August 1994.

[APV91] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-Stabilization by Local Checking and Correction. In *Proc. of the 32nd IEEE Ann. Symp. on Foundation of Computer Science*, pages 268–277, October 1991.

[AS88] B. Awerbuch, and M. Sipser. Dynamic Networks Are as Fast as Static Networks. In *Proc. of the 29th IEEE Ann. Symp. on Foundation of Computer Science*, pages 206–220, October 1988.

[AV91] B. Awerbuch, and G. Varghese. Distributed Program Checking: A Paradigm for Building Self-Stabilizing Distributed Protocols. In *Proc. of the 32nd IEEE Ann. Symp. on Foundation of Computer Science*, pages 258–267, October 1991.

[BS88] A. E. Baratz, and A. Segall. Reliable Link Initialization Procedures. *IEEE Transaction on Communication*, February 1988. Also in: IFIP 3rd Workshop on Protocol Specification, Testing and Verification, III.

[Dij74] E. Dijkstra. Self Stabilization in Spite of Distributed Control. *Comm. of the ACM*, 17:643–644, 1974.

[DF88] E. W. Dijkstra, and W. H. J. Feijin. *A Method of Programming*. Addison-Wesley, 1988.

[DW93] S. Dolev, and J. Welch, Crash Resilient Communication in Dynamic Networks. In *Proc. of the 7th International Workshop on Distributed Algorithms*, Springer-Verlag LNCS 725, pp. 129-144, 1993.

[Fin79] S. Finn. Resynch Procedures and a Fail-Safe Network Protocol. *IEEE Trans. on Commun.*, COM-27(6):840–845, June 1979.

[IL94] G. Itkis, and L. Levin. Fast and Lean Self-stabilizing Asynchronous Protocols. In *Proc. of the 35th IEEE Ann. Symp. on Foundation of Computer Science*, pages 226–239, November 1994.

[KP90] S. Katz, and K. Perry. Self-Stabilizing Extensions for Message-Passing Systems. In *Proc. of the 9th ACM Symp. on Principles of Distributed Computing*, pages 91–101, August 1990.

[MOOY92] A. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung. Self-Stabilizing Symmetry Breaking in Constant-Space. In *Proc. 24th ACM Symposium on the Theory of Computing*, pages 667–678, 1992.

[Vis83] U. Vishkin. A Distributed Orientation Algorithm. *IEEE Info. Theory,* June 1983.

A. Proofs of Technical lemmas.

Proof of Lemma 3.1. Upon initialization, the invariant holds. This is because the variable $bound[e]$ is initialized to 1, the counters $message_tokens[e]$, $nil_tokens[e]$ and $tokens_left_pending[e]$ are initialized to 0, and no message is in transit in the network. By induction on the events that change any of the values participating in the invariant, we show that the invariant holds for any time t. There are six types of events to be considered: send and receive events of TOKEN messages from u to v on e; send and receive events of TOKEN_LEFT messages from v to u on e; send events of TOKEN messages from v, when the token sent is accounted for in the counters of e in v; and receive events of a TOKEN at v, when this arrival causes a token accounted for at e to become a nil token.

If both v and u are *not* the receiver then the following describes the effects of any of the events:

- Send event of a TOKEN message from u to v on e: $bound[e]_u$ is incremented by 1, but so is $tokens_{u\to v}$.
- Receive event of a TOKEN message at v on link e: The sum $message_tokens[e]_v + nil_tokens[e]_v$ is incremented by 1, but $tokens_{u\to v}$ is decremented by 1.
- Send event of a TOKEN_LEFT message from v to u on e: $tokens_left_pending[e]_v$ is decremented by 1, but $tokens_left_{v\to u}$ is incremented by 1.
- Receive event of a TOKEN_LEFT message at u from v, on e: $bound[e]_u$ is decremented by 1, but so is $tokens_left_{v\to u}$.
- A Send event of a TOKEN message from u on some link e', when the token sent was accounted for at the counters of e: $message_tokens[e]$ and $nil_tokens[e]$ is decremented by 1, but $tokens_left_pending[e]$ is incremented by 1.
- A Receive event of a TOKEN message at v (on some link) may cause a token accounted for at the counters of e to become nil: $message_tokens[e]$ is decremented by 1, but $nil_tokens[e]$ is incremented by 1.

If v is the receiver:

- Send event of a TOKEN message from u to v on e: $bound[e]_u$ is incremented by 1, but so is $tokens_{u\to v}$.
- Receive event of a TOKEN message at v on e: $tokens_{u\to v}$ is decremented by 1 but $tokens_left_pending[e]$ is incremented by 1.
- Send event of a TOKEN_LEFT message from v to u on e: $tokens_left_pending[e]_v$ is decremented by 1, but $tokens_left_{v\to u}$ is incremented by 1.
- Receive event of a TOKEN_LEFT message at u from v, on e: $bound[e]_u$ is decremented by 1, but so is $tokens_left_{v\to u}$.
- A Send event of a TOKEN message from u on some link e', when the token sent was accounted for at the counters of e: One of the counters $message_tokens[e]$ and $nil_tokens[e]$ is decremented by 1, but $tokens_left_pending[e]$ is incremented by 1 (However, note that this event cannot happen, since u never receives any token on e).
- A Receive event of a TOKEN message at v (on some link) never changes the values of $message_tokens[e]$ or $nil_tokens[e]$.

If u is the receiver:

- A send event of a TOKEN message from u no e cannot happen, by the code.
- Receive event of a TOKEN message at v on link e: The sum $message_tokens[e]_v + nil_tokens[e]_v$ is incremented by 1, but $tokens_{u \to v}$ is decremented by 1 (note, however, that such an event cannot happen as u does not send TOKEN messages).
- Send event of a TOKEN_LEFT message from v to u on e: $tokens_left_pending[e]_v$ is decremented by 1, but $tokens_left_{v \to u}$ is incremented by 1 (note that this event cannot happen too).
- Receive event of a TOKEN_LEFT message at u from v, on e: $bound[e]_u$ is decremented by 1, but so is $tokens_left_{v \to u}$ (note that this event cannot happen).
- A send event of a TOKEN message accounted for at u cannot happen, by the code.
- A Receive event of a TOKEN message at v (on some link) may cause a token accounted for at the counters of e to become nil: $message_tokens[e]$ is decremented by 1, but $nil_tokens[e]$ is incremented by 1.

□

Proof of Lemma 3.2. Let t be the time just before the token is sent from u, and t' the time just before it is received at v. Because the sum of $message_tokens$ and nil_tokens can increment only when tokens arrive on the link, and because the links are FIFO, we have:

$$\begin{aligned} & message_tokens[e]_v^{t'} + nil_tokens[e]_v^{t'} \\ \leq\ & message_tokens[e]_v^{t} + nil_tokens[e]_v^{t} + tokens_{u \to v}^{t}. \end{aligned}$$

By Lemma 3.1,

$$message_tokens[e]_v^t + nil_tokens[e]_v^t + tokens_{u \to v}^t + 1 \leq bound[e]_u^t \ .$$

Thus,

$$message_tokens[e]_v^{t'} + nil_tokens[e]_v^{t'} + 1 \leq bound[e]_u^t.$$

By the code $i > bound[e]_u^t$ and $j = message_tokens[e]_v^{t'} + nil_tokens[e]_v^{t'} + 1$, hence $i > j$.

□

Proof of Lemma 3.3. We prove the claim by contradiction. Assume $bound[e]_u^t > n$ for some t, u, and $e = (u, v)$. Then a token must have been sent over e from u to v when $bound[e]_u \geq n$. By the code, this can happen only if, for some $e' \in E \cup \{I\}$, $message_tokens[e']_u + nil_tokens[e']_u > n$. However, for any e', the value of $message_tokens[e']_u + nil_tokens[e']_u$ is initialized to 0. Therefore, consider the first time that, for some e' and u, $message_tokens[e']_u + nil_tokens[e']_u > n$. For the "virtual link" I at the sender this cannot happen by the code. For every $e' = (u, v') \in E$, the value of $message_tokens[e']_u + nil_tokens[e']_u$ increases only when a token is received at u over e'. Consider the token that upon its receipt the value of $message_tokens[e']_u +$

$nil_tokens[e']_u$ became strictly larger than n, and denote by e'' the link incident to v whose counters accounted for this token before it was sent from v. By Lemma 3.2, when the token was sent from v the value of $message_tokens[e'']_v + nil_tokens[e'']_v$ was already strictly greater than n, contradicting the fact that we are considering the first such event. □

Proof of Lemma 3.4. For $e = I$, the claim follows from the code. For any $e \in E$, by Lemmas 3.1 and 3.3,

$$message_tokens[e]_v^t + nil_tokens[e]_v^t + tokens_{u \to v}^t \;\; \leq \;\; bound[e]_u^t - 1 \leq n - 1 \;\; ,$$

and

$$tokens_left_pending[e]_v^t \leq bound[e]_u^t - 1 \leq n - 1 \;\; .$$

□