Slide - The Key to Polynomial End-to-End Communication *

Yehuda Afek† & Baruch Awerbuch ‡ & Eli Gafni § &
Yishay Mansour ¶ & Adi Rosén ‖ & Nir Shavit **

Abstract

We consider the basic task of of *end-to-end communication* in dynamic networks, that is, delivery in finite time, of data items generated on-line by a sender, to a receiver, in order and without duplication or omission.

A dynamic communication network is one in which links may repeatedly fail and recover. In such a network, though it is impossible to establish a communication path consisting of non-failed links, reliable communication is possible, if there is no cut of permanently failed links between a sender and receiver.

This paper presents the first polynomial complexity end-to-end communication protocol in dynamic networks. In the worst case the protocol sends $O(n^2m)$ messages per data item delivered, where n and m are the number of processors and number of links in the network respectively. The centerpiece of our solution is the novel *slide* protocol, a simple and efficient method for delivering tokens across an unreliable network. *Slide* is the basis for several self-stabilizing protocols and load-balancing algorithms for dynamic networks that have subsequently appeared in the literature.

We use our end-to-end protocol to derive a file-transfer protocol for sufficiently large files. The bit communication complexity of this protocol is $O(nD)$ bits, where D is the size in bits of the file. This file-transfer protocol yields an $O(n)$ *amortized* message complexity end-to-end protocol.

---

*Preliminary versions of the various results in this paper appeared in Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science 1989, and Proc. of the Eleventh Annual ACM Symp. on Principles of Distributed Computing, 1992 [AMS89, AGR92].

†Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel.

‡Computer Science Department, Johns Hopkins Univ. and Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139; Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

§Computer Science Department, U.C.L.A., CA 90024. Supported by NSF Presidential Young Investigator Award under grant DCR84-51396 & matching funds from XEROX Co. under grant W881111.

¶Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel. Part of the research was done while the author was at Laboratory for Computer science, MIT, partially supported by NSF 865727-CCR, ARO DALL03-86-K-017 and ISEF fellowship, and at IBM - T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598.

‖Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel.

**Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel. Part of this work was performed while this author was at the Hebrew University, Jerusalem and the *TDS* group at *MIT*. Supported by Israeli Communications Ministry Award, and by *NSF* contract no CCR-8611442, by *ONR* contract no N0014-85-K-0168, by *DARPA* contract no N00014-83-K-0125, and a special grant from *IBM*.

1 Introduction

A basic problem in computer networks is that of *end-to-end* communication, that is, the delivery in finite time of data items, generated at a designated *sender* processor to a designated *receiver* processor without duplication, omission, or reordering of the data-items. The data items could represent transactions of a stock exchange, files to be transferred, interactive remote processing messages, etc. In almost all cases, the sequence of data items is produced on-line and is not available at the beginning of the protocol's execution.

In a reliable network, where communication links never fail, end-to-end communication is easily performed by establishing a fixed communication path between the sender and the receiver, and sending all data items along this path. However, communication networks like ARPANET [MRR80] and DECNET [Wec80], have a dynamic topology, i.e., links may repeatedly fail and recover, making it impossible to rely on any single communication path.

The "classical" approach to the problem in dynamic networks is to construct a new communication path every time the previous path fails, purging any messages in transit on the old path. However, this approach is limited since its implementations (e.g., [Fin79, Gal76, AAG87, AS88, AAM89, AGH90]) require strong assumptions regarding the allowable patterns of link failures in the network. In some works [AAG87, AS88, AAM89], the assumption is that the whole network stabilizes for a period of time long enough to allow the construction of a path and the delivery of at least one data item over it. Moreover, as networks become larger and presumably topological changes occur more often, the above approach yields protocols that might grind to a halt. However, as noted in previous works [Vis83, AE86], the existence of an operational communication path is not a necessary condition for communicating between two processors. A necessary condition for communication is that even though there is never a point in time during which there is a path of operational links between sender and receiver, they are *eventually connected*: there is no cut of permanently failed links separating the sender from the receiver. Formally this means that there exists no partition of the network into two sets, one containing the sender and the other the receiver, such that from some time and on, no message can be delivered from any processor in one set to any processor in the other set.

Early papers [Vis83, AE86] solving the end-to-end communication problem under the eventual connectivity condition employed "unbounded sequence numbers", implying that both the message size and the amount of memory needed grow with the number of data items transmitted. Therefore, the space and communication complexity of those protocols is unbounded in terms of the size of the network. In recent years a sequence of works gave bounded and increasingly efficient solutions to the problem. The first bounded end-to-end communication protocol under the eventual connectivity fairness condition [AG88] required only $O(1)$ space per-link, but had an exponential communication complexity.

This paper presents the first polynomial complexity end-to-end communication protocol in dynamic networks. In the worst case the protocol sends $O(n^2m)$ messages per data item delivered, where n and m are respectively the total number of processors and links in the network. A preliminary version by three of the authors [AMS89] presented an algorithm with a message complexity of $O(n^9)$. Another preliminary end-to-end protocol by two of the authors, based on the resynchronization protocol [AG91] for unreliable networks, constructed an end-to-end communication protocol with an improved $O(nm)$ message complexity. Though the presented algorithm is a factor of n slower than [AG91], it is substantially simpler and more streamlined than either of the above, and unlike them can be used to yield an efficient file-transfer protocol. Furthermore, its key component is the novel *slide* protocol which we have reason to beleive will become a popular building block for dynamic network algorithms. In fact, a recent work [KOR95] builds upon the *slide* to obtain an end-to-end communication protocol with logarithmic space complexity, and at the same time polynomial communication complexity. In the table below we compare the performace of the various known end-to-end communication protocols.

Paper	Communication Complexity	Space Complexity
[Vis83, AE86]	**unbounded:** ∞	**unbounded:** ∞
[AG88], Alg. 1.	**unbounded:** ∞	**constant:** $O(D)$
[AG88], Alg. 2.	**exponential:** $O(D \cdot \text{exponential}(n))$	**logarithmic:** $O(\log n + D)$
present work	**polynomial:** $O(n^2 mD)$	**linear** : $O(nD)$
[AG91]	**polynomial:** $O(nm \log n + mD)$	**linear:** $O(n + D)$
[KOR95]	**polynomial:** $O(n^2 mD)$	**logarithmic:** $O(\log n + D)$

Slide is a simple and efficient method for delivering tokens across an unreliable network. Like the Merlin-Schweizer deadlock avoidance algorithm [MS80], it uses store-and-forward buffer hierarchies to control packet flow. However, the similarity ends here: *slide* allows packets the freedom to move in the network obliviously and permits deadlocks caused by individual packets that are delayed in the network for an indefinite periods of time. It uses the buffer hierarchy to balance the flow of packets, so that if enough packets of a given type are put into the network by a sender, some packets must reach the receiver processor.

We construct our first end-to-end communication protocol by combining *slide* with the majority selection mechanism of [AAF$^+$90]. We then present a second protocol without majority selection, which has the advantage of being data-oblivious, i.e. the protocol does not access the data being transmitted. This modular separation of messages into a "control bits" part and a "data" part is standard practice in communication protocols. A combination of key elements of the two protocols, in conjunction with the Information Dispersal Algorithm (IDA) of Rabin [Rab89], allows us to design a file-transfer protocol with $O(nD)$ bit communication complexity for files of sufficiently large size D.

The rest of the paper is organized as follows. The *slide* protocol is presented in subsection 4.1. Section 2 introduces the dynamic network model. Section 3 provides an informal overview of our protocols. Section 4 includes their formal statement, proof of correctness and analysis, specifically that of the slide protocol.

2 Model and Problem Statement

2.1 The Network Model

Consider a communication network in the form of an undirected graph $G = (V, E)$, $|V| = n$ $|E| = m$, where the nodes are the processors and the edges are the links of communication.

Processors are modeled as interactive Turing machines, and run message driven programs. We do not require that they have distinct identifiers, and in fact except for the sender and the receiver they all run the same program. Each *undirected* link consists of two *directed* links, delivering messages in the opposite directions. Below we describe the properties of a directed link. We associate with each message a *send event* and a *receive event*; each event has its time of occurrence according to a global time, unknown to the nodes. We assume no two events occur exactly at the same time. A message is said to be *in transit* at any time after its send event and before its receive event.

Each link has *constant* capacity, in the sense that only a constant number of messages can be in transit on a given link at a given time. For clarity of presentation we present the protocols in a model in which each link has $O(n)$ capacity (Lemma 4). However, The model of $O(n)$ capacity links is easily reduced to the model of constant capacity links by maintaining a buffer of $O(n)$ outstanding messages for each link. This reduction does not increase the space complexity of our protocols since their space complexity is $O(n)$ in either case. Each link delivers messages in FIFO order, that is, the sequence of messages *received* over it is a prefix of the sequence of messages sent over the link. Also, the communication is *asynchronous*: There is no a-priori bound on message transmission delays over the links.

A directed link is *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 a node v is *eventually connected* to a node u if there exists a (simple) path from v to u consisting entirely of undirected viable 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 become non-viable, then it becomes impossible to deliver messages from the sender to the receiver.

Note that we model the undirected graph as a by-connected directed graph. We thus assume that for each link either both its directed link are viable, or both are non-viable. In this case, the assumption stated above of eventual connectivity between the sender and the reciever is a necessary minimal condition to allow communication between the sender and the receiver. In the model of *directed* graphs, it could be the case that there exists a directed viable path from the sender to the receiver (and maybe a different one from the receiver to the sender), yet all undirected links are non-vaible. We do not consider in the present paper this (more difficult) model, and are dealing only with undirected graphs.

2.2 Other Models

The model described above is called the "∞-delay model" in [AG88], and the "fail-stop model" in [AM88]. As mentioned in the introduction, we deal with networks that frequently change their topology. In such *dynamic* networks, links may fail and recover many times (yet processors never fail) [AAG87], and each failure or recovery of a network link is eventually reported at both its endpoints by some underlying link protocol. It is not hard to see that any problem defined in the context of the dynamic-network model can be reduced to the same problem defined in the context of the fail-stop networks model. Given a network under the dynamic model, and an algorithm for networks of the fail-stop model, one can apply the given algorithm as follows: A message to be forwarded on a link is stored in a buffer, which is manipulated by a lower-level protocol that leaves the message in the buffer until all previous messages have been delivered, and until the link recovers, if it is down. A protocol similar to the data-link initialization protocol [BS88] is used to guarantee that no message is lost or duplicated. Any 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 in the fail-stop model; each link that eventually recovers for such a long enough period of time is represented by a viable link. Any two nodes that are eventually connected in the dynamic network model are eventually connected in the fail-stop model.

2.3 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 inserted from the environment to one distinguished processor, called the sender and usually denoted by S, to a second distinguished processor, called the receiver and usually denoted by R, that in turn will extract them to its environment. 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:

Safety: The sequence of data items output by the receiver 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.

An algorithm for the end-to-end communication problem generates a sequence of input events of data items at the sender and a sequence of receive events of data items at the receiver, that obey the safety and liveness properties.

2.4 The Complexity Measures

We consider the following complexity measures:

Message: The total number of messages sent in the worst case in the period of time between two successive data item output events at the receiver.

Communication: 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.

Space: The maximum amount of space per incident link, measured in bits, required by a node's program throughout the protocol.

Definition 1 *A protocol is* bounded *if its communication and space complexities are* independent *of the number of data items, depending only on the size of the network and the size of a data item.*

Definition 2 *A protocol is* polynomial *if its communication and space complexities are upper-bounded by* polynomials *of the size of the network.*

We would like to stress the fact that being able to send (receive) an infinite number of messages does not require either the sender or the receiver to have infinite space. A single buffer at the sender (receiver) suffices in order to store the next data item to be transmitted. The precise formulation of this "interactive" statement of the problem can be found in [LMF88].

3 Informal Description

In this section, we informally describe the *slide* protocol, and then describe three end-to-end communication protocols that use it as a building block. The formal presentation of these protocols, their proof of correctness and their analysis follow in the next section.

3.1 The *Slide* Protocol

The purpose of the *slide* protocol is to deliver messages from a sender to a receiver over an unreliable network. We refer to these message as *tokens*, since for the purpose of the *slide* protocol we are indifferent to the contents of the messages. In the *slide* protocol one designated processor, the sender, inputs tokens (messages) into the network. The sender can be in either of two states, *enabled*, or *disabled* and it may insert new tokens into the network only if it is enabled. A second designated processor, the receiver, outputs the tokens from the network. Tokens are neither lost nor duplicated in the network, and the total number of tokens in it at any given time is bounded. If the sender and the receiver are eventually connected, then eventually the sender is in the *enabled* state, that is, the insertion of a new token into the network is possible. The order in which the tokens are output by the receiver is, however, not necessarily that in which they were input by the sender. More formally, if the sender and the receiver are *eventually connected*, then the *slide* protocol establishes between them a non-FIFO, bounded-capacity virtual communication link that does not lose or duplicate messages.

The *slide* protocol is based on the storing and forwarding of tokens between the processors of the network. Each undirected link is viewed as a pair of directed anti-parallel links. Each processor maintains for each incident incoming link an array of slots numbered 1 through n. We regard the elements of the array as ordered in increasing order of *levels*. Each slot has room for one token, and each array is used to store tokens arriving on the link associated with it; tokens from an array can be sent over any outgoing link. The key to the protocol is the condition that a token be sent from any slot i at processor v to slot j at the (v, u) array at processor u, only if $j < i$. To this end, the processors maintain for each outgoing link a variable holding an upper bound on the lowest numbered slot available at the other side of the link. The tokens are sent from slots with a number higher than the bound, and thus are guaranteed to conform to the above condition. Every time a token is removed from an array, a signal to this effect is sent over the incoming link associated with the array. Since the only source of tokens for a specific array is the processor on the other end of its associated link, the bound can be maintained by incrementing it every time a token is sent over the link, and decrementing it every time a signal is received over the link. Thus the bound is never smaller than the number of tokens in the array on the other side of the link plus the number of tokens in transit over the link. As the links obey the FIFO rule, the above mentioned variable is at any time t an upper bound for the lowest numbered slot that is available in the receiving processor upon the arrival of a token that is sent at time t.

New tokens enter the network only at the sender and to a special slot at level n. The receiver has always a vacant slot of level 1, and removes and outputs any token it receives [1]. If the sender and the receiver are eventually connected, then eventually the special slot at the sender is vacant. The tokens travel in the network from the sender to the receiver, *sliding* from higher numbered slots to lower numbered slots as they advance from link to link. Therefore, each token can make at

[1] We remark that for every node but the receiver, slots of level 1 are redundant, as a token cannot be sent from such slots.

most n hops in the network. Since the protocol maintains for each link $2n$ slots, and (as we prove in the sequel) this also bounds the total number, per link, of tokens in slots plus tokens in transit at any given time, the total number of tokens in the network at any given time is at most $2nm$. This is the *capacity* of the *slide* protocol, denoted $\mathcal{C}$. In Lemma 4 we show that we can replace the assumption that link capacity is $O(n)$ by an assumption that link capacity is $O(1)$, by maintaining a $2n$ messages buffer of outstanding messages for each link.

3.2 The Majority Algorithm

We construct a simple end-to-end communication algorithm by operating the *slide* from the sender processor S to the receiver processor R. To send a data item to R, processor S sends consecutively $2{\cdot}\mathcal{C}+1$ duplicates of the data item to R using the *slide*. To output the first data item, R waits for $\mathcal{C}+1$ data items and outputs one of them, and for each subsequent data item R waits for the next $2{\cdot}\mathcal{C}+1$ data items, takes the majority of the values received, and outputs this value. This is similar to the protocol of [AAF+90]. Since S sends $2{\cdot}\mathcal{C}+1$ duplicates of each data item and the *slide* can delay only up to $\mathcal{C}$ data items, the receiver is ensured to receive enough data items to allow the output of the next data item.

3.3 The Labels Algorithm

In the *labels* algorithm, each data item is marked with a unique label, enabling the receiver to distinguish between a new data item that has yet to be output and an old item that has already been output. The protocol is thus data-oblivious in that it does not use the data itself for the control of the protocol. The labels are not "sequence numbers" since they need not define an order on the items. Since the *slide* protocol has a bounded token capacity, one can design an algorithm requiring only a bounded range of labels by devising a technique allowing the sender to know which labels it can reuse. We do so in the following way.

Given a designated sender (S) and receiver (R) of an end-to-end communication problem, we operate two *slide* protocols, one from S to R and another from R to S. The *slide* operated from R to S is used by R to return to S tokens it received.

Let $\mathcal{C}=O(nm)$ denote the maximum number of tokens that a single *slide* protocol can delay. Let $\mathcal{L}$ denote a set of $O(nm)$ labels, and at any point in time, let free_$\mathcal{L}$ be a variable holding the subset of $\mathcal{L}$ from which S can take a label to mark a new data item since the label does not appear in any token in the network. Initially, free_$\mathcal{L}=\mathcal{L}$.

Processor R keeps for each label an indicator saying whether R may accept a new data item with this label or not; initially, R may accept a data item with any label. Whenever S wishes to send a data item to R, it extracts a label l from free_$\mathcal{L}$ and starts sending tokens of the form $(l, data\_item)$ to R. S stops sending these tokens either when the first such token is received back from R, or after $\mathcal{C}+1$ such tokens are sent. Any token that arrives at R is returned to S using the R-to-S *slide* protocol. Before returning the token, R processes the token as follows: If the status of the label appearing in the token is `acceptable`, it outputs the received data item and sets the status of that label to `not acceptable`; otherwise (if the status of the label is `not acceptable`) it ignores the token since the received data item has already been output. Processor S counts, for each label, the number of tokens it sends and the number of tokens it receives back from R. If and when all the tokens containing a certain label arrive back, S can use the label again for transmitting future data items. Before doing so, S must inform R that it should again set the status of the label to `acceptable`. This is done by 'reset' messages sent to R. In order not to increase the complexity of the algorithm by adding the 'reset' messages and in order to avoid deadlocks, a 'reset' message is "piggy-backed" on the tokens sent to R. To this end, upon the receipt by S of the last token having label l, l is added to a set of 'pending reset' labels. To each data item sent, S adds a 'reset' message for a label from the 'pending reset' set (if the set is not empty). When R receives a token contains a 'reset' message for l, it sets l to the `acceptable` status. If and when S receives back all the tokens containing a 'reset' message for a certain label l, S concludes that l is in the `acceptable` state at R, and that the S-to-R *slide* is "clean" of tokens carrying either a data item labeled by l, or a reset message for l. Therefore, l can be safely returned to free_$\mathcal{L}$ for future use by S.

Since the capacity of each *slide* is bounded by $\mathcal{C}$, no more than $\mathcal{C}+1$ tokens have to be sent by S before at least one reaches R and the data item is output. The algorithm is technically designed so that to ensure that at any time the number of tokens stored in R (before being returned to S) is bounded. Together with the fact that each *slide* can delay up to $\mathcal{C}$ token, this implies that a set of $6 \cdot \mathcal{C}+3$ labels allows the algorithm to run without deadlocks (see Section 4.3).

3.4 The Data Dispersal Algorithm

We now show an algorithm that achieves $O(nD)$ bit communication complexity, for the cases in which the data items are large with respect to the size of the network (having size of $\Omega(nm \log n)$ bits). The same algorithm can also be used for smaller data items if the sender is allowed to lump together several data items and transmit them together.

Recall that the *slide* protocol allows only a finite number of packets to be delayed in the network. Based on this property we are able to combine the *slide* protocol with Rabin's Information Dispersal Algorithm [Rab89] to achieve the $O(nD)$ bit complexity. The general idea is that the sender splits the data item into packets using the Information Dispersal Algorithm (IDA) and sends them to the receiver using *slide*. As the IDA allows the construction of the full data item from only a subset of these packets, the protocol can tolerate the loss of the finite number of packets that can be delayed in the network during the execution of *slide*. In addition, the total size of the packets in any group from which the data item can be constructed is not larger than the size of the data item itself; therefore we build an efficient algorithm with $O(nD)$ bit communication complexity.

More specifically, the sender creates, using the IDA, $2 \cdot \mathcal{C}+1$ packets, each of size $O(\frac{D}{\mathcal{C}+1})$ bits, where D is the size of the data item. The sender sends each of these packets to the receiver, each one along with its serial number as required by the IDA. This allows the receiver to construct the full data item from only $\mathcal{C}+1$ packets. The sender sends the $2 \cdot \mathcal{C}+1$ packets and, since at most $\mathcal{C}$ packets can be delayed, the receiver will receive enough packets to reconstruct the data item. The only difficulty left is to make sure that the receiver does not use old delayed packets to reconstruct data items subsequently sent. To overcome this difficulty, the sender selects for each data item a label and adds it to all the packets of the data item. The receiver outputs the first data item after calculating it from the first $\mathcal{C}+1$ packets it receives; for each subsequent data item it waits for another $2 \cdot \mathcal{C}+1$ packets, checks which label has the majority among the labels in the packets, and uses only the packets having this label; this is similar to the Majority Algorithm. For each new data item the sender must use a label that is not present in the network. Therefore, as in the Labels Algorithm, the receiver sends back to the sender every packet it receives through another *slide* operated in the opposite direction. Thus the sender always knows which labels are present in the network. As the capacity of each *slide* is bounded by $\mathcal{C}$, $2 \cdot \mathcal{C}+1$ different labels suffice.

The bit communication complexity of the Data Dispersal Algorithm is $O(nD)$ bits per data item if it is applied to large enough data items. As each packet is sent with a serial number of size $O(\log mn) = O(\log n)$ bits, the size of a data item that yields this complexity should be $\Omega(nm \log n)$ bits. If the algorithm is applied to smaller data items, it achieves an *amortized* bit communication complexity of $O(nD)$ bits, by combining several data items together.

4 Formal Description and Proofs

In this section, we formally state the code of the *slide*, the Majority, the Labels and the Data Dispersal Algorithms, prove their correctness, and analyze their complexities. 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)).

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. $\mathcal{X}_v^t$).

4.1 The *Slide* Protocol

The protocol, given in Figure 1, uses two types of messages: TOKEN messages which are used to transfer the tokens themselves, and TOKEN_LEFT messages that are used as signals to inform the other side of a link that a token from the array associated with it was removed from the array.

Each node has associated with each incoming link an array of n slots ordered in levels from 1 to n. Each of these slots is used to store a single token arriving on the respective incoming link. In addition, each node maintains for each outgoing link a variable called *bound*, which is an upper bound on the number of tokens in the array on the other end of the link plus the number of tokens on the link plus 1. Thus, *bound* is an upper bound on the height of the slot available for a token if it is sent. This bound is maintained by initializing it to 1, incrementing it by 1 every time a token is sent over the outgoing link, and decrementing it by 1 every time a TOKEN_LEFT message is received over the corresponding incoming link. Whenever there is a token stored in a slot with a higher number than the *bound* of some outgoing link, the token is removed from the slot and sent over the link.

The differences between the sender and an ordinary node are due to the fact that the sender is the node that inputs new tokens to the network. Therefore it has an additional "special array" into which tokens are input from an external process. These tokens are input into slot number n of the "special array". Like all other arrays, tokens from this array can be sent over any link.

The receiver outputs any token it receives and never sends tokens.

```
Select
Initialization ⟶
    for every incident link e
                bound[e]:=1;
                top[e]:=0;
□
Receive TOKEN_LEFT on e ⟶
    bound[e]:=bound[e]-1;
□
Receive TOKEN(data) on e ⟶
    top[e]:=top[e]+1;
    slots[e][top[e]]:=data;
□
∃e, e' s.t. top[e'] > bound[e] ⟶
                /* e' not necessarily ≠ e */
    send TOKEN(slots[e'][top[e']]) on e;
    send TOKEN_LEFT on e';
    top[e']:=top[e']-1;
    bound[e]:=bound[e]+1;

End Select
```

a: ordinary node's code

```
Initialization ⟶
    input_array[n]:=vacant;
□
input_array[n]=vacant ⟶
    input_array[n]:= next input;
□
input_array[n]≠vacant
                and ∃e s.t. bound[e]< n ⟶
    send TOKEN(input_array[n]) on e;
    input_array[n]:=vacant;
    bound[e]:=bound[e]+1;
□
```

b: additions for the sender (sender code is a & b)

```
Select
Receive TOKEN(data) on e ⟶
    output(data);
    send TOKEN_LEFT on e;

End Select
```

c: receiver's code

Figure 1: The *slide*

4.1.1 Correctness Proof of the *slide* Protocol

In this section we prove that:

Theorem 4.1 *The* slide *protocol satisfies the following four properties:*

- $\mathcal{P}1$. *For each token, the total number of times it is sent over a link, is at most n. (we say that each time a token is sent it is* passed *over a link, and that it performs a* hop *in the network).*
- $\mathcal{P}2$. *At any time t, the number of tokens in the network is bounded by $2nm$. ($\mathcal{C} = 2nm$).*
- $\mathcal{P}3$. *In any time interval in which* `new` *new tokens are inserted into the network, at most $O(n^2m + \texttt{new} \cdot n)$ token-passes can occur.*
- $\mathcal{P}4$. *If the sender and the receiver are eventually connected, the sender will eventually input a new token.*

Proof: We start with several definitions. The definition are used to count the number of different messages on a given link at a given time.

Definition 3 *Let $\underline{tokens^t_{u \to v}}$ be the number of TOKENs in transit from u to v at time t. Let $\underline{signals^t_{u \to v}}$ be the number of TOKEN_LEFT messages in transit from u to v at time t.*

Lemma 1 *At any time t and for any $e = (u, v)$,*

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

Proof: Upon initialization, the invariant holds, since the $bound[e]$ variables are initialized to 1, the $top[e]$ variables 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 can show that it holds for any t. There are four 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 2 *If a token from slot i at node u is sent to node v and is stored there at slot 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. Denote by e the link between v and u, by *new_slot* the slot number in which the token is stored in v, and by *old_slot* the slot number where it was stored in u.

Because *top* is incremented only when tokens arrive on the link, and because the links are FIFO, we have:

$$top[e]^{t'}_v \leq top[e]^t_v + tokens^t_{u \to v}.$$

By Lemma 1,

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

By the code $old\_slot > bound[e]_u^t$ and $new\_slot = top[e]_v^{t'} + 1$, hence $old\_slot > new\_slot$. $\square$

Since new tokens enter the network into slot n, this proves property ($\mathcal{P}1$) of the *slide*.

Since all the tokens in the network are either stored in the arrays or in transit over links, the following lemma proves property ($\mathcal{P}2$).

Lemma 3 *At any time t and for any $e = (u, v)$,*

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

Proof: By Lemma 1 $top[e]_v^t + tokens_{u\to v}^t \leq bound[e]_u^t - 1$. For $bound[e]$ to be strictly greater than n, a token must be sent over e when $bound[e] = n$. By the code, this token must be stored in level $\geq n + 1$. By Lemma 2, and since new tokens enter the network into level n slots, such a token cannot exist. Thus for any t $bound[e]_u^t \leq n$. $\square$

We can now also prove properties ($\mathcal{P}3$) and ($\mathcal{P}4$). We start by proving property ($\mathcal{P}3$). By property ($\mathcal{P}2$) the total number of tokens in the network at the beginning of the time interval is $O(nm)$. By Property ($\mathcal{P}1$) each can make up to n hops in the network, thus contributing up to $O(n^2m)$ token passes. Any token from the `new` new tokens can also make up to n hops.

The rest of the proof is devoted for proving property ($\mathcal{P}4$). By way of contradiction assume that t is the last time at which the sender inputs a token.

As a result of property ($\mathcal{P}3$) and as there is only one TOKEN_LEFT message per token pass, there is a time $t' \geq t$ after which no TOKEN or TOKEN_LEFT messages are sent. As S and 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 v_i cannot have a token in a slot at level strictly greater than i after time t'' .

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 inductive hypothesis that v_{i-1} has no token stored at level strictly greater than $i - 1$. Since at t'' all messages between v_{i-1} and v_i have arrived, by Lemma 1 and the inductive assumption $bound[e]_{v_i}^{t''} \leq i$. As $t'' \geq t'$, no token is sent after t'', but according to the code this can happen only if v_i has no tokens in slots of level $i + 1$ or more, proving the induction step.

Thus slot n at S is vacant, and S will enable the input of a new token, contradicting the assumption. $\square$

The following lemma shows that our protocol applies in the model where links have constant capacity by having an $O(n)$ space buffer at the tail of each link and sending every message only after receiving an acknowledgment for the previous one. As the space complexity of the protocol is already $O(n)$ per link (see below), this change does not affect any of the complexity measures.

Lemma 4 *At any time t, there are at most $2n$ messages in transit in each direction on any link.*

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

$$tokens^t_{u \to v} \leq bound[e]^t_u - 1, \text{ and } signals^t_{v \to u} \leq bound[e]^t_u - 1 \ .$$

By the same arguments as in the proof of Lemma 3, $bound[e]^t_u \leq n$, for any t. Hence $tokens^t_{u \to v} \leq n$ and $signals^t_{v \to u} \leq n$.

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

4.1.2 The Complexity of the *Slide* Protocol

Lemma 5 *The number of messages sent by the* slide *protocol in any time interval where* `new` *new tokens are input by the sender is bounded by* $O(n^2 m + \texttt{new} \cdot n)$.

Proof: The only messages in the protocol are TOKEN messages and TOKEN_LEFT messages, and there is exactly one TOKEN_LEFT message per TOKEN message. The lemma thus follows from Property $(\mathcal{P}3)$. □

Corollary 6 (Communication Complexity) *The number of bits sent by the* slide *protocol in any time interval where* `new` *new tokens are input by the sender is bounded by* $O((n^2 m + \texttt{new} \cdot n)D)$, *where D is the maximal number of bits in a token.*

The following claim follows from the code of the protocol and its correctness.

Claim 7 (Space complexity) *The space required at each node is $nD + 2\log n$ bits per incident link, where D is the maximal number of bits in a token (if links have constant capacity then it is $2nD + n + 4\log n$).*

4.2 The Majority Algorithm

The algorithm is informally described in Section 3.2, and its code is given in Figures 2 and 3.

The Majority Algorithm uses the *slide* protocol, given in Section 4.1, as a lower-level building block. The sender and the receiver of the Majority Algorithm communicate using this protocol: each token to be sent by the sender of the Majority Algorithm is `input` by the sender of the *slide*, and upon the arrival of a token to the receiver of the *slide* it is `output` by this receiver and *received* by the receiver of the Majority Algorithm.

4.2.1 Correctness Proof of the Majority Algorithm

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

Theorem 4.2 (Safety) *At any time the output of the receiver is a prefix of the input of the sender.*

```
Select
Initialize ⟶
    items-set:=φ;
    first_item:=true;
□
Receive(data-item) ⟶
    items-set:=items-set ∪ {data-item};
    call check_and_output;

End Select
```

a: receiver's code

```
Select
true ⟶
    data-item:=next input;
    for i:=1 to 2·C + 1 do
            Send(data-item);
    od

End Select
```

b: sender's code

Figure 2: The Majority Algorithm

```
Procedure check_and_output
    if first_item and |items-set| = C + 1 then
                    /* first data item */
        output(any_data_item_of(items-set));
        items_set:=φ;
        first_item:=false;
    else if (not first_item) and |items-set| = 2·C + 1 then
                    /* all other data items */
            output(majority(items-set));
            items_set:=φ;
        endif
    endif
```

c: procedure check_and_output

Figure 3: The Majority Algorithm

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 the majority of the 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 token that carries I_k, $k > i$ could have been received before t_i.

The following definitions are used to count the number of tokens in the system.

Definition 4 *Let* $\underline{in^{(t,t']}}$ *be the number of tokens input by the sender to the* slide *in interval of time* $(t, t']$. *Let* $\underline{out^{(t,t']}}$ *be the number of tokens received by the receiver from the* slide *in the interval of time* $(t, t']$.

Denote by t_0 some time before the beginning of the execution of the algorithm.

Definition 5 $delay^t = in^{(t_0,t]} - out^{(t_0,t]}$ *(the number of tokens delayed by the* slide *at time* t*).*

By the code, the total number of tokens that have been received by the receiver by time t_i is:

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

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,

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

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, $t < t_i$.

We claim that no more than $\mathcal{C}$ tokens containing data item I_k, $k < i$ may be received in the interval of time $(t_{i-1}, t_i]$. This, together with the fact that no token carrying I_k, $k > i$ can arrive at time $t < t_i$, completes the proof of the safety property because it implies that of the $2 \cdot \mathcal{C} + 1$ tokens received in $(t_{i-1}, t_i]$ at least $\mathcal{C} + 1$ carry data item I_i.

To prove the claim, we distinguish between two sets of tokens, those that carry data items I_k, $k < i$, which we call *old*, and all other tokens. We have already proved that all the tokens received by t_{i-1} are *old* and that the total number of such tokens received by the receiver by t_{i-1} is $(2 \cdot \mathcal{C} + 1)(i-1) - \mathcal{C}$. Since the total number of *old* tokens ever sent by the sender is $(2 \cdot \mathcal{C} + 1)(i-1)$, at most $\mathcal{C}$ may be received by the receiver in the interval of time $(t_{i-1}, t_i]$. □

Theorem 4.3 (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 ($\mathcal{P}4$) of the *slide* all the tokens are eventually input by the *slide*. Since the *slide* 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. □

4.2.2 The Complexity of the Majority Algorithm

Lemma 8 *The message complexity of the majority algorithm is $O(n^2m)$ messages .*

Proof: Clearly in $(t_{i-1}, t_i]$ the receiver receives $2 \cdot \mathcal{C} + 1$ tokens. Since the *slide* 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 5. □

Since every bit in this algorithm is duplicated $O(n^2m)$ times, we establish the following corollary.

Corollary 9 (Communication Complexity) *The bit communication complexity of the majority algorithm is $O(n^2mD)$ bits, where D is the size in bits of a data item.*

Lemma 10 (Space Complexity) *The space complexity of any node except the receiver is $O(nD)$ bits and $O(nmD)$ bits for the receiver, where D is the size in bits of a data item.*

Proof: Each token sent in the Majority Algorithm consists of D bits. Combining that with the space complexity of the *slide* results in space complexity of $O(nD)$ for the Majority Algorithm for any node except the receiver. The receiver requires in addition $O(nmD)$ bits. □

4.3 The Labels Algorithm

The algorithm is informally described in Section 3.3, and the code of the algorithm is given in Figure 4. In the algorithm we use two *slide* protocols between the sender and the receiver, operating in opposing directions. In the code we use the subscripts $S \rightarrow R$ and $R \rightarrow S$ to denote operation with respect to the *slide* from the sender to the receiver and the *slide* from the receiver to the sender, respectively. Similarly to the Majority Algorithm, the *slide* is a lower-level building block used by the Labels Algorithm. Tokens to be *sent* by the Labels Algorithm are `input` by the sender of the corresponding *slide* protocol, and upon their arrival to the corresponding receiver, they are `output` by it, and *received* by the process of the Labels Algorithm.

Each token sent from S to R consists of three fields: a label, marking the token; a data item; and a piggy-backed reset-label. The set $\mathcal{L}$ is a set of $6{\cdot}\mathcal{C}+3$ labels, where $\mathcal{C}$ is the capacity of a single *slide*. Each token received by R is stored in a buffer before being returned to R. As the two *slide* pprotocols may operate at different paces, many tokens may be stored in the buffer. Therefore, we use at S a variable *missing* that counts the number of tokens that were sent but not returned yet. By delaying the input of a new data item until $missing \leq 2 \cdot \mathcal{C}$, we can limit the number of tokens stored at R. The array *count* counts for each label l how many tokens labeled by l are currently in the network. The function `extract`(*set*) extracts an arbitrary element from *set*. If *set* is empty the function returns null.

4.3.1 Correctness Proof of the Labels Algorithm

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

The 'life-cycle' of each label, as viewed by the sender, consists of four periods of time. First the label is in free_$\mathcal{L}$, second it is removed from free_$\mathcal{L}$ to label tokens in the network, third it is pending

```
Select
Initialization ⟶
   labels_to_reset:=φ;
   free_L:=L;
   sending:=false;
   missing:=0;
□
sending=false and missing ≤ 2·C ⟶
   data-item:=next input;
   current_label:=extract(free_L);
   current_reset_label:=extract(labels_to_reset);
   count[current_label]:=0;
   sending:=true;
□
sending=true ⟶
   Send_{S→R} (current_label,data-item,current_reset_label);
   count[current_label]:=count[current_label]+1;
   missing:=missing+1;
   if (count[current_label]= C + 1) then
             sending:=false;
   endif
□
Receive_{R→S} (l,⊥,reset_label) ⟶
   if (l=current_label) then sending:=false; endif
   missing:=missing-1;
   count[l]:=count[l]-1;
   if (count[l]=0) then
             labels_to_reset:=labels_to_reset ∪ {l};
             free_L := free_L ∪ {reset_label};
   endif

End Select
```

a: sender's code

```
Select
Initialization ⟶
   send_buffer:=φ;
   ∀ l ∈ L status[l]:=acceptable;
□
Receive_{S→R} (l,data-item,l-reset) ⟶
   if (status[l]=acceptable ) then
             output(data-item);
             status[l]:=not acceptable ;
   endif
   status[l-reset]:=acceptable ;
   send_buffer:=send_buffer∪ {(l ,⊥,l-reset)};
□
send_buffer ≠ φ ⟶
   (l,⊥,l-reset):=extract(send_buffer);
   send_{R→S}(l,⊥,l-reset);

End Select
```

b: receiver's code

Figure 4: The Labels Algorithm

reset, and then it is piggy-backed to tokens in order to be reset at the receiver. After all tokens resetting a label return to S, the label is returned to free_$\mathcal{L}$ to start a new 'life-cycle'. We define subsets of the labels, corresponding to the sets of labels that are in each of the above mentioned periods in the 'life-cycle' of a label.

Definition 6 *Let $\underline{sending^t}$ be the set of labels that at time t are used to label tokens that are either delayed by any of the two* slide *protocols or are in the receiver's send_buffer. Let $\underline{pending\_reset^t}$ be the set of labels that at time t are in the set* labels_to_reset *of the sender. Let $\underline{resetting^t}$ be the set of labels that at time t are piggy-backed on tokens that are either by any of the two* slide *protocols or in the receiver's send_buffer.*

Claim 11 *At the sender, at any time t, $missing^t \leq 3{\cdot}\mathcal{C}$.*

Proof: The variable *missing* is incremented when a token is sent by the sender. By the code, at most $\mathcal{C}+1$ tokens are sent between any two input events at the sender. The input event at the sender can occur only when $missing \leq 2{\cdot}\mathcal{C}$. Therefore, for any time t $missing \leq 3{\cdot}\mathcal{C}+1$. □

Note that this implies that, at any time *send_buffer* at the receiver stores at most $3{\cdot}\mathcal{C}+1$ tokens.

Lemma 12 *At any time t,*

1. $|sending^t| \leq 3{\cdot}\mathcal{C}+1$ *and* $|resetting^t| \leq 3{\cdot}\mathcal{C}+1$.
2. $|sending^t| + |pending\_reset^t| \leq 3{\cdot}\mathcal{C}+1$.

Proof: Part 1 follows immediately from Claim 11. To prove part 2, note that each time a label is added to *sending*, either *pending_reset* is empty, or a label is extracted from it. Therefore part 2 follows from part 1. To formally prove it, assume by way of contradiction that t_0 is the earliest time when $|sending| + |pending\_reset| > 3{\cdot}\mathcal{C}+1$. This means that, at t_0, a label was added either to *sending* or to *pending_reset*.

By the code, a label is added to *pending_reset* if and only if at t_0 the last token containing the label at the 'sending' field arrived at S, which means that the label is extracted at the same time from *sending*, contradicting the assumption that t_0 is the earliest time $|sending| + |pending\_reset| > 3{\cdot}\mathcal{C}+1$.

If the label is added to *sending*, then by the code, if at this time *pending_reset* is not empty, a label to be reset is sent with the tokens and this label is extracted from *pending_reset*, contradicting the assumption that at t_0 the sum of the cardinalities increases.

Thus, *pending_reset* must be empty at t_0, therefore $|sending| + |pending\_reset| > 3{\cdot}\mathcal{C}+1$ yields $|sending| > 3{\cdot}\mathcal{C}+1$, contradicting part 1. □

Lemma 13 *If $|\mathcal{L}| \geq 6{\cdot}\mathcal{C}+3$, then free_$\mathcal{L}$ is never empty.(i.e., the sender will always have a label to send with a new data item).*

Proof: The lemma follows Lemma 12. □

Theorem 4.4 (Liveness) *If the sender and the receiver are eventually connected, then any data item input by the sender is eventually output by the receiver.*

Proof: Let us first prove the following two lemmas:

Lemma 14 *If the sender and the receiver are eventually connected, then there is no deadlock at the sender (i.e. eventually, missing* $\leq 2{\cdot}\mathcal{C}$*).*

Proof: Assume by way of contradiction that there is a time t such that for any $t' > t$, $missing \geq 2{\cdot}\mathcal{C}+1$. By the code, the sender can send after t at most $\mathcal{C}+1$ tokens; therefore there is a time t'', after which the sender does not send any more tokens. Assume the sender has sent until t'' k tokens (counted over the whole run). As the *slide* can delay only up to $\mathcal{C}$ tokens, the receiver has received by t'' at least $k - \mathcal{C}$ tokens. All these tokens are added to *send_buffer*. Since the sender and the receiver are eventually connected by property ($\mathcal{P}4$) of the *slide*all these tokens are eventually input by the R-to-S *slide*. As this *slide* can delay at most $\mathcal{C}$ tokens as well, the sender will eventually receive at least $k - 2{\cdot}\mathcal{C}$ tokens. Therefore $missing$ will eventually be $\leq 2{\cdot}\mathcal{C}$. □

This implies that any data item available for input will eventually be input by the sender. Clearly, at least one token with a copy of each data item is received by the receiver. Thus it remains to prove that one copy of each data item will be output. For this, we need the following.

Lemma 15 *Let* $acceptable^t$ *be the set of labels whose state is* `acceptable` *at time* t *in* R*. Then at any time* t*, free_*$\mathcal{L} \subseteq acceptable^t$*.*

Proof: Clearly the invariant holds when the algorithm starts.

A label l is extracted from *acceptable* only when R receives a token with l at the 'labeling' field. Since at this time there is no token in the network labeled with l, l cannot be in free_$\mathcal{L}$.

A label l is added to free_$\mathcal{L}$ only when all tokens with the label at the 'reset' field return to S. Assume this happens at time t. Since these tokens return to S, they were received by R, setting l to the `acceptable` status. Assume the last one was received by R at t', $t' \leq t$. But in the time interval $(t', t]$ there is no token with l at the 'labeling' field in the *slide*to R. Therefore at t l is in the `acceptable` status in R. □

Thus, the label l used by the sender with a new data item at time t is in the acceptable status at time t at the receiver. Furthermore, at t there is no other token in the network with label l in it. Thus, when the first copy of a token with label l, after time t, arrives at the receiver, the receiver outputs the new data item. □

Theorem 4.5 (Safety) *At any time the output of the receiver is a prefix of the input of the sender.*

Proof: The liveness property implies that every data item that is input at the sender is eventually output at the receiver. Next we claim that there is no duplication in the sequence of data items output by the receiver. This claim is proved by way of contradiction. Assume that data item I_i is output twice by the receiver at times t_1 and t_2. Thus at both times the receiver received a token of the form (l, I_i, l') and status$[l]$ was `acceptable`. Since at t_1 $status[l]$ is set to `not acceptable` this implies that at some time t', $t_1 < t' < t_2$, a token of the form $(*, *, l)$ is received by R. At time t no such tokens exist in the network (since l is extracted from free_$\mathcal{L}$) and any new such tokens can be created by the sender only after all tokens of the form $(l, I_i, *)$ have arrived to S. Therefore such token are created only after t_2, contradicting the fact that such token arrives at R at t'.

It remains to show that there is no reordering in the output sequence. This follows from the fact that the sender sends the $(i+1)$'st data item only after the i'th data item has been output by the receiver. □

4.3.2 The Complexity of the Labels Algorithm

Lemma 16 *The message complexity of the labels algorithm is $O(n^2 m)$.*

This lemma follows from the properties of the *slide* and from the bounded number of tokens input into each of the two *slide* protocols used, as proved in the following two lemmas.

Lemma 17 *In any time interval between two consecutive output events of the receiver at most $O(nm)$ tokens are input to the S-to-R* slide.

Proof: The lemma follows from the following two facts: (1) the maximum number of tokens that can be input to the *slide* between any two consecutive inputs events is bounded by $\mathcal{C} + 1$, and (2) any interval between two consecutive output events may overlap in time at most two intervals between consecutive input events. □

Lemma 18 *In any time interval between two consecutive output events of the receiver, at most $O(nm)$ tokens are input to the R-to-S* slide.

Proof: Denote by OUT_i and OUT_{i+1} the two output events that form the time interval. The tokens that can be input to the R-to-S *slide* in this time interval are the tokens in *send_buffer* at OUT_i, the tokens that are delayed by the S-to-R *slide* at OUT_i, and the tokens sent by the sender in the time interval $(OUT_i, OUT_{i+1}]$.

By Claim 11 the number of tokens in *send_buffer* is at most $O(nm)$. *Slide* can delay also at most $O(nm)$ tokens at any time. By the previous lemma the sender sends in this time interval at most $O(nm)$ tokens. □

Each token sent in the Labels algorithm consists of a data item plus a label of size $O(\log n)$ bits. We thus get the following corollaries.

Corollary 19 (Communication Complexity) *The bit communication complexity of the labels algorithm is $O(n^2 m(D + \log n))$, where D is the number of bits in a data item.*

Lemma 20 (Space Complexity) *The space complexity of any node except the receiver and the sender is $O(nD + n\log n)$, where D is the size in bits of a data item.*

Proof: Each token sent in the Labels Algorithm consists of $O(D + \log n)$ bits. Combining that with the space complexity of the *slide*, results in space complexity of $O(nD + n\log n)$ for the Labels Algorithm. □

Note that the space complexity of the sender and the receiver is $O(nm\log n + D)$.

4.4 The Data Dispersal Algorithm

The algorithm is informally described in Section 3.4, and the code of the algorithm is given in Figure 5. As in the Labels Algorithm, we use two separate *slide* protocols, one from the sender to the receiver and another in the opposite direction. We use the subscripts $S \to R$ and $R \to S$

```
Select
Initialization ⟶
    free_ℒ := ℒ;
    sending:=false;
    missing:=0;
□
sending=false and missing ≤ 2·𝒞 ⟶
    data-item:=next input;
    l:=extract(free_ℒ);
    using the IDA with parameters
          𝒞 + 1 and 2·𝒞 + 1,
          create packets 1 to 2·𝒞 + 1;
    send_buffer:=⋃_{i=1}^{2𝒞+1} {(l,i,packet_i)};
    count[l]:=0;
    sending:=true;
□
sending=true ⟶
    (l,i,packet):=extract(send_buffer);
    Send_{S→R} (l,i,packet);
    count[l]:=count[l]+1;
    missing:=missing+1;
    if |send_buffer|=0 then sending:=false ; endif
□
Receive_{R→S} (l,i,packet) ⟶
    missing:=missing-1;
    count[l]:=count[l]-1;
    if (count[l]=0) then free_ℒ= free_ℒ ∪ {l}; endif

End Select
```

a: sender's code

```
Select
Initialization ⟶
    packets-set:=ϕ;
    first_item:=true;
    packets-to-return:=ϕ;
□
Receive_{S→R}(l,i,packet) ⟶
    packets-set:=packets-set ∪ {(l,i,packet)} ;
    call check_and_output;
□
packets-to-return ≠ ϕ ⟶
    (l,i,packet):=extract(packets-to-return);
    Send_{R→S} (l,i,packet);

End Select
```

b: receiver's code

```
Procedure check_and_output
    if first_item=true and
        |packets-set|=𝒞 + 1 then
                                   /* first data item */
        using the IDA calculate the data item from the
          𝒞 + 1 packets in packets-set;
        output(data-item);
        packets-to-return:=packets-to-return ∪ packets-set;
        packets-set:=ϕ;
        first_item:=false;
    else if first_item=false and
        |packets-set|=2·𝒞 + 1 then
                                   /* all other data items */
          majority-label:=majority-of-labels(packets-set);
          using the IDA calculate the data item
                    from the packets in packets-set
                    having the label 'majority-label';
          output(data-item);
          packets-to-return:=packets-to-return ∪ packets-set;
          packets-set:=ϕ;
        endif
    endif
```

c: procedure check_and_output

Figure 5: The Data Dispersal Algorithm

the same way as for the Labels Algorithm. The interaction between the *slide* protocols and the processes of the present algorithm is the same as the interaction stated for the Labels Algorithm.

Let $\mathcal{L}$ denotes a set of $2{\cdot}\mathcal{C}+1$ labels . The sender maintains for each label $l \in \mathcal{L}$ a counter, $count[l]$, that holds at any time the number of tokens labeled l that are present in the network. The sender can, therefore, conclude at any time which labels are present in the network. The function $\texttt{extract}(set)$ extracts an arbitrary element from *set*.

Rabin's Information Dispersal Algorithm requires that the data be represented as a sequence of numbers over the field $\mathcal{Z}_p$, where p is a prime bigger than the number of packets to be created by the IDA. We use the IDA to create $2{\cdot}\mathcal{C}+1$ packets; therefore we need a prime p, such that $p > 4nm+1$. Since $m \leq n^2$, any p such that $p > 4n^3+1$ would do. In order to keep the size of the smallest data item to which the Data Dispersal Algorithm can be applied as small as possible, we should use the smallest p for which the above inequality holds. Since for any x there is a prime q such that $x \leq q \leq 2x$, there is always a prime that can be represented in $\lceil \log(8n^3+2) \rceil$ bits. Since each packet must contain at least one full number over $\mathcal{Z}_p$, the size of the smallest data item to which the Data Dispersal Algorithm can be applied is $\Omega(nm \log n)$.

4.4.1 Correctness Proof of the Data Dispersal Algorithm

In this section we prove the Safety and Liveness properties of the Data Dispersal Algorithm.

Theorem 4.6 (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. Let t_i be the time when the receiver outputs O_i, and denote by l_i the label added to the $2{\cdot}\mathcal{C}+1$ packets calculated by the IDA from I_i at the sender. By the code, the tokens used at t_i to calculate O_i at the receiver are the $2{\cdot}\mathcal{C}+1$ tokens received by it in the time interval $(t_{i-1}, t_i]$. By the same arguments as in the proof of Theorem 4.2, at least $\mathcal{C}+1$ of these tokens contain the label l_i; thus the majority of labels will be l_i, and the receiver will calculate O_i from the tokens containing l_i. Since at the time the sender extracts l_i from free_$\mathcal{L}$, there is no token containing it in the network, the receiver will use at t_i only packets calculated from I_i at the sender. As noted before, the receiver has at least $\mathcal{C}+1$ such packets at t_i, and the IDA will correctly calculate I_i at t_i. Thus $O_i = I_i$ for any i. □

The proof of the Liveness property requires the following technical lemma.

Lemma 21 *For any time t, missing $\leq 4{\cdot}\mathcal{C}+1$.*

The proof is similar to the proof of Claim 11.

Note that this also implies that the receiver never stores more than $4{\cdot}\mathcal{C}+1$ tokens in all its buffers.

Theorem 4.7 (Liveness) *If the sender and the receiver are eventually connected, then the receiver will eventually output any data item input by the sender.*

The proof is similar to the proof of Theorem 4.3.

4.4.2 The Complexity of the Data Dispersal Algorithm

Lemma 22 *The message complexity of the data dispersal algorithm is $O(n^2m)$ messages.*

Proof: Denote by t_i the time the i'th data item is output at the receiver. We use for the S-to-R *slide* the same notation as in Section 4.2.1. $out^{(t_i,t_{i+1}]} = 2{\cdot}\mathcal{C}+1$ and for any t, $0 \leq delay^t \leq \mathcal{C}$, thus $\mathcal{C}+1 \leq in^{(t_i,t_{i+1}]} \leq 3{\cdot}\mathcal{C}+1$. By Property ($\mathcal{P}2$) of the *slide* $\mathcal{C} = O(nm)$ and applying this to Lemma 5 yields a message complexity of $O(n^2m)$ for the S-to-R *slide*.

The tokens that are input to R-to-S *slide* in the time interval $(t_i, t_{i+1}]$ must be in $tokens\_to\_return$ just after t_i, since new tokens are added to this set only at output events at the receiver. By Lemma 21, the receiver stores at any time at most $4{\cdot}\mathcal{C}+1$ tokens. In the worst case all of them are in $tokens\_to\_return$ at t_i. Thus at most $4{\cdot}\mathcal{C}+1$ tokens are input to the R-to-S *slide* in the time interval $(t_i, t_{i+1}]$. Applying this to Property ($\mathcal{P}2$) of the *slide* and the results of Lemma 5, we obtain a message complexity of $O(n^2m)$ for this *slide*.

Combining the two *slide* protocols results in a message complexity of $O(n^2m)$ for the Data Dispersal Algorithm. □

Lemma 23 (Communication Complexity) *The bit communication complexity of the data dispersal algorithm is $O(nD)$ bits, where D is the size in bits of a data item, if $D = \Omega(nm\log n)$.*

Proof: Each token sent in the Data Dispersal Algorithm consists of a packet of size $O(\frac{D}{\mathcal{C}+1})$ bits, a label of size $O(\log n)$ bits, and a serial number of size $O(\log n)$ bits. The message complexity of the algorithm is $O(n^2m)$, and half of the messages are of size $O(\frac{D}{\mathcal{C}+1}+\log n)$ bits, while the other half have constant size. The total number of bits sent between any two consecutive output events at the receiver is, therefore, $O((n^2m)(\frac{D}{\mathcal{C}+1}+\log n)) = O(nD + n^2m\log n)$. For data items of size $\Omega(nm\log n)$ the bit complexity is thus $O(nD)$. □

Lemma 24 (Space Complexity) *The space complexity of any node except the receiver and the sender is $O(\frac{D}{m}+n\log n)$, where D is the size in bits of a data item.*

Proof: Each token sent in the algorithm is of size $O(\frac{D}{\mathcal{C}+1}+\log n)$. Combining that with the space complexity of the *slide*, results in space complexity of $O(n(\frac{D}{\mathcal{C}+1}+\log n))$. Since $\mathcal{C} = 2nm$, the space complexity is $O(\frac{D}{m}+n\log n)$. □

Note that using the analysis of the IDA [Rab89] the space complexity of the sender and the receiver is $O(n^2m^2\log n)$. [2]

5 Conclusion

This paper introduces the *slide* protocol and uses it to provide the first polynomial complexity end-to-end communication protocol in dynamic networks. Since its initial publication [AGR92], *slide* has been used as the basis for several new algorithms, including the elegant self-stabilizing protocols [AV91, APSV91, Var92], a load-balancing scheme [AAMR93], and a multi-commodity flow algorithms [AL94, AL93]. We believe it will find further applications in network protocol design as issues of availability and fault-tolerance become more critical in distributed applications.

[2]It was pointed out to us by Michael Saks [Sak91] that based on the *slide* and the majority mechanisms, for files of size at least $\Omega(n^2m^2\log n)$ bits, one can build an $O(nD)$ communication complexity file-transfer protocol without resorting to coding techniques such as the IDA.

6 Acknowledgments

We thank Michael Merritt and Mike Saks for their many helpful comments.

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 channels. *Journal of the ACM*, 41(6):1267–1297, 1994.

[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.

[AAMR93] W. Aiello, B. Awerbuch, B. Maggs, and S. Rao. Approximate load balancing on dynamic and asynchronous networks. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 632–641. ACM, May 1993.

[AE86] B. Awerbuch and S. Even. Reliable broadcast protocols in unreliable networks. *NETWORKS*, 16:381–396, 1986.

[AG88] Y. Afek and E. Gafni. End-to-end communication in unreliabel 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.

[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. Rosen. The slide mechanism with applications in dynamic networks. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 35–46, August 1992.

[AL93] B. Awerbuch and T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *Proc. 34th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 459–469, 1993.

[AL94] B. Awerbuch and T. Leighton. Improved approximation algorithms for multicommodity flow problem and local competative routing in dynamic networks. In *Proc. 26th ACM Symposium on Theory of Computing (STOC)*, 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.

[APSV91] 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.

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

[Fin79] S. G. Finn. Resynch procedures and fail-safe network protocol. *IEEE Trans. on Comm.*, COM-27:840–845, June 1979.

[Gal76] R. G. Gallager. A shortest path routing algorithm with automatic resynch. Unpublished note, March 1976.

[KOR95] E. Kushilevitz, R. Ostrovsky, A. Rosén, Log-Space Polynomial End-to-End Communication. In *Proc. of the 27th Ann. ACM Symposium on the Theory of Computing* (STOC), pages 559–568, May 199.

[LMF88] N. Lynch, Y. Mansour, and A. Fekete. The data link layer: Two impossibility results. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, August 1988.

[MRR80] J. M. McQuillan, I. Richer, and E. C. Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Communication*, COM-28(5), May 1980.

[MS80] P. M. Merlin and P. J. Schweitzer. Deadlock avoidance in store-and-forward networks 1: Store-and-forward deadlock. *IEEE Transaction on Communications*, 28(3):345–354, March 1980.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[Sak91] M. Saks. personal communication. 1991.

[Var92] G. Varghese. *Dealing with Failure in Distributed Systems.* PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 1992.

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

[Wec80] S. Wecker. DNA: the digital network architecture. *IEEE Trans. on Comm.*, COM-28:510–526, April 1980.