The Slide Mechanism with Applications in Dynamic Networks

(Extended Abstract)

Yehuda Afek* Eli Gafni† Adi Rosén‡

Abstract

This paper presents a simple and efficient building block, called *slide*, for constructing communication protocols in dynamic networks whose topology frequently changes. We employ *slide* to derive (1) an end-to-end communication protocol with optimal *amortized* message complexity, and (2) a general method to efficiently and systematically combine dynamic and static algorithms. (Dynamic algorithms are designed for dynamic networks, and static algorithms work in networks with stable topology.)

The new end-to-end communication protocol has amortized message communication complexity $O(n)$ (assuming that the sender is allowed to gather enough data items before transmitting them to the receiver), where n is the total number of nodes in the network (the previous best bound was $O(m)$, where m is the total number of links in the network). This protocol also has bit communication complexity $O(nD)$, where D is the data item size in bits (assuming data items are large enough; i.e., for $D = \Omega(nm \log n)$). In addition we give, as a byproduct, an end-to-end communication protocol using $O(n^2m)$ messages per data item, which is considerably simpler than other protocols known to us (the best known end-to-end protocol has message complexity $O(nm)$[AG91]). The protocols above combine in an interesting way several ideas: the *information dispersal algorithm* of Rabin [Rab89], the *majority* insight of [AFWZ88, AAF+], and the *slide* protocol.

The second application of *slide* develops a systematic mechanism to combine a dynamic algorithm with a static algorithm for the same problem, such that the combined algorithm automatically adjusts its communication complexity to the network conditions. That is, the combined algorithm solves the problem in a dynamic network, and if the network stabilizes for a long enough period of time then the algorithm's communication complexity matches that of the static algorithm. This approach has been first introduced in [AM88] in the context of topology update algorithms.

1 Introduction

One of the most important tasks of distributed algorithm designers is to construct *simple* and efficient building blocks for various network models. While simple and efficient building blocks for synchronous and asynchronous static networks have been presented [Awe85, AAG87, CL85, BGP89, AP90, DS80], only complicated, though theoretically efficient, algorithms are known for dynamic networks whose topology frequently change (like the ARPANET ([MRR80]) and DECNET ([Wec80])).

In this paper we present a simple and efficient building block, called *slide*, for dynamic networks with frequently changing topologies (i.e. for the ∞-delay model [AG88]). Essentially, *slide* establishes a non-fifo virtual communication link between two arbitrary nodes in the network. Its effectiveness is demonstrated in three applications:

- An end-to-end protocol that is considerably simpler then any previous known protocol and whose message communication complexity is $O(n^2m)$ (compared to the best known $O(nm)$ messages),
- An $O(n)$ *amortized* message complexity end-to-end protocol, assuming that the sender is allowed to gather enough data items before transmitting them to the receiver, and
- A mechanism that senses topological changes and controls the amortized (per topological change)

*Computer Science Department, Tel-Aviv University, Tel-Aviv 69978, Israel, and AT&T Bell Laboratories.

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

‡Computer Science Department, Tel-Aviv University, Tel-Aviv 69978, Israel.

communication complexity of any dynamic algorithm. If a static algorithm is run in parallel with a dynamic algorithm controlled by the mechanism the resulting algorithm adjusts its communication complexity to the network conditions.

In addition to these applications, the *slide* protocol has already proved its versatility and applicability in [APSV91] where it was shown that both it and its first two applications above (with minor changes), are self-stabilizing (as a result the third application also preserves the self-stabilization property).

The end-to-end protocols presented in this paper are data non-oblivious. The transport protocol uses the data. The first application, the *majority algorithm*, employs the majority idea introduced in [AFWZ88, AAF$^+$], therefore using the contents of the delivered data-items to make its decisions. Our second application, the *data dispersal algorithm*, breaks each large data item into many packets (using the information dispersal algorithm of Rabin [Rab89]) and achieves an $O(n)$ communication complexity (assuming large data items, $\Omega(nm \log n)$ bits). This is in contrast to other known end-to-end communication algorithms, which are all data-oblivious, and require that each data item as a whole traverses each link in the network, resulting in $O(m)$ communication complexity. Our solution does not yield $O(n)$ communication complexity algorithms for smaller data items, since it breaks each data item into $O(nm)$ pieces.

These algorithms designed to tolerate faults have high communication complexity. They incur this complexity even when the network is stable (the obvious end-to-end communication algorithm for stable networks has $O(n)$ communication complexity). To adapt the complexity to the case at hand the second part of this paper uses *slide* to develop a systematic method to alleviate this drawback by combining a dynamic algorithm with a static one to get a new dynamic algorithm whose communication complexity matches that of the static algorithm if the network becomes static, and still works correctly even if the network topology never stabilizes. The method applies to on-line algorithms in the sense that these algorithms receive input values during their run, they never terminate and they produce a series of output values (like end-to-end communication algorithms). The complexity of such algorithms in dynamic networks is usually measured as the maximum cost they incur between any two successive output events. In order to capture in one complexity measure the complexity of the combined algorithm both when the network is dynamic and when topological changes cease, we give a two component amortized cost: The cost per topological change plus the cost per output (input) event. The former captures the cost of the algorithm when there are topological changes in the network, while the latter captures the cost in times when the network topology is stable. Given a static algorithm with communication complexity C_s and a dynamic algorithm, we construct a combined algorithm with amortized communication complexity of $O(n + C_s + m)$ per topological change plus $O(C_s)$ per output event.

The implementation of this method depends on the possibility to merge the outputs of two algorithms that in parallel solve the same problem, into one output stream. The merging itself is not systematic and depends on the specific problem being solved. Such a merging mechanism for the end-to-end problem was given in [AG88] and is used here to exemplify the method, resulting in an algorithm that runs in dynamic networks, and achieves $O(n)$ communication complexity if the network stabilizes.

1.1 Related work

Techniques The algorithms in this paper combine several techniques from recently reported research. The *slide* mechanism combines in an interesting way ideas from [AMS89] with a technique appearing in [MS80]. Combining *slide* with a concept introduced in [AFWZ88, AAF$^+$], we construct an end-to-end algorithm, the *majority* algorithm. This algorithm, in conjunction with the information dispersal algorithm (IDA) of Rabin [Rab89], yields our data dispersal algorithm.

In the second part of the paper we combine the *slide* protocol with a an idea introduced in [AM88], to generate a simple and systematic method for combining a dynamic version of an algorithm with a static version. The method is exemplified on the end-to-end problem by combining the bootstrap mechanism for dynamic networks of [AG91] with a static broadcast and echo along a path, to create a dynamic algorithm, that automatically adjusts its communication complexity to the network conditions.

The end-to-end problem One approach to solve the end-to-end problem in a dynamic network is to deliver the data items over a fixed path and to construct a new path every time a link on the path fails. Implementations of this approach appear in [Fin79, AAG87, AS88, AAM89]. This approach requires, however, the whole network to stabilize for a period of time allowing the construction of the path and the communication over it [AAG87, AS88, AAM89], or at least requires links forming some path between the sender and the receiver to be operational for such a period of time [AGH90].

In [AE86] it is stated that the *eventual connectivity* fairness condition, namely that there is no edge-cut all of whose links are permanently down, is sufficient for communication between the sender and the receiver. The problem is solved in [AE86, Vis83] under this condition by using *sequence numbers* to number the data items sent by the sender. This approach yields theoretically `unbounded` algorithms, as the sequence numbers grow with the number of data items transmitted.

In recent years a sequence of works gave bounded, and increasingly efficient, solutions to the problem. The first `bounded` end-to-end communication algorithm under the *eventual connectivity* fairness condition was presented in [AG88]; however, this solution has exponential message complexity per data item. Subsequently, in [AMS89], the first `polynomial` message complexity algorithm was presented, having bit communication complexity of $O(n^9 + mD)$. Recently, a new resynchronization protocol for dynamic networks was developed in [AG91] and was used to construct an end-to-end communication algorithm whose bit communication complexity is $O(nm \log n + mD)$.

Outline: We start by introducing the model in Section 2 and give a description of the *slide* and the end-to-end communication protocols in Section 3; An outline of proof of correctness and complexity analysis is given in the appendix. Section 4 contains the method to turn any dynamic algorithm into a dynamic algorithm that adjusts its communication complexity to network conditions.

2 The Model

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

Each link has *bounded* capacity. By capacity we refer to the maximal number of messages in transit on a given link at a given time. The bounded capacity can be either constant, or a function of the network size. For clarity of description, we consider networks in which each link has $O(n)$ capacity. As noted in the analysis our algorithms can be easily modified to conform to the constant capacity model without effecting their complexities.

The communication over the links obey the FIFO discipline, and no bound on the transmission delay time is known. A directed link is *non-viable* if starting from some message it does not deliver any message. The transmission delay time for this message and the subsequent messages sent on this link is considered infinite. A directed link is *viable* if any message sent over it is eventually delivered. Similarly, an undirected link is *viable* if both of its directed links are viable. (We assume that link that fails, fails in both directions.) Two nodes in the network are *eventually connected* if there is a path of viable links connecting them.

The model defined here is called the "∞-delay model" in [AG88], and as stated there the model of failing and recovering links [AAG87] can easily be reduced to it. When combining dynamic algorithms with static algorithms (Section 4) we consider the model of [AAG87] in which topological changes are part of the input to the algorithm. The reduction of the changing topology model of [AAG87] to the ∞-delay model, adds at most $O(1)$ messages per topological change to the protocol's communication complexity in the ∞-delay model.

2.1 The End to End Communication Problem

An end-to-end communication algorithm transmits a sequence of data items from a sender to a receiver. The data items are input to the sender on-line, that is the sequence is not known at the beginning of the operation of the algorithm and the data items are input one by one.

The algorithm must satisfy the following properties:

- **Safety:** At any time the sequence of the data items output by the receiver is a prefix of the sequence of data items input by the sender.
- **Liveness:** If the sender and the receiver are eventually connected, then every data item input by the sender will eventually be output by the receiver.

The Complexity Measures Since the algorithm transmits a sequence of data items input one by one, the complexity analysis should evaluate the performance of the algorithm per data item. This evaluation is done by measuring the cost of the algorithm between any two successive data item output events at the receiver.

We consider the following complexity measures: *Message Complexity:* The number of messages transmitted in the network in the worst case between any two successive output events at the receiver; *Bit Communication Complexity:* The number of bits transmitted in the network in the worst case between any two successive output events at the receiver; *Space Complexity:* The maximal amount of space required at each node, per incident link, measured in bits of memory.

3 Description of the Algorithms

When presenting the code of the algorithms we use the *guarded commands* language of Dijkstra [DF88], where the code of each process is in the form $G_1 \rightarrow A_1 \Box G_2 \rightarrow A_2 \Box \ldots G_l \rightarrow A_l \Box$. 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 following sections describe the various algorithms; An outline of the correctness proofs and analysis is added in the appendix.

3.1 The *slide* Protocol

In this protocol one node, the sender, inputs tokens into the network, and one node, the receiver, outputs the tokens. The protocol guarantees that tokens are neither lost nor duplicated, and that the total number of tokens in the network at any given time is bounded. If the sender and the receiver are eventually connected, then eventually the input of a new token to the network is enabled at the sender. However, tokens are output at the receiver not necessarily in the order they are input at the sender. Thus the *slide* establishes between the sender and the receiver a non-fifo, bounded capacity virtual communication link that neither loses nor duplicates messages.

The *slide* protocol is implemented by storing and transferring tokens between the nodes of the network as follows: Each undirected link is considered as two antiparallel links. Each node maintains for each incident incoming link a pile of slots numbered 1 to n. Each slot has room for one token, and each pile is used to store tokens arriving on the link associated with it; Tokens from a pile can be forwarded over any outgoing link. The crux of the protocol is that a token is sent from any slot i at node v to slot j at the (v,u) pile at node u, only if $j < i$. To this end the nodes maintain for each outgoing link a variable, called *bound*, holding an upper bound to 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 rule. Every time a token is removed from a pile, a signal to this effect is sent over the incoming link associated with the pile. Since the only source of tokens for a specific pile is the node 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 pile on the other side of the link plus the number of tokens in transit over the link.

New tokens enter the network only at the sender, to a special slot at level n, and only when this slot is vacant. The receiver has always a vacant slot of level 1, and removes and outputs any token it receives. If the sender and the receiver are eventually connected, then

```
Initialization ⟶
    for every incident link e
            bound[e]:=1;
            top[e]:=0;
□
Receive SIGNAL 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 SIGNAL on e';
    top[e']:=top[e']-1;
    bound[e]:=bound[e]+1;
□
```

a: ordinary's node code

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

b: additions for the sender

```
Receive TOKEN(data) on e ⟶
    output(data);
    send SIGNAL on e ;
□
```

c: receiver's code

Figure 1: The *slide*

eventually the special slot at the sender is vacant. Thus the tokens slide in the network from the sender to the receiver by sliding from higher numbered slots to lower numbered slots as they travel over links. Clearly, each token can make at most n hops in the network. Since the protocol maintains for each link $2n$ slots, and (as we prove in the sequel) at most $2n$ tokens can be in transit on each link at any given time, the total number of tokens in the network is at most $O(nm)$. This is the capacity of the *slide*, denoted $\mathcal{C}ap$.

The code of the algorithm is given in Figure 1; It uses two types of messages: TOKEN messages which are used to transfer the tokens themselves, and SIGNAL messages that are used to inform over a link that a token from the pile associated with it was removed from the pile. 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. The code of the sender is the code of an ordinary node and in addition the code appearing in figure 1b. Note that the sender is disabled to input new tokens if input_pile[n] is not vacant. The receiver does not send tokens and outputs any token it receives. Therefore, its code is restricted to the code appearing in figure 1c.

In the following subsections we present two solutions to the problem of end-to-end communication in dynamic networks, that use the *slide* as a building block. The second solution, the *data dispersal* algorithm, is our main result that achieves linear communication cost for large enough data items.

3.2 The Majority Algorithm

Given the *slide* we construct a simple end-to-end communication algorithm by operating the *slide* from the sender (S) to the receiver (R) and combining it with an idea of [AFWZ88, AAF+]. Whenever S wishes to send a data-item to R it sends consecutively $2{\cdot}\mathcal{C}ap+1$ duplicates of the data item to R using the *slide*. To receive the first data item R waits for $\mathcal{C}ap+1$ data items and outputs one of them. For each subsequent data item it waits for $2{\cdot}\mathcal{C}ap+1$ data items, takes the majority of the values received, and outputs this value. Since no more than $\mathcal{C}ap$ data items can be delayed in the network at any given time, the majority of the data items received represent the next data item (see Theorem 18 in the appendix).

The sender's algorithm, described below in Figure 2, interfaces with the sender's algorithm of the *slide* protocol in a way that each token *sent* here is `input` by the *slide* sender, and each token `output` by the *slide* receiver is *received* by the receiver of the majority algorithm.

The bit communication complexity of this algorithm is $O(n^2mD)$, where D is the size of the data item.

We remark that by adding a toggle bit to each data item the protocol can be made robust against leftover

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

a: receiver's code

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

b: sender's code

```
Procedure check_and_output
    if first_item=true and |items-set| = Cap + 1 then
                /* first data item */
        output(any_data_item_of(items-set)) ;
        items_set := φ ;
        first_item:=false ;
    else if first_item=false and |items-set| = 2·Cap + 1
        then        /* all other data items */
            output(majority(items-set)) ;
            items_set := φ ;
        endif
    endif
```

c: procedure check_and_output

Figure 2: The Majority Algorithm

data items that can be in the network before the algorithm has started.

3.3 The Data Dispersal Algorithm

For the cases were the data items are large with respect to the size of the network, i.e. having size of $\Omega(nm \log n)$ bits, we construct an algorithm that achieves linear ($O(n)$) bit communication complexity. The same algorithm can also be used for smaller data items if the sender is allowed to wait for more than one data item and transmit several data items together.

In order to derive this algorithm, we combine Rabin's Information Dispersal Algorithm [Rab89] with the fact that the *slide* can delay only a bounded number of packets. Using the Information Dispersal Algorithm (IDA), the sender splits the data item into packets and sends them to the receiver over the *slide*. Since the IDA allows the construction of the data item from a subset of the packets, we can tolerate the loss of the bounded number of packets that the *slide* can hold.

The sender creates using the IDA $2 \cdot \mathcal{C}ap + 1$ packets, each of size $O(\frac{D}{\mathcal{C}ap+1})$ bits, where D is the size of the data item; The receiver is thus able to construct the full data item from only $\mathcal{C}ap + 1$ packets. The sender sends the $2 \cdot \mathcal{C}ap + 1$ packets, and as the *slide* can delay at most $\mathcal{C}ap$ of them the receiver will receive enough packets to reconstruct the data item. The only problem left is to make sure that the receiver does not use old delayed packets to reconstruct subsequently sent data items. To alleviate this problem the sender adds to the packets of each data item a label. The receiver outputs the first data item after calculating it from the first $\mathcal{C}ap + 1$ packets it receives; For each subsequent data item it waits for $2 \cdot \mathcal{C}ap + 1$ packets, checks which label has the majority among the labels in the packets, and uses only the packets having this label. For each new data item the sender must use a label that is not present in the network. Therefore 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 $\mathcal{C}ap$, $2\mathcal{C}ap+1$ different labels suffice.

In the code, presented in Figure 3, we use the subscripts $*_{S \to R}$ and $*_{R \to S}$ to denote operations with respect to the *slide* from the sender to the receiver and the *slide* from the receiver to the sender, respectively. The interface with the *slide* protocols is similar to this interface for the majority algorithm. $\mathcal{L}$ denotes a set of $2 \cdot \mathcal{C}ap + 1$ labels. Note that the function *extract* can extract an arbitrary member from the set it is applied to.

The bit communication complexity of the data dispersal algorithm is $O(nD)$, if it is applied to data items of size $\Omega(nm \log n)$ bits. If the algorithm is applied to smaller data items, it achieves an amortized bit communication complexity of $O(nD)$, by combining several data items together.

4 Combining Static and Dynamic Algorithms

Dynamic algorithms that are designed to operate correctly in eventually connected dynamic networks usually suffer the drawback that even if the network becomes static their communication complexity does not decrease. In this section we use a variation of the *slide* protocol, called *generalized slide* to present a systematic methodology to combine a static algorithm with a dynamic one into a single dynamic algorithm whose communication complexity matches that of the static algorithm if the network topology becomes static for a large enough period of time, and operates correctly even if the network topology changes frequently. The approach was first introduced in [AM88] in an ad-hoc manner for a specific problem, the topology update problem.

The essence of our method is a mechanism ensuring that the dynamic algorithm sends no more than k messages per topological change (i.e. its amortized communication complexity [AAG87, AM88] is $O(k)$ per topological change), where k is a design parameter. In particular, if topological changes cease, the dynamic algorithm grindes to a halt. To ensure that progress is made in that case (if the network topology stabilizes) we run in parallel a static algorithm used in conjunction with the reset procedure of [AAG87]. Since both algorithms run in parallel, their output should be merged into a single series of output events for the problem being considered. The implementation of the merging mechanism depends on the particular problem; As an example we describe below how to merge two algorithms in the context of the end-to-end communication problem.

4.1 Generalized *slide*

The *slide* is generalized to a distributed object which interacts with two sets of nodes, the *producers* and the *consumers*. Producers deposit tokens into the *slide* and consumers may consume tokens, thus extracting them from the *slide* locally. The generalized *slide* has the following properties: (1) At any time, the total number of tokens consumed is no more than the total number of tokens produced, (2) the total number of tokens stored in the *slide* at any given time, called the *capacity*, is bounded, (3) each token performs at most n hops in the network, and (4) if a *consumer* wishes to consume tokens and is eventually connected to a *producer* that infinitely many times produces tokens, then eventually the consumer will have a token to consume. We achieve

```
Initialization ⟶
    L̄ := L ;
    sending:=false;
    missing:= 0;
□
sending=false and missing ≤ 2·Cap ⟶
    data-item:=next input;
    l:=extract(L̄);
    using the IDA with parameters
          Cap + 1 and 2·Cap + 1,
          create packets number 1 to 2·Cap + 1;
    send_buffer:=⋃_{i=1}^{2Cap+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 L̄= L̄ ∪ l; endif
□
```

a: sender's code

```
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);
□
```

b: receiver's code

```
Procedure check_and_output
    if first_item=true and
          |packets-set|=Cap + 1 then
                                   /* first data item */
          using the IDA calculate the data item from the
            Cap + 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·Cap + 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 3: The Data Dispersal Algorithm

this by requiring strong fairness in token distribution as outlined below.

Briefly stated, given the *slide* of Section 3, the generalized *slide* is implemented as follows: An additional pile, with a single slot at height n, is added to each *producer* node. This pile is managed in the same way as the special pile of the sender in the original *slide*. Whenever this pile is vacant, the producer can locally add a token to the *slide*. A *consumer* node is allowed to extract from its piles tokens for consumption; This is in addition to its ability to send tokens to adjacent nodes. Whenever it extracts a token from a pile it has to report this event to the other end of the link associated with this pile, as if the token was sent over a link. The strong fairness property is achieved as follows: If a node can forward tokens on a particular link infinitely often, then tokens will be forwarded on the link infinitely often. Furthermore, we assume that there is a conceptual link from a slot in a pile to the slot below it, which transfers tokens down if the slot is empty. This conceptual link is treated as a regular link as far as fairness goes. Finally, a token goes down only one slot when traversing any link; More than one token can temporarily reside at the same slot. However, since the rules for sending tokens over a physical link remain the same, the total number of tokens in a pile is still at most n. The mechanism described above is a version of *slide* that avoids starvation. We will use a generalized *slide* where all nodes are both consumers and producers. Note that the *slide* of Section 3 is a generalized *slide* with one producer and one consumer.

4.2 Combining the Algorithms

To build the combined algorithm we run two separate algorithms, a dynamic one and a static one, both receiving inputs and delivering outputs to a combination mechanism that outputs a single output for the problem being solved. We assume that this mechanism uses at most a constant number of output events of the dynamic and the static algorithms in order to produce one output event for the whole algorithm, and that it can continue to generate output events even if one of the two algorithms halts. There are three major steps in building the combined algorithm:

1. Whenever a node senses a topological change on one of its incident links it produces a *resource token*; A node running the dynamic algorithm has to consume such a token for each message it sends. Thus, each topological change creates at most 2 *resource tokens*, one per each node adjacent to the topological change. These two tokens can account for at most two messages sent in the dynamic algorithm. The generalized slide is employed to spread around the *resource tokens*. If the new *resource token* cannot be accepted by the *slide* because the node's pile is full, the *resource token* is discarded. This ties the dynamic algorithm to the topological changes.

2. A *reset* protocol, [AAG87], is used to tie the static algorithm to the topological changes. That is, the static algorithm should abort its execution if topological changes occur. Moreover, the *local* reset of [AAG87, AAM89] guarantees that topological changes in remote areas of the network do not unnecessarily affect the static algorithm.

3. We now have two independent algorithms running in the network. The question still stands how to distribute the inputs among them, and more importantly, how to merge their outputs into one correct output. The answer to this question is dependent on the particular problem and algorithms used; We discuss in the next section, as an example, the case of the end-to-end communication problem.

We first have to argue that the combined algorithm is correct, that is, eventually a new output event of either the static or the dynamic algorithm will occur: If the network topology stabilizes then the static algorithm never aborts and output events occur. If the network never stabilizes, then eventually nodes running the dynamic algorithm will have a token to consume and the algorithm will make progress.

For problems where one can build an algorithm that employs two totally independent algorithms, uses at most a constant number of outputs of each one to produce an output for the problem, and can operate even if one of the two halts, we have the following:

Theorem 1 *Given a dynamic algorithm $\mathcal{A}_d$ and a static algorithm $\mathcal{A}_s$ with communication complexity C_s per output event, there is an algorithm that has the following amortized communication complexity: $O(n+C_s+m)$ messages per topological change plus $O(C_s)$ per output event, where n and m are the number of nodes and links in the network, respectively.*

Proof Sketch. All messages sent by the combined algorithm are sent by one of the following four components: the dynamic algorithm $\mathcal{A}_d$, the static algorithm $\mathcal{A}_s$, the generalized *slide* and the *reset* mechanism of [AAG87].

The dynamic algorithm consumes a *resource token* to send each message, therefore sending at most $O(1)$ messages per topological change. Each token of the *generalized slide* makes at most n hops in the network, thus the *generalized slide* has $O(n)$ amortized message complexity per topological change. The amortized communication complexity of the reset mechanism [AAG87] is $O(m)$ per topological change. For each C_s messages of the static algorithm either at least one output event of $\mathcal{A}_s$ occurs, or the output is purged because of a reset initiated by a topological change. Since after some constant number of output events of $\mathcal{A}_s$ there is an output event of the combined algorithm, any series of at most $O(C_s)$ messages of $\mathcal{A}_s$ can be charged to either an output event of the combined algorithm or to a topological change. □

4.3 Combined end-to-end

In this subsection, we outline the application of the combining methodology to create a dynamic end-to-end protocol that diverges into the static algorithm if the network is static for long stretches of time.

The problem of end-to-end communication from a sender to a receiver can be reduced to the problem of implementing two *probes* between these two processors, one in each direction (A probe implements the read of a remote variable); See [AG88]. Moreover, this paper shows how two independent probes in the *same* direction, can be merged into a single probe.

We use the combining methodology to solve the end-to-end problem as follows: We apply the methodology to static and dynamic probe algorithms, dispersing the inputs and merging the outputs as in [AG88], resulting in a combined probe (in one direction) from the sender to the receiver. Similarly, a corresponding combined probe is constructed in the opposite direction. The result is a pair of efficient probes, one in each direction, as is required (by the reduction of end-to-end to probe).

The static probe algorithm used is the obvious one–broadcast and echo along a fixed path. The dynamic algorithm is a flood forward and a flood back, in conjunction with the bootstrap mechanism of [AG91]. The end result is an end-to-end protocol whose amortized communication complexity is $O(n + m)$ messages per topological change plus $O(n)$ per data item. In particular, when the network topology stabilizes it achieves $O(n)$ message complexity.

Acknowledgments: We thank Baruch Awerbuch, Yishay Mansour, Michael Merritt, Michael Saks, Nir Shavit, and Gadi Taubenfeld for their cooperation and for helpful discussions.

References

[AAF+] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck. Reliable communication over unreliable channels.

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 358–370, October 1987.

[AAM89] Y. Afek, B. Awerbuch, and H. Moriel. A complexity preserving reset procedure. Technical Report MIT/LCS/TM-389, MIT, May 1989.

[AE86] B. Awerbuch and S. Even. Reliable broadcast protocols in unreliable networks. *NETWORKS*, 16(4):381–396, 1986. Previously titled "A Rigorous Treatment of a Communication Protocol: Broadcast as a Case Study.".

[AFWZ88] H. Attiya, M. J. Fischer, D. Wang, and L. D. Zuck. Reliable communication using unreliable channels. Manuscript, 1988.

[AG88] Y. Afek and E. Gafni. End-to-end communication in unreliabel networks. In *Proc. of the Seventh 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 Annual 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. of the Ninth Annual ACM Symp. on Principles of Distributed Computing (PODC)*, pages 189–203, August 1990.

[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 Annual Symp. on Foundation of Computer Science*, pages 358–363, October 1989.

[AP90] B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 503–513, October 1990.

[APSV91] B. Awerbuch, B. Patt-Shamir, and G. Verghese. Self-stabilization by local checking and correction. In *Proc. of the*

33rd IEEE Annual 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 Annual Symp. on Foundation of Computer Science*, pages 206–220, October 1988.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[BGP89] P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 410–415, October 1989.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63–75, January 1985.

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

[DS80] W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11-1:1–4, August 1980.

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

[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.

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

[Wec80] S. Wecker. Dna: the digital network architecture. *IEEE Transactions on Communication*, COM-28:510–526, April 1980.

A The *slide* protocol

A.1 Correctness Proof

Throughout the proof 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. $\mathcal{X}_v^t$).

Definition 2 *$tokens\_intransit_{u\to v}^t$ is the number of tokens in transit from u to v at time t.*

Definition 3 *$signal\_intransit_{u\to v}^t$ is the number of SIGNAL messages in transit from u to v at time t.*

Lemma 4 *At any time t and for any e, $e=(u,v)$,*
$$bound[e]_u^t - 1 = top[e]_v^t + tokens\_intransit_{u\to v}^t + signal\_intransit_{v\to u}^t\ .$$

Proof Sketch. By induction on the events that change any of the values participating in the equation.

Lemma 5 *A token received at node v from node u is stored at v in a slot at a lower level than the level it was stored in at u.*

Corollary 6 *Since new tokens enter the network into slot number n, every token can make at most n hops in the network.*

Lemma 7 *At any time t and for any e, $e=(u,v)$, $top[e]_v^t + tokens\_intransit_{u\to v}^t \le n$.*

Lemma 8 *At any time t there are no more than $2n$ messages in transit on any link.*

Note that the previous lemma shows that the protocol applies also in the model where links have constant (not just finite) capacity by having an $O(n)$ space buffer at the tail of each link and sending every message only after receiving an ACK for the previous one. As the space complexity of the protocol is anyhow $O(n)$, this change does not effect any of the complexities.

Lemma 9 *At any time t the number of tokens in the network is bounded by $O(nm)$. ($Cap = O(nm)$).*

Lemma 10 *In any time interval $[t_0, t_1]$, where* `new` *new tokens are added to the network, at most $O(n^2m + \texttt{new}\cdot n)$ token-passes can occur.*

Proof: By Lemma 9 the total number of tokens in the network at t_0 is bounded by $O(nm)$. By Lemma 6 each can make up to n hops in the network, thus contributing up to $O(n^2m)$ token passes. Any new token added in the time interval $[t_0, t_1]$ can also make up to n hops. □

Theorem 11 *If S and R are eventually connected, S will eventually input a new token.*

Proof: By way of contradiction assume that t is the last time the sender inputs a token.

As a result of Lemma 10 and as there is only one SIGNAL message per each token passing, there is a time $t' \geq t$ after which no token or SIGNAL message is 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} have been delivered.

By induction on the length of the viable path from v_i to R, we can show that v_i cannot have a token in level $> i$ after time t'' .

The receiver, v_0, has no tokens stored at all. Denote by e the link between v_i and v_{i-1} ($i \geq 1$), and assume the induction assumption that v_{i-1} has no token stored at level $> i-1$. Since at t'' all messages between v_{i-1} and v_i have arrived, by Lemma 4 $bound[e]^{t''}_{v_i} = top[e]^{t''}_{v_{i-1}} + 1$; by the induction assumption $top[e]^{t''}_{v_{i-1}} \leq i-1$, thus $bound[e]^{t''}_{v_i} \leq i$. As $t'' \geq t'$, no token is sent after t'', but according to the code this can happen only if v_i does not have tokens of level $i+1$ or more, proving the induction step.

Since $k < n$, S does not have a token of level n at t'', and by the code will enable input of a new token, contradicting the assumption that t is the last time S inputs a token. □

A.2 Complexity

Lemma 12 *The number of messages sent by* slide *in any time interval where* `new` *new tokens are input by the sender is bounded by* $O(n^2 m + \texttt{new} \cdot n)$.

Corollary 13 *The number of bits sent in the* slide *protocol in any time interval where* `new` *new tokens are input by the sender is bounded by* $O((n^2 m + \texttt{new} \cdot n)L)$, *where* L *is the maximal number of bits in a token.*

Lemma 14 *The space needed at each node is* $O(nL)$ *per incident link, where* L *is the maximal number of bits in a token.*

B The Majority Algorithm

B.1 Correctness Proof

Definition 15 $in^{[t,t']}$ *is the number of tokens deposited by the sender into the* slide *in the time interval* $[t, t']$.

Definition 16 $out^{[t,t']}$ *is the number of tokens received by the receiver from the* slide *in the time interval* $[t, t']$.

Definition 17 $delay^t$ *is the number of tokens delayed by the* slide *at time* t.

Theorem 18 (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_0 some time before the beginning of the execution of the algorithm, and 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 carrying I_k, $k > i$ could have been received before t_i. By the code, the total number of tokens that have been received by the receiver until time t_i is:

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

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

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

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 $\mathcal{C}ap$ tokens containing data item I_k, $k < i$ may be received in the interval of time $[t_{i-1}, t_i]$. This completes the proof of the safety property because together with the above it implies that from the $2 \cdot \mathcal{C}ap + 1$ tokens received in $[t_{i-1}, t_i]$ at least $\mathcal{C}ap + 1$ of them 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 the all other tokens. We have already proved that all the tokens received until t_{i-1} are *old* and that the total number of such tokens received by the receiver until t_{i-1} is $(2 \cdot \mathcal{C}ap + 1)(i-1) - \mathcal{C}ap$. Since the total number of *old* tokens ever sent by the sender is $(2 \cdot \mathcal{C}ap + 1)(i-1)$, at most $\mathcal{C}ap$ may be received by the receiver in the interval of time $[t_{i-1}, t_i]$. □

Theorem 19 (Liveness) *If the sender and the receiver are eventually connected, then the receiver will eventually output any data item input by the sender.*

B.2 Complexity

Lemma 20 *The message complexity of the majority algorithm is* $O(n^2 m)$.

Proof: Let T_i be the interval of time from the time O_{i-1} is output to the time O_i is output. Clearly in T_i the receiver receives $2 \cdot \mathcal{C}ap + 1$ tokens. Since the *slide* can hold at most $\mathcal{C}ap$ tokens, at most $3 \cdot \mathcal{C}ap + 1$ tokens are sent by the sender in T_i, and the lemma follows. □

Corollary 21 *The bit communication complexity of the majority algorithm is $O(n^2mD)$, where D is the size in bits of a data item.*

Lemma 22 *The space complexity of the majority algorithm is $O(nD)$, where D is the size in bits of a data-item.*

Proof: Each token sent in the majority algorithm consists of $O(D)$ bits. Applying this to Lemma 14, we get the space complexity of $O(nD)$ bits. $\square$

C The Data Dispersal Algorithm

C.1 Correctness Proof

Lemma 23 *Whenever the sender tries to extract a label from $\overline{\mathcal{L}}$, $\overline{\mathcal{L}}$ is not empty.*

Theorem 24 (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 the i'th data item, and denote by l_i the label added to the $2{\cdot}\mathcal{C}ap+1$ packets calculated from I_i at the sender. By the code, the tokens used at t_i to calculate the i'th data item at the receiver are the $2\cdot\mathcal{C}ap+1$ tokens received by it in the time interval $[t_{i-1}, t_i]$. By the same arguments used for the safety proof of the majority algorithm (Theorem 18), at least $\mathcal{C}ap+1$ of these tokens contain the label l_i; thus the majority of labels will be l_i, and the receiver will calculate the i'th data item from the tokens containing l_i. Since at the time the sender extracts l_i from $\overline{\mathcal{L}}$ there is no token containing it in the network, the IDA at the receiver at t_i will use only packets calculated from I_i at the sender. As noted before the receiver has at least $\mathcal{C}ap+1$ such packets at t_i, and the IDA will correctly calculate I_i at t_i. Thus $O_i = I_i$ for any i. $\square$

Lemma 25 *For any time t $missing \leq 4\cdot\mathcal{C}ap+1$.*

Proof: The variable *missing* is incremented when a token is extracted from *send_buffer*. The only event where tokens are added to *send_buffer* is when $2\mathcal{C}ap+1$ tokens are added to it when it is empty and $missing \leq 2\cdot\mathcal{C}ap$. Thus at any time $missing \leq 4\cdot\mathcal{C}ap+1$. $\square$

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

Lemma 26 *If the sender and the receiver are eventually connected, there is no dead-lock at the sender (eventually $missing \leq 2\cdot\mathcal{C}ap$).*

Theorem 27 (Liveness) *If the sender and the receiver are eventually connected, then the receiver will eventually output any data item input by the sender.*

C.2 Complexity

Lemma 28 *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 *slide* from the sender to the receiver the same notation as in Section B.1. By the code $out^{[t_i,t_{i+1}]} = 2\cdot\mathcal{C}ap+1$, and $0 \leq delay^t \leq \mathcal{C}ap$, thus $\mathcal{C}ap+1 \leq in^{[t_i,t_{i+1}]} \leq 3\cdot\mathcal{C}ap+1$. By Lemma 9 $\mathcal{C}ap = O(nm)$, and applying this to Lemma 12 yields a message complexity of $O(n^2m)$ for the *slide* from the sender to the receiver.

The tokens that are sent through the *slide* from the receiver to the sender 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 25 the receiver stores at any time at most $4\cdot\mathcal{C}ap+1$ tokens. In the worst case all of them are in *tokens_to_return* at t_i. Thus at most $4\mathcal{C}ap+1$ tokens are input to this *slide* in the time interval $[t_i, t_{i+1}]$. Applying this to the results of Lemmas 9 and 12 we obtain a message complexity of $O(n^2m)$ for this *slide*.

Combing the two *slide* protocols we obtain a message complexity of $O(n^2m)$ for the data dispersal algorithm. $\square$

Corollary 29 *The bit communication complexity of the data dispersal algorithm is $O(nD)$, where D is the size in bits of a data item.*

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

Lemma 30 *The space complexity of the data dispersal algorithm 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 $O(\frac{D}{\mathcal{C}ap+1}+\log n)$. By applying Lemma 14 we get the space complexity of $O(n(\frac{D}{\mathcal{C}ap+1}+\log n))$. Since $\mathcal{C}ap = O(nm)$ the space complexity is $O(\frac{D}{m}+n\log n)$. $\square$