LOG-Space Polynomial End-to-End Communication

Eyal Kushilevitz[*] Rafail Ostrovsky[†] Adi Rosén[‡]

TR-94-068

December 1994

[*]Dept. of Computer Science, Technion, Haifa 32000, Israel. E-mail: eyalk@cs.technion.ac.il . 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.

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 is a classical problem that asks how to carry out fault-free communication between two processors over a network, in spite of such *frequent* communication faults. 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, at any time, any operational path between the two processors that are trying to communicate.)

For the first time, we present a protocol which solves this fundamental 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 edge.

Our algorithm maintains a simple-to-compute $O(\log n)$-bits potential function at each edge in order to perform routing, and uses a novel technique of packet canceling which allows us to keep only *one* packet per edge. We stress that both the computation of our potential function and our packet-canceling policy are totally local in nature; we believe that they are applicable to other settings as well.

1 Introduction

In this paper we address one of the most fundamental problems in distributed computing: 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 (hereafter referred to as "communication faults").

<u>COMMUNICATION DURING INFREQUENT FAULTS:</u> If communication faults are "infrequent", then after each fault a new path between the two communicating processors can be computed, which will be operational till the next fault. That is, in case faults occur *rarely*, it is possible, upon a fault, to "reset" the network and compute a new path between the communicating nodes (e.g., [Fin79, AAG87, AS88, AAM89, AGH90]). Alternatively, a reliable communication path can 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). Notice that both the reset and the self-stabilizing solutions work *only if faults are not too frequent.*

<u>COMMUNICATION DURING FREQUENT FAULTS:</u> We consider a setting where faults occur very frequently. The so-called end-to-end communication problem [AG88, AMS89] is to deliver, in finite time, data-items from a sender node to a receiver node, 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 node. This should be done even if there does not exist at any time a path of active links that connects the two communicating parties. The sole assumption is that the two communication 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.*

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

<u>SPACE COMPLEXITY:</u> Another important complexity measure is the *space complexity* — the amount of space needed at the nodes, 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 setting on *infrequent* communication faults, small memory and communication efficiency are simultaneously attainable. In contrast, in the setting of *frequent* communication faults, all 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 a *linear* (in the number of nodes of the network) amount of space, at each node, 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

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

has exponential communication complexity, and another protocol which uses constant amount of space but has unbounded communication complexity.

Our Result: The question whether there exists an end-to-end communication protocol with sub-linear space complexity and at the same time polynomial communication complexity remained open. In this paper, we give an affirmative answer to this question: we exhibit 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 nodes and links in the network respectively, and D is data-item size. This is an *exponential space complexity improvement* over all known polynomial-communication protocols. 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**: $O(D \cdot \exp(n))$	**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] and the Majority algorithm of [AGR92, AAF+90]. The Slide protocol requires keeping $O(n)$ packets per incident edge, and decides on the recency of the received item only at the receiver node (using [AAF+90] technique.). If we wish to reduce space per edge, we can no longer afford keeping $O(n)$ different packets, but must keep far fewer packets per edge, and we can no longer afford the Majority algorithm of [AAF+90], which collects a large number of packets arriving at the receiver and then takes their majority. 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 node, yet, manage to compute the majority value. However, if we are beginning 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:

- A potential function that controls the flow of data-items in the network; and
- A novel data-item cancelling policy which makes sure that in any node there will be at most two distinct values of data-items per edge. The same policy is used both in the intermediate nodes and at the receiver node.

We show that a careful combination of these two techniques yields the desired result. Hence, even in the frequent communication faults model, it is possible to achieve *both* polynomial communication and logarithmic space. These techniques exponentially improve the space efficiency of all end-to-end polynomial-communication protocols; as in [AMS89, AGR92], they have the additional benefit that they are totally local in nature. For example, the locality of Slide was used for establishing its self-stabilizing extension in [APV91], as well as for various multi-commodity flow problems in dynamic graphs [AL93, AL94].

ORGANIZATION: Section 2 contains all the necessary background including 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 somewhat more detailed description (Section 3.2); Then we prove some of the properties of the protocol (Section 3.3) and conclude with its proof of correctness and complexity (Sections 3.4 and 3.5 respectively). The code itself as well as some of the proofs appear in the Appendix.

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. Each *undirected* link consists 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 nodes. 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. Below we describe the properties of each *directed* link:

- Each link has *constant* capacity; that is, only a constant number of messages is allowed to be in transit on a given link at any given time.
- Communication over links obeys the *FIFO* rule; that is, the sequence of messages *received* over the link is a prefix of the sequence of messages sent over the link.
- Communication is *asynchronous*: 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 of the two 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.

2.2 The End-to-End Problem

The purpose of the 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 a second special processor, called the *receiver*. It is required that this virtual link be viable if the sender is eventually connected to the receiver. 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 is a *proper 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.

An algorithm 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. That is, 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, the size of the data-item).

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

In addition, we require that the local routing computation of the network be polynomial for each send/receive event at each node, (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 deal with networks that frequently change their topology; 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 end-points by some underlying link protocol. This model should be contrasted with a 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, this corresponds to *infrequent faults*, and is incomparable to the model of *frequent faults* addressed here. The question how, in addition to frequent faults of edges, one can allow bad initial state was addressed in the end-to-end setting by [APV91].

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)[2].

3 The Protocol

3.1 High-level Description

Our starting point is a linear space, yet simple, solution of [AGR92]. This solution combines (as black-boxes) two components: the Slide protocol and the Majority algorithm. Before we explain our algorithm, we give a quick overview of the approach taken there. The slide is used to transfer tokens (data-items) in the network. This is done by letting each node maintain a stack of tokens for each of its edges. On each edge, 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 has no stack hence it only receives tokens. The Majority algorithm enables the receiver to decide, by collecting 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, 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, and the majority algorithm needs even more space to collect the tokens in order to decide on the correct value. To overcome this, we introduce the following idea:

- While transferring packets from the sender to the receiver, our algorithm "cancels" some of these packets *both* en route and upon their arrival at the receiver node. 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 node (including the receiver) at any given time – a "real" packet, and a "nil" value. We use a potential function that only *counts* the packets of both types, rather than storing all of them and use this function to control the flow of tokens. We prove that 2 counters of $\log n$ bits each are sufficient for this.

Note that, usually, algorithms that change the values of packets are undesirable. However, our algorithm changes the values in a very restricted way – it may replace a "real" packet by a "nil"

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

packet. There is no other use of the content of packets. Our "canceling" policy does not effect the routing properties of the algorithm: it guarantees that if the sender and the receiver are eventually connected, then data-items will be transferred from the sender to the receiver. Moreover, our algorithm guarantees an upper bound on the number of tokens that are in transit 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 (i.e. $2\mathcal{C}+1$) number of tokens (some of which 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.*[3] We emphasize that the Receiver computes this majority *without storing the tokens*; rather, it does so by using the same "canceling policy" as in the intermediate nodes.

As it is clear from the above description, the heart of our algorithm is the new (local) "cancelling policy", described below. Whenever a token arrives into a node we do the following:

- If it is a nil token, then the counter of nil tokens is augmented by one.
- If it is a data token and the data-item is identical to the data-item currently stored in the node, then the data tokens counter is augmented by one.
- If it is a data token which is different than the data-item currently stored in the node, then the arriving token, and one data-item already accounted for in the node both are "cancelled" and become two nil tokens (and the counters are updated accordingly); if as a result there are no more data tokens in the node, we erase the current data-item stored in the node.
- If it is a data token and there is no data-item currently stored in the node, the arriving data-item becomes the current one.

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 items has only a minor effect; the receiver only needs to ignore the nil-items. 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 data-item canceled is an old one. Therefore, the majority of the correct data-item is maintained.

3.2 A Formal Description of the Algorithm

We begin by describing the data-structures and messages used by our algorithm. The protocol uses three types of messages:

TOKEN messages:	to carry data-items (either "real" data-items or the 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.

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

ACK messages: are used to acknowledge the arrival of a TOKEN message.

The following data is stored at each node:

- 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 $nil\_tokens[e]$ that count how many TOKEN messages can (potentially) be sent from the node. The first is used to count those TOKEN messages that will carry the data-item stored in *current_message* and the second to count those messages that will carry the nil "data-item".
- 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 1 every time a token is sent over the outgoing link, and decremented by 1 every time a TOKEN_LEFT message is received over the corresponding incoming link.
- For each incident link e, a flag $free\_link[e]$ indicating if an ACK message has already been received for the previous token sent on e.

The code itself appears in the appendix. Throughout the proofs we assume a global time, unknown to the nodes, and we denote the value of variables in a node at a given time by a subscript of the node 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 $\underline{tokens_{u\to v}^t}$ be the number of TOKENs in transit from u to v at time t. Let $\underline{acks_{u\to v}^t}$ be the number of ACKs in transit from u to v at time t. Let $\underline{signals_{u\to v}^t}$ be the number of TOKEN_LEFT messages in transit from u to v at time t.

3.3 Properties of the Algorithm

In this section we present properties of the protocol which later allow us to prove its correctness and complexity. Some of the proofs are similar to those of [AMS89, AGR92]. We first state the following technical lemmas (whose proofs appear in the Appendix).

1. At any time t and for any $e = (u, v)$, $message\_tokens[e]_v^t + nil\_tokens[e]_v^t + tokens_{u\to v}^t \leq n$ (Lemma 8). In addition, $bound[e]_u^t \leq n$.

2. Consider a TOKEN message sent from node u to node v, on link $e = (u, v)$. Let e' be the edge whose counters accounted for the message sent. If just before it is sent $message\_tokens[e']_u + nil\_tokens[e']_u = i$ and just after its receipt $message\_tokens[e]_v + nil\_tokens[e]_v = j$, then $j < i$ (Lemma 7).

Then, we use the above lemmas to prove the following theorem:

Theorem 3.1 The protocol has the following properties:

$\mathcal{P}1$. At any time t, the number of tokens in the network is bounded by $2nm$. ($\mathcal{C} = 2nm$).

$\mathcal{P}2$. In any time interval in which `new` new tokens are inserted into the network, at most $O(n^2m + \texttt{new} \cdot n)$ TOKEN messages are sent.

$\mathcal{P}3$. If the sender and the receiver are eventually connected, the sender will eventually introduce a new token into the network.

Proof: Since all the tokens in the network are either "stored" in the the nodes or in transit over links, Lemma 8 proves property ($\mathcal{P}1$).

We can now also prove property ($\mathcal{P}2$). Define the following potential function. For any node v and for any incident link $e = (v,u)$ denote $J = message\_tokens[e]^t_v + nil\_tokens[e]^t_v$ and let $H^t(v,e) = \sum_{k=1}^{J} k = \binom{J}{2}$. Also define a function $T^t(p)$ which operates on a token p. $T^t(p) = 0$ if the token p is in any node at time t. If p is in transit at time t then let t' be the time just before it was sent from v to u. If e' is the the edge whose counters accounted for the token sent, then $T^t(p) = message\_tokens[e']^{t'}_v + nil\_tokens[e']^{t'}_v$ (that is, the token "carries" the "number" of tokens in v, at edge e' just before it left this node). The potential function is $\Phi^t = \sum_{e=(u,v)\in E} H^t(v,e) + H^t(u,e) + \sum_p T^t(p)$. This potential function may change upon one of the following three events:

1. A new token p enters the protocol – the potential function increases by n.
2. A token is sent – the potential function does not change, since the relevant H function decreases by exactly the same amount that the relevant T function increases.
3. A token is received – the potential function decreases by at least 1. This follows from Lemma 7, as the value of the function T that becomes 0 is larger by at least 1 than the sum $message\_tokens + nil\_tokens$ at the accepting edge, which is the exact increase in the corresponding function H.

Since the number of tokens in the network at any given time is at most $2nm$, the value of Φ is at most $2n^2m$; Also Φ is clearly always non-negative. For each token received it decreases by 1, and it can increase only upon the entry of a new token to the protocol and by n. This proves property $\mathcal{P}2$.

The proof of property ($\mathcal{P}3$) is postponed to the Appendix. ■

The following lemma bounds the amount of space needed for the link-queues of the link-level protocols (see the formal statement of the algorithm in the Appendix). We show that these queues can be maintained by having an $O(\log n)$ space counter for pending TOKEN_LEFT messages, and additional two buffers, one for a single TOKEN message and one for a single ACK message. The proof appears in the appendix.

Lemma 1 At any time t, there are at most n TOKEN_LEFT messages, one TOKEN message, and one ACK message in transit in each direction on any link.

3.4 Correctness Proof of the Algorithm

In this section we prove the Safety and Liveness properties of the protocol.

Theorem 3.2 (Safety) At any time the output of the receiver is a prefix of the input of the sender.

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

To prove the theorem, we claim that more than half the message tokens (as opposed to nil tokens) received by the receiver in the interval of time $(t_{i-1}, t_i]$, carry data-item I_i. First, we show that no message token that carries I_k, $k > i$ could have been received before t_i. We use the following definitions to count the number of tokens in the system.

Definition 1 Let $\underline{in}^{(t,t']}$ be the number of tokens input by the sender in interval of time $(t, t']$. Let $\underline{out}^{(t,t']}$ be the number of tokens received by the receiver in the interval of time $(t, t']$.

Definition 2 Let t_0 be some time before the execution of the algorithm started. Define $delay^t = in^{(t_0,t]} - out^{(t_0,t]}$ (the number of tokens that are in the network at time t).

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

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

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

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

Therefore, no token carrying I_k, $k > i$ can be sent by the sender before t_i. Hence, no such token can be received by the receiver at t, $t < t_i$.

As to the first data-item this guarantees that all TOKEN messages received actually carry the first data-item.

As to the next data-items, distinguish between three sets of tokens: nil-tokens, tokens carrying data-item I_i, and tokens that carry data-item I_j for $j < i$, called "old". We wish to show that in the $2 \cdot \mathcal{C} + 1$ tokens received in the interval of time $(t_{i-1}, t_i]$, the number of tokens carrying data-item I_i is greater than the number of "old" tokens. The maximum number of "old" tokens ever sent is $(2 \cdot \mathcal{C} + 1)(i-1)$, and (as shown above) all tokens received by time t_{i-1} are old. Therefore in $(t_{i-1}, t_i]$, the receiver can receive at most $\mathcal{C}$ "old tokens". The receiver receives in $(t_{i-1}, t_i]$ a total of $2\mathcal{C} + 1$ tokens, therefore if none of the $2\mathcal{C} + 1$ tokens carrying data-item I_i has become a nil-token during its transit in the network then we are done. Assume that N such tokens have become nil tokens. However, at the same time that such a token has become nil an "old" token, which was still in the network, and did not reach (yet) the receiver, has become a nil-token too. Thus the total number of "old" tokens the receiver can receive is also decreased by one. Therefore, we still have a majority of tokens carrying data-item I_i over the "old" tokens.

■

Theorem 3.3 (Liveness) If the sender and the receiver are eventually connected, then the receiver eventually outputs any data-item given 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. ■

3.5 The Complexity of the Algorithm

Lemma 2 The number of messages sent by the protocol in any time interval where `new` new tokens are added to the network is $O(n^2 m + \texttt{new} \cdot n)$.

Proof: The only messages in the protocol are TOKEN messages, TOKEN_LEFT messages, and ACK messages. There are exactly one TOKEN_LEFT message, and one ACK message per TOKEN message. The lemma thus follows from Property ($\mathcal{P}2$). ■

Lemma 3 The message complexity of the protocol is $O(n^2 m)$ messages.

Proof: Clearly in $(t_{i-1}, t_i]$ the receiver receives $2 \cdot \mathcal{C} + 1$ tokens. Since the network can hold at most $\mathcal{C}$ 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 2. ■

Since message has size at most the size of the data-item, we establish the following corollary.

Corollary 4 (Communication Complexity) The communication complexity of the protocol is $O(n^2 mD)$ bits, where D is the size in bits of a data-item.

Lemma 5 (Space complexity) The space required at each node is $O(D + \log n)$ bits per incident link, where D is the size in bits of a data-item.

Proof: The list of variables stored at each node (per incident link) is given at Section 3.2. By Lemma 8, the value of each of the counters *message_tokens* and *nil_tokens* is at most n, hence requires only $O(\log n)$ bits. The same is true for the counter *bound*. In addition each node stores a constant number of a single-bit flags. Finally, for the link-level protocol, each nodes uses (per link) buffers of size $O(D + \log n)$ (Lemma 1). ■

References

[AAF+90] Y. Afek, H. Attiya, A. Fekete, M. J. Fischer, N. Lynch, Y. Mansour, D. Wang, L. D. Zuck. Reliable Communication Over Unreliable Channel. Manuscript. 1990.

[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.

[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 (PODC)*, August 1991.

[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.

[AKY90] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, Italy, 1990.

[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.

[AE86] B. Awerbuch and S. Even Reliable broadcast protocols in unreliable networks *Networks* 16(4):381-396, 1986.

[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.

[AL93] B. Awerbuch and T. Leighton A simple local-control approximation algorithm for multi-commodity flow. FOCS-93.

[AL94] B. Awerbuch and T. Leighton Improved Approximation Algorithms for the Multi-Commodity Flow Problem and Local Competitive Routing in Dynamic Networks. STOC-94.

[AM88] B. Awerbuch and Y. Mansour. An efficient topology update protocol for dynamic networks. Unpublished manuscript, January 1988.

[AKM+93] A. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese Time Optimal Self-stabilizing Synchronization STOC-93.

[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.

[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.

[AO94] B. Awerbuch, and R. Ostrovsky Memory-Efficient and Self-Stabilizing Network RESET, PODC-94

[AV91] B. Awerbuch, and G. Varghese, Distributed program checking: a paradigm for building self-stabilizing distributed protocols, FOCS-91.

[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.

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

[KP90] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. PODC-90.

[IL94] G. Itkis, L. Levin. Fast and Lean Self-Stabilizing Asynchronous Protocols FOCS-94.

[MOOY92] A. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung Self-Stabilizing Symmetry Breaking in Constant-Space STOC-92.

[Vis83] U. Vishkin A distributed orientation algorithm *IEEE Info. Theory,* June 1983.

A APPENDIX: The Code

In this section, we formally state 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

$$\textbf{Select } G_1 \rightarrow A_1 \Box G_2 \rightarrow A_2 \Box \ldots G_l \rightarrow A_l \textbf{ End Select}.$$

The code is executed by repeatedly selecting an arbitrary i from all guards G_i which are true and executing A_i. A 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 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 puts the message m on a queue of outstanding message for the link e. An additional link-level protocol ensures the delivery of the messages in the queue over the link e, by sending the head of the queue once an acknowledgment for the previous one is received. We describe the protocol in this way to simplify the presentation. In our analysis we consider the space needed for the link-queues.

We present the code of a regular node (Figure 1), which is every node except the receiver, whose code is presented separately (Figure 3). For the sender we define an additional virtual node (Figure 2), the sender, which is connected to the "real" node by a single link, denoted simply e in the code. The node runs separately the code of a regular node, having as an additional edge this virtual edge.

```
Select
Initialization ⟶
   current_message=nil
   for every incident link $e$
            bound[$e$]:=1;
            message_tokens[$e$]:=0;
            nil_tokens[$e$]:=0;
            free_link[$e$]:=TRUE ;
□
Receive TOKEN_LEFT on $e$ ⟶
   bound[$e$]:=bound[$e$]-1;
□
Receive TOKEN(data) on $e$ ⟶
   if (data=nil) then
            nil_tokens[$e$]:=nil_tokens[$e$]+1 ;
   else
            if (current_message=nil or current_message=data) then
                        message_tokens[$e$]:=message_tokens[$e$]+1 ;
                        current_message=data;
            else
                        nil_tokens[$e$]:=nil_tokens[$e$]+2 ;
                        message_tokens[$e$]:=message_tokens[$e$] -1;
                        if (for every $e$ message_tokens[$e$]=0) then
                                    current_message: =nil ;
                        endif
            endif
   endif
   Send ACK on $e$ ;
□
Receive ACK on $e$ ⟶
free_link[$e$]:=TRUE ;
□
$\exists e, e'$ s.t. nil_tokens[$e'$]+message_tokens[$e'$] $>$ bound[$e$] and free_link[$e$]=TRUE ⟶
            /* $e'$ not necessarily $\neq e$ */
   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_link[$e$]:=FALSE ;
   bound[$e$]:=bound[$e$]+1;
   send TOKEN_LEFT on $e'$;

End Select
```

Figure 1: Ordinary node code

```
Select
Initialization ⟶
    current_message=nil
    bound[$e$]:=1;
    free_link[$e$]:=TRUE ;
    left_cur_message :=0 ;
□
left_cur_message = 0 ⟶
    current_message := input data-item ;
    left_cur_message:= 2 C+1 ;
□
free_link[$e$] and bound[$e$] $\leq$ n ⟶
    Send TOKEN(current_message) on $e$;
    bound[$e$]:=bound[$e$]+1;
    free_link[$e$]:=FALSE ;
    left_cur_message:=left_cur_message - 1;
□
Receive ACK on $e$ ⟶
    free_link[$e$]:=TRUE ;
□
Receive TOKEN_LEFT on $e$ ⟶
    bound[$e$]:=bound[$e$]-1;
□
End Select
```

Figure 2: "virtual" Sender's code

B APPENDIX: Proofs

Lemma 6 At any time t and for any $e = (u, v)$,

$$bound[e]^t_u - 1 = message\_tokens[e]^t_v + nil\_tokens[e]^t_v + tokens^t_{u \to v} + signals^t_{v \to u} \ .$$

In words, this relates the estimate that u has on the number of token in the other side of e (i.e., $bound[e]^t_u$) to the actual number of tokens (i.e., $message\_tokens[e]^t_v + nil\_tokens[e]^t_v$) and those which are still in transit (i.e., $tokens^t_{u \to v} + signals^t_{v \to u}$).

Proof: Upon initialization, the invariant holds, since $bound[e]$ is initialized to 1, the counters $message\_tokens[e]$ and $nil\_tokens[e]$ 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 it holds for any t. There are four types of events to be considered: send and receive events of TOKEN messages from u to v, and send and receive events of TOKEN_LEFT messages from v to u. Consider the first case – a send event of a TOKEN message from u to v: $bound[e]_u$ is incremented by 1, but so is $tokens_{u \to v}$. The other three cases are proved similarly. ■

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

Lemma 7 Consider a TOKEN message sent from node u to node v, on link $e = (u, v)$. Let e' be the edge whose counters accounted for the message sent. If just before it is sent $message\_tokens[e']_u + nil\_tokens[e']_u = i$ and just after its receipt $message\_tokens[e]_v + nil\_tokens[e]_v = j$, then $j < i$.

Proof: 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 both $message\_tokens$ and nil_tokens are incremented only when tokens

```
Procedure check_and_output
    if first_item and $count = \mathcal{C} + 1$ then
                                                                    /* first data-item */
            output(current_message);
            current_message = nil ;
            count:= 0 ;
            current_message_count:= 0 ;
            first_item:=false;
    else
            if (not first_item) and $count = 2\cdot\mathcal{C} + 1$ then
                                                                    /* all other data-items */
                            output(current_message);
                            current_message = nil ;
                            count := 0 ;
                            current_message_count:= 0 ;
            endif
    endif
End Procedure
Initialize ⟶
    current_message = nil ;
    current_message_count:=0 ;
    count:=0 ;
    first_item:=TRUE;
□
Receive TOKEN(data) on $e$ ⟶
    count:=count + 1;
    if (data $\neq$ nil) then
            if (current_message=nil 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: =nil ;
            endif
    endif
    Send ACK on $e$ ;
    call check_and_output;
End Select
```

Figure 3: Receiver's code

arrive on the link, and because the links are FIFO, we have:

$$message\_tokens[e]_v^{t'} + nil\_tokens[e]_v^{t'} \leq message\_tokens[e]_v^t + nil\_tokens[e]_v^t + tokens_{u\to v}^t.$$

By Lemma 6,

$$message\_tokens[e]_v^{t'} + nil\_tokens[e]_v^{t'} + 1 \leq bound[e]_u^t.$$

By their definition $i > bound[e]_u^t$ and $j = message\_tokens[e]_v^{t'} + nil\_tokens[e]_v^{t'}$, hence $i > j$. ■

Since all the tokens in the network are either "stored" in the the nodes or in transit over links, the following lemma proves property ($\mathcal{P}1$).

Lemma 8 At any time t and for any $e = (u, v)$,

$$message\_tokens[e]_v^t + nil\_tokens[e]_v^t + tokens_{u\to v}^t \leq n \ .$$

Proof: By Lemma 6, $message\_tokens[e]_v^t + nil\_tokens[e]_v^t + tokens_{u\to v}^t \leq bound[e]_u^t - 1$. For $bound[e]_u$ to be strictly greater than n, a token must be sent over e when $bound[e]_u = n$. By the code, this can happen only if, for some e', $message\_tokens[e']_u + nil\_tokens[e']_u > n$. By Lemma 7, such an event cannot exist. Thus for any t $bound[e]_u^t \leq n$. ■

Proof of ($\mathcal{P}3$), Theorem 3.1: 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 only one TOKEN_LEFT message and one ACK message per TOKEN message, there is a time $t' \geq t$ after which no TOKEN or TOKEN_LEFT or ACK 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 $0 \leq i \leq k-1$, $e = (v_i, v_{i+1})$ 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.

By induction on the length of the viable path from v_i to R, we will show that for any edge e incident to v_i, after t'' $message\_tokens[e]_{v_i} + nil\_tokens[e]_{v_i} \leq i$. The receiver, v_0, has no tokens stored at all. Denote by e the (v_{i-1}, v_i) link $(i \geq 1)$, and assume the induction hypothesis, applying it to the edge $e = (v, v_i)$, in v_{i-1}, i.e., $message\_tokens[e]_{v_{i-1}} + nil\_tokens[e]_{v_{i-1}} \leq i - 1$.

Since at t'' all messages between v_{i-1} and v_i have arrived, by Lemma 6 and the inductive hypothesis $bound[e]_{v_i}^{t''} \leq i$. As $t'' \geq t'$, no token is sent after t'', but according to the code this can happen only if v_i, for any time after t'', and any e incident to v_i, $message\_tokens[e]_{v_i} + nil\_tokens[e]_{v_i} \leq i$. Thus at S, $message\_tokens[e]_S + nil\_tokens[e]_S \leq k < n$, and by the code the "virtual sender" will introduce a new token into the network, contradicting the assumption. Property $\mathcal{P}3$ follows.

Proof of Lemma 1: By Lemma 6, for any e, $e = (u, v)$, and any time t

$$tokens_{u\to v}^t \leq bound[e]_u^t - 1, \text{ and } signals_{v\to u}^t \leq bound[e]_u^t - 1 \ .$$

By the same arguments as in the proof of Lemma 8, $bound[e]_u^t \leq n$, for any t. Hence $signals_{v\to u}^t \leq n$. A TOKEN message is sent only after an ACK for the previous one was received, therefore there can be at most one TOKEN message in transit. The same applies for the ACK message.

The same arguments hold for the opposite directions, thus on any link at any time there are at most one TOKEN, one ACK message messages and n TOKEN_LEFT messages in each direction. ■